Problem-solving Systematization: Introducing Formal Methods in Basic Education[*][†]

Júlia Veiga da Silva[1], Braz Araujo da Silva Junior[1], Luciana Foss[1], Simone André da Costa Cavalheiro[1]

[1]Fundamentals of Computing Laboratory – Federal University of Pelotas (UFPel)
CEP 96.010-610 – Pelotas – RS – Brazil

`{jvsilva,badsjunior,lfoss,simone.costa}@inf.ufpel.edu.br`

Abstract. *Widely employed within critical systems, Formal Methods for specification and verification have gained significance in a world where computer systems continue to expand in scale. Teaching Formal Methods at the higher education level has long been accompanied by well-documented challenges. However, since it is often considered an advanced subject within software engineering, it is rarely included in basic education. Targeting this unconventional audience, this paper explores a new approach to formal specification based on the systematization of problem-solving. An example of recursion elimination is presented.*

1. Introduction

With the evolution in size and complexity of computer systems, which have become critical and intolerant of errors, the necessity to develop more precise specifications to address this complexity and ensure the correctness of these systems has become apparent. In this context, **F**ormal **M**ethods (**FM**) are techniques that gained prominence for providing rigorous and mathematical formalisms for specifying and verifying systems. These methods can be seen as the formalization and practical application of a scientific approach to programming, a practice that predates even the field of software development itself [Roggenbach et al. 2021].

Computing education is currently undergoing a significant global transformation. It is shifting from a primary focus on technical aspects in higher education, such as programming, to a more comprehensive curriculum that emphasizes problem-solving skills. This shift is encapsulated by the concept of **C**omputational **T**hinking (**CT**), which reflects the perspective that computing extends beyond mere programming and offers much more than just technical coding skills. It emphasizes the idea that individuals, not limited to computer scientists, can benefit from developing problem-solving abilities related to computing [Wing 2006]. These abilities include skills such as abstraction, problem decomposition, algorithmic thinking, data practices, and automation.

Although computing education has been revitalized with the rise of CT and has expanded into basic education, its efforts have predominantly focused on areas such as visual programming, like Scratch [Resnick et al. 2009], games [Krath et al. 2021], and robotics

[*]Work concluded.

[†]Undergraduate student; Graduate student; Professor; Professor.

[Yang et al. 2020]. Meanwhile, **T**heoretical **C**omputer **S**cience (**TCS**) has received relatively little attention in education. In a systematic review of TCS in basic education [Silva Junior et al. 2021a], out of the 17 studies analyzed, only a few have delved into TCS, despite its potential in developing analytical skills and fostering critical thinking. These studies covered a range of topics, including the use of digital tools, traditional lectures, and practical activities unrelated to computers. It is noteworthy that many of these studies emphasized problem-solving as a central element of their educational proposals. However, it becomes evident that TCS is underrepresented compared to more popular approaches.

Given this context, this paper proposes the use of computing fundamentals and FM to develop computing concepts. Our proposal is to present a technique for identifying and resolving an infinite loop problem in a game through problem-solving systematization using **G**raph **G**rammar (**GG**). The rest of this paper is organized as follows: Section 2 discusses how these fundamentals have been explored in studies that approached TCS in basic education; Section 3 presents how we can approach FM fundamentals with children using an example; Section 4 concludes the paper by presenting future directions for this research.

2. Supporting Tools

Although most of the efforts in the field of computing education in basic education do not focus on topics related to TCS, there are some supporting tools that illustrate how the fundamentals of FM have been introduced to children and adolescents.

2.1. Kara

Commonly taught as an introduction to the theoretical foundations of computer science, formal languages represent sets of strings/words (sequences of symbols) that have a specific structure or pattern. They are usually presented alongside their respective grammar, which contains the set of rules to produce each word of the language; and the automaton, an abstract machine that computes those rules. A **F**inite **S**tate **M**achine (**FSM**) is an automaton with a fixed number of states that can only be in one state at a time. The machine reads an input symbol and, based on its current state and the input symbol, transitions to a new state according to a set of rules.

