GrameStation: Specifying Games with Graphs *

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

[1]Programa de Pós-Graduação em Computação - Universidade Federal de Pelotas
CEP 96.010-610 - Pelotas - RS - Brazil

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

Abstract. *This paper presents a platform for creating games using graphs. The proposed game engine is based on a mathematical formalism called Graph Grammar. It aims to rescue, within computer science education, the stage of specification, that precedes programming. The proposal is aligned to the trends of the problem-solving focus, development of computational thinking, use of visual languages, game-related environments and the maker movement. The structure of the platform, the creation and execution of an example game are described and a brief discussion about specification in computer science education is given.*

1. Introduction

The undeniable impact of **Computer Science (CS)** on everyday life has brought major efforts to make its education available to everyone. As the CS education progressed, the perception that computing is not just coding and it should rather be focused in problem-solving skills has grown and conquered ground. A milestone of this progress in the scientific community is a viewpoint recalling the term **Computational Thinking (CT)** and advocating it includes general purpose skills that should be learned by everyone, not just CS professionals [Wing 2006]. Some popular and successful approaches of teaching/learning CS and fostering CT skills are visual programming activities [Hu et al. 2021]; and gamified programming environments/programming games [Lindberg et al. 2019]. Which are often aligned with the Maker Culture, putting the learners as creators, rather than just consumers [Martin 2015].

Aligned with these ideas of being about more than just coding, having problem-solving as focus, using visual languages, being in game-related environments, and treating students as makers, we step back from programming to specification by proposing a new approach for CS education: using **Graph Grammars (GG)** to create games. A GG is a formal language used in software engineering for formal specification and verification of software, a modeling stage that precedes the implementation/coding of the actual software. We built a game engine based on GG, the **GrameStation**, to allow the use of this new approach while following successful trends in CS education. Previous experiments with GG in k-12 led us to the development of the game engine by showing that: despite being formal, GG can be friendly (presented without heavy use of mathematical formulas and notation) due to be visual [Silva Junior et al. 2017]; GG

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, do MCTIC/CNPq (Rede Sacci), da SMED/Pelotas e da PREC e PRPPG/UFPel.

concepts (concrete) can be more easily assessed and then associated with CT skills (abstract) [Silva Junior et al. 2019a]; a GG activity can provide data useful for powering assessment if automated [Silva Junior et al. 2018]; and virtual games interfaces and mechanics are able to ease and reinforce the learning of GG [Silva Junior et al. 2019b].

The rest of this paper is organized as follows. The Section 2 provides the theoretical background and related work, situating the paper within CS education, games in education and the maker movement, as well as presenting basic notions of GG. The section 3 presents the GG-based game engine, its foundational concepts, how to build and play games using it and the resources for guiding users added to the platform. The section 4 concludes the paper with the discussion about what changes when moving from programming to specification and how GG are related to CT.

2. Theoretical Background and Related Work

2.1. Computer Science Education

It is a recurring concern not to equate CS to programming or coding. Yet, these terms have been used interchangeably even inside CS education research. For [Armoni 2016], the differentiation of these terms is important: **coding** is the most concrete of the terms, it refers to the making of code, a collection of information represented (coded) in a certain way; **programming** refers to the making of programs, abstractions of what is coded, entities that someone (a programmer) created or will create, and can be run or executed; and **CS** is the systematic study of algorithmic processes that describe and transform information; their theory, analysis, design, efficiency, implementation and application. Although the distinction is important, programming and machines are essential for CS education and have been the center of it for a long time now [Denning and Tedre 2021].

A **Systematic Literature Review (SLR)** captured the main challenges faced by introductory programming students in higher education [Medeiros et al. 2018]: **problem-formulation**, understanding and conceptualizing the problem being addressed; **abstraction**, dealing with concepts that cannot be easily related to a real-life object, such as variables, data types and memory addresses; **algorithmic reasoning**, organizing thoughts in a systematic, well defined way, for a machine to be able to execute it; **syntax**, transforming an informal solution or even a pseudo-code into a syntactically correct program; **control and data structures**, selecting the best fit for a given problem according to their properties; **motivation and engagement**, being personally interested and willing to learn.