In this context, Kara [Reichert 2003] is an educational software system that allows students to program a virtual ladybug based on an FSM. They have a set of commands (turn left or right, move ahead, collect or lay down leaves) that they can sequence for each state, leading to the next state at the end of the sequence. By default, when in a given state, it will run the sequence and go to the next state. However, students have the option to branch the execution according to inputs coming from the “sensors” of the ladybug. For instance, Figure 1 shows an FSM with four sequences for the state “run” where the input branching them is determined by checking conditions: whether there is a tree in front of the ladybug and whether there is a leaf under it. For each possible outcome of those conditions, the student defines a different sequence and the state it should go to. Students receive complete, informal specifications as descriptions of tasks and are not required to formalize them or prove anything for their solution. There is also no mention of discussing the features (e.g., strengths and weaknesses) of the model itself (FSM).

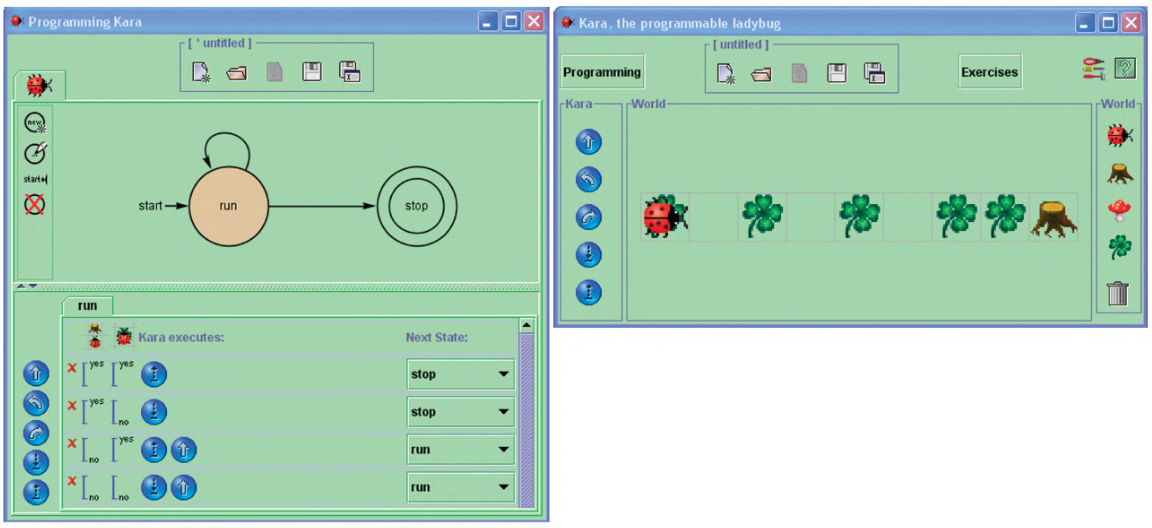

Figure 1. Kara programming environment (left) and board (right). Source: [Kiesmüller 2009].

2.2. Automata Puzzle

An automata puzzle game was created to introduce fundamental concepts of automata to primary and lower secondary school children (9–12 years old) [Isayama et al. 2016]. The tool was designed with a focus on **D**eterministic **F**inite **A**utomatons (**DFA**), which are FSMs where for each state and input symbol, there is exactly one next state. To avoid the use of mathematical notations for the formal definition of DFA, they represented the concepts as follows: the alphabet was portrayed as a set of materials; the transition function was depicted as a diagram; the input sequence was represented as a "recipe"; and execution was likened to a robot that reads the recipe and follows de diagram accordingly.

The game consists of several phases, including labeling questions, where students must define the transitions for a given DFA to recognize the provided recipes (as shown in Figure 2, left); and recognition questions, where students must determine whether a given DFA recognizes a given recipe or not (as in Figure 2, right). The game also includes "bad recipes" to engage with the reverse logic, i.e., sequences that should NOT be recognized by the DFA. To assist users in solving the problems, the game offers three classes of "walk-through" hints: path matching hints for labeling questions, which suggest that the user looks for recipes of a certain size that only one path in the diagram corresponds to, and then label the transitions of the path with the sequence of the recipe; label matching hints, which recommend comparing the last materials in recipes to the terminal arrows in diagrams; and deterministic property hints, which remind the user that a given state cannot have two transitions labeled with the same material leaving it.

2.3. GrameStation

GrameStation [Silva Junior et al. 2021b] is a tool based on GG used for creating and running games modeled this way. GG is a formal language that generalizes grammars by replacing strings with graphs [Rozenberg 1997]. Graphs are essentially composed of dots (vertices) and arrows (edges) representing elements and their relations. If you embellish them, you can end up with a grammar that generates scenes (images) instead of

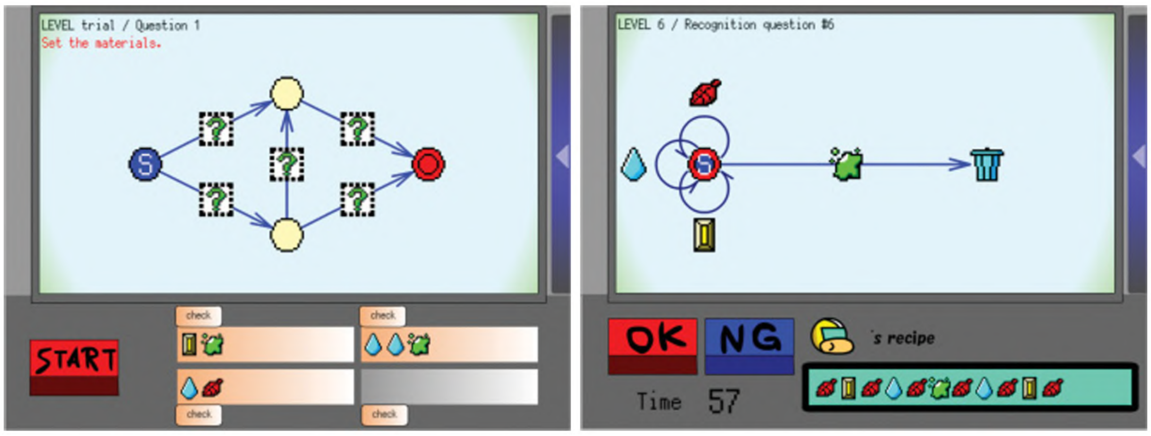

Figure 2. Automata Puzzle game labeling (left) and recognition (right) questions. Source: [Isayama et al. 2016].

words (text). A GG simulates a system using states (snapshots of the system) represented by graphs and events (transitions between states) defined by graph transformation rules. Therefore, a GG consists of a type graph, which specifies elements and their relations, ensuring that all elements in the GG have a type and are only allowed to have relations specified in the type graph; a start graph, defining the initial conditions; and a set of rules that define changes in state. For instance, Figure 3 illustrates the type graph and the start graph of the Pac-Man game as a GG. The type graph (left) declares the existence of Pac-Man, ghosts, fruits, places (grey dots), a score counter (pink triangle), and the relations between these elements. The start graph (right) shows one Pac-Man with a score of zero points, one ghost, and three fruits in a 4x3 arrangement of places.