We can trace a relation of those challenges with the classification of programming languages in generations: the **first** comprises machine languages, which are binary code strictly bound to the hardware; the **second** comprises assembly, which are languages with lower number of instructions that represent the first abstraction to machine languages; the **third** comprises high-level languages, which are machine independent languages such as C++ and Java; and the **fourth** generation comprises declarative languages, which are languages that allow a programmer to specify what should be achieved, rather than how a goal should be achieved [Gaggioli 2017]. The lower the generation, the higher are the concerns with syntax, control and data structures. The lower the generation, the more specific and strict the problem-formulation and algorithmic reasoning must be. The higher the generation, the higher the abstraction, meaning the closer to human natural languages. At last, the higher the generation, the higher the motivation and engagement. This last

statement is yet to be proven, but we can see this trend in CS education, as can be implied by the rest of this section.

Following this trend of elevating the level of abstraction and minimizing syntax concerns, the **Low-Code Development (LCD)** rose. The LCD is described by its practitioners as a programming environment where the coding effort is low [Luo et al. 2021]. It is meant to be non-professional programmers friendly, as in visual programming (drag and drop) and what you see is what you get (WYSIWYG) platforms. A study gathered the perceptions of LCD practitioners from two popular online developer communities, Stack Overflow and Reddit [Luo et al. 2021]. The study collected a series of benefits of LCD from its practitioners' perspectives, among them are: faster development; ease of study and use; newbie friendly; superior usability; and better user experience. Given the communities where the data were extracted from, the study was business-oriented, showing most of the use of LCD for the development of websites and mobile applications. Although, if we consider that visual programming is included, then it has been massively used in education [Hu et al. 2021].

Visual Programming Languages (VPL) has shown a higher performance in education environments in comparison to **Textual Programming Languages (TPL)** when it comes to motivation [Tsukamoto et al. 2016], basic programming concepts [Tsai 2019] and academic achievement [Hu et al. 2021]. It is highlighted that VPL specially outperform TPL on students with low self-efficacy [Tsukamoto et al. 2016] and on those in elementary and middle schools [Hu et al. 2021]. Among other VPL used in CS education, Scratch [Resnick et al. 2009], that offers colorful blocks of code to drag and drop, stand out, generally linked to CT [Oliveira et al. 2019, de Lima Sousa et al. 2020].

Among other definitions, CT can be defined as "the mental skills and practices for designing computations that get computers to do jobs for us, and explaining and interpreting the world as a complex of information processes." [Denning and Tedre 2019]. More specific definitions split according to their target, some more related to programming and computing concepts [Brennan and Resnick 2012, Weintrop et al. 2016], others to more general problem-solving skills [Selby and Woollard 2013, ISTE and CSTA 2011]. A model based on a SLR that gathered the most cited terms in CT definitions [Silva Junior 2017] brings six major lines of CT: **abstraction**, approaching simplification, generalization, modeling and pattern recognition; **decomposition**, approaching breaking down problems into smaller ones, emergence and reuse; **algorithm**, approaching flow of control and parallelism; **data**, approaching data structures, visualization and representation; **automation**, including development of software, tinkering and simulation; and **evaluation**, approaching efficiency, test and debug.

Many efforts have been made to assess CT, being the majority based on traditional paper-pencil tests and very few presenting reliability and validity evidence [Tang et al. 2020]. This area still needs further research, but looking at the few with reliability and validity evidence, we highlight: the **Computational Thinking test (CTt)** [Román-González 2015], an online test with closed answer questions about algorithms, sequences, conditionals, loops, functions and debugging; and the **Dr. Scratch** [