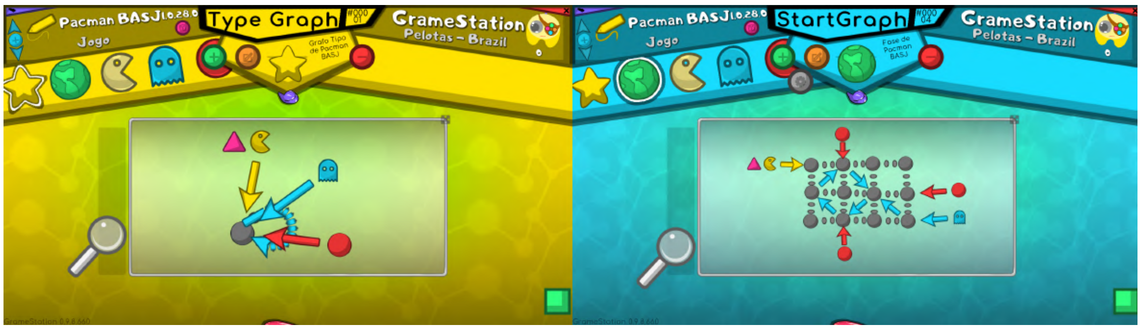

Figure 3. Type graph (left) and start graph (right) for the Pac-Man game in GrameStation.

To generalize the concept of numerical functions, which map numbers from a domain set to numbers in a codomain set, GG uses graph morphisms: structure-preserving mappings of elements (vertices, edges, and attributes) from a domain graph to elements in a codomain graph, preserving source and target edges. Rules consist of pairs of graph morphisms that indicate which elements will be created, deleted, or preserved when the rule is applied. To simplify, we can explain rules by focusing on the two target graphs of the morphisms: the **L**eft **H**and **S**ide (**LHS**) and the **R**ight **H**and **S**ide (**RHS**). They represent, respectively, the condition and consequence. In other words, they describe the situation before (LHS) and after (RHS) an event that alters the system's state (rule application). Elements present in both graphs will be preserved throughout the states, elements only in the LHS will be deleted, remaining in the previous state, and elements only in the RHS will be created, appearing in the future state. So, if one wants a specific event to occur (applying a rule) to transform a particular situation (LHS) into a new one (RHS), one must first find the situation in its current state. This involves mapping each element from the LHS to an element in the state graph. This mapping (morphism) is referred to as a *match* and must adhere to the element types, as well as the source and target of each edge.

Returning to our example, the set of rules for Pac-Man (Figure 4) includes *Pac Move*, *Ghost Move*, *Pac Eat*, and *Ghost Eat*. The rules are represented by a pair of graphs connected by an arrow. In Pac Move rule, for example, the LHS defines the condition for applying the rule: having a Pac-Man in a place that connects to another one. The RHS defines the consequence of this rule: the connection of Pac-Man is removed from its initial place and is restored at the next one.

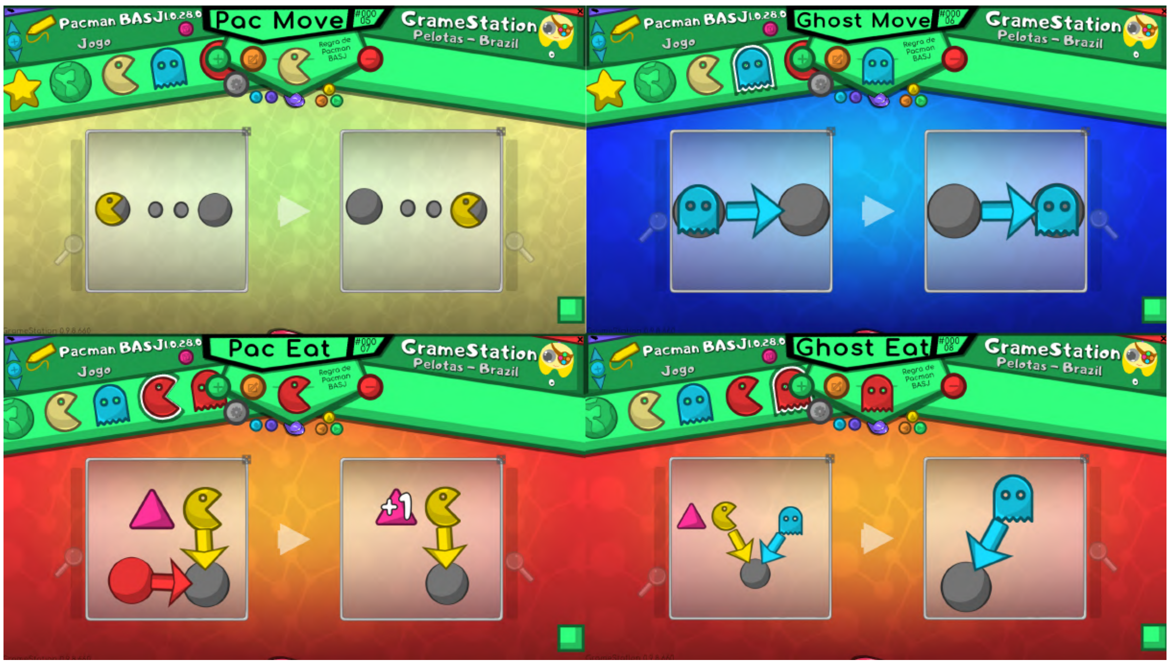

Figure 4. Rules Pac Move (left, top), Ghost Move (right, top), Pac Eat (left, bottom), and Ghost Eat (right, bottom) in GrameStation.

3. Our Approach

Considering an adapted version of a game from the Super Mario franchise, specified as GG in the GrameStation tool. The game in question consists of a set of rules – rules for movement, interaction, victory, defeat, etc. Among these rules, there is one that allows the main character, Mario, to jump on the cloud that is part of the game environment. For this purpose, two rules have been specified (Figure 5 and Figure 6).

However, there is a problem: if this action is selected during the game, the only thing Mario will be able to do is keep jumping on the cloud. With these two rules, the loop exists because the character Mario always returns to the original state – the left side of the first rule is generated again, reconstructing the match, i.e., the original match

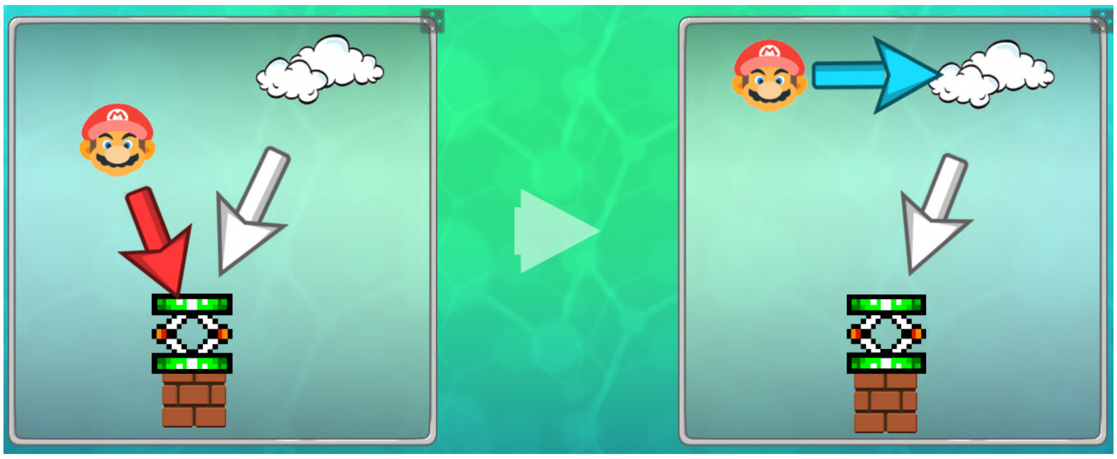

Figure 5. First rule that generates the infinite loop in the game.

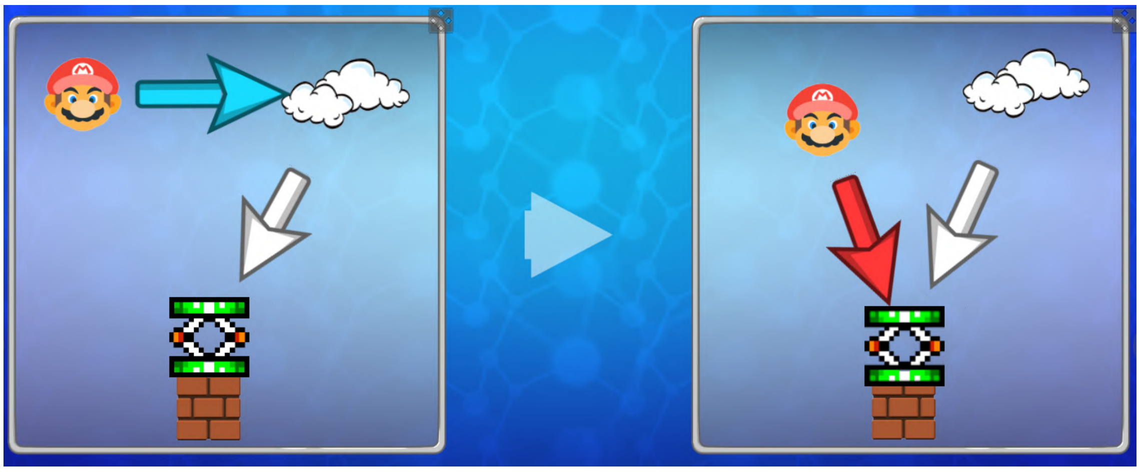

Figure 6. Second rule that generates the infinite loop in the game.

can be applied again with these two rules. Thus, this loop creates an infinite recursion. Recursion is a fundamental concept in the field of computing and programming, based on the idea that a function can call itself as part of its problem-solving process. This allows complex problems to be divided into smaller, simpler problems, making resolution easier. Therefore, when a recursive behavior exists, it is important to ensure that it terminates and, in this context, to stop the recursion it is necessary to add a rule that can be chosen to interrupt this infinite loop (Figure 7). The new rule interrupts the cycle and allows the character Mario to move forward to the next blocks in the scenario.

In this sense, in any grammar, it's possible to identify that a set of rules is in a loop if the left-hand side of the first rule is the same as the right-hand side of the second rule. In this way, GG helps recognize when recursion occurs as follows: if the right-hand side of one rule is equal to the left-hand side of another rule, then this rule is likely to enter into a loop. Thus, to break the loop, the right-hand side of one of the rules cannot contain the left-hand side of the other (i.e., it cannot have all the elements of the left-hand side of the other). This is not an exhaustive method that will guarantee that all cases of recursion are characterized in this way; but it makes it clear that if the case exhibits these characteristics, it will result in recursion. In this way, since GGs are visual, it is possible,

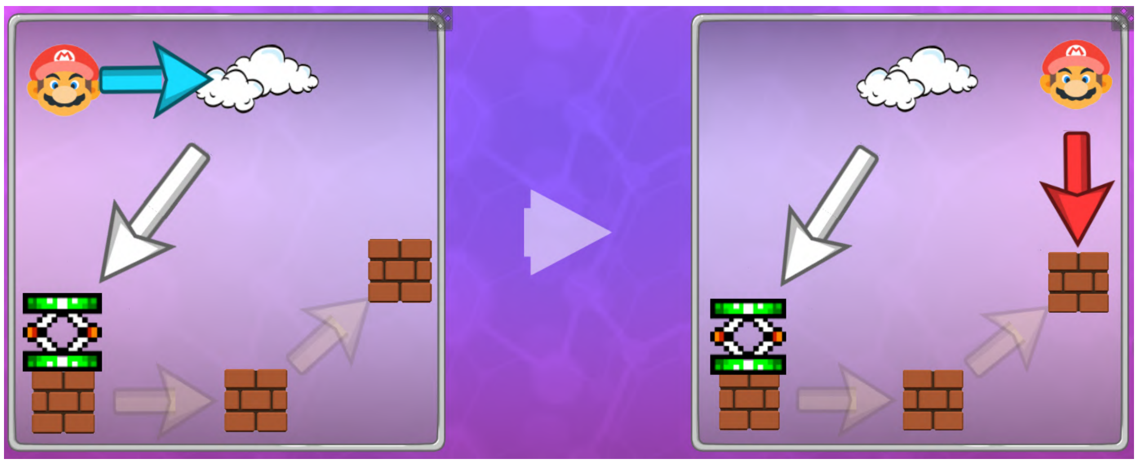

Figure 7. Rule created to stop the game's infinite loop.

initially, to recognize the existence of recursion by observing images and checking for visual similarities between them – afterward, you can verify element by element to ensure that it is indeed this situation.

4. Conclusion

This work presented initial steps for the introduction of FM in basic education through the problem-solving systematization. It was proposed to use GG to introduce computer science concepts. In this work, we address recursion, which appears in a "natural" way in grammars – as it is the available form to create loops/repetitions in this context. Moreover, we use as an example a problem (recursion) that is identified and solved using FM, and we manage to present essentially the same problem-solving procedure in a simplified manner for children. This indicates that it is possible to work with these concepts at a more simplified level in basic education, introducing students to the idea that there are techniques that can be followed and applied to various types of problems, regardless of their application domain and specific details.

Initiatives like this are particularly important in the current context of Brazil, given the addition of computing skills as a complement to the National Common Curricular Base (BNCC). Following this same idea of introducing FM into basic education, other concepts and skills will be addressed in the next steps of this work. For example, we can mention the skill "(EF08CO05) Understanding the concepts of parallelism, concurrency, and distributed storage/processing" from the BNCC. From this skill, it is possible to explore the concepts of parallelism and concurrency using GG. We propose providing specified games and challenging students to identify parallel and/or concurrent actions based on specific game states. Another skill could be "(EM13CO02) Exploring and constructing problem solutions through refinements, employing various levels of abstraction from specification to implementation". In this case, the activity could involve specifying a predefined game with students, starting in natural language (before and after actions) and gradually formalizing it as a graph. This could be an activity where the teacher initially describes actions in natural language, and students then select which graph rules actually represent those actions, evaluating the consequences of using one rule over another in the game's specification through testing in GrameStation.

References

Isayama, D., Ishiyama, M., Relator, R., and Yamazaki, K. (2016). Computer Science Education for Primary and Lower Secondary School Students: Teaching the Concept of Automata. *ACM Transactions on Computing Education (TOCE)*, 17(1):1–28.

Kiesmüller, U. (2009). Diagnosing Learners' Problem-solving Strategies Using Learning Environments with Algorithmic Problems in Secondary Eeducation. *ACM Transactions on Computing Education (TOCE)*, 9(3):1–26.

Krath, J., Schürmann, L., and von Korflesch, H. F. (2021). Revealing the Theoretical Basis of Gamification: A Systematic Review and Analysis of Ttheory in Research on Gamification, Serious Games and Game-based Learning. *Computers in Human Behavior*, 125:106963.

Reichert, R. (2003). *Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science*. PhD thesis, ETH Zurich.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al. (2009). Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67.

Roggenbach, M., Cerone, A., Schlingloff, B.-H., Schneider, G., and Shaikh, S. A. (2021). *Formal Methods for Software Engineering*. Springer.

Rozenberg, G. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World scientific.

Silva Junior, B. A., Cavalheiro, S. A., and Foss, L. (2021a). Theoretical Computer Science in Basic Education: A Systematic Review. In *Anais do VI Workshop-Escola de Informática Teórica*, pages 133–140. SBC.

Silva Junior, B. A., Cavalheiro, S. A. C., and Foss, L. (2021b). GrameStation: Specifying Games with Graphs. In *Anais do XXXII Simpósio Brasileiro de Informática na Educação*, pages 499–511, Porto Alegre, RS, Brasil. SBC.

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

Yang, K., Liu, X., and Chen, G. (2020). The Influence of Robots on Students' Computational Thinking: A Literature Review. *International Journal of Information and Education Technology*, 10(8):627–631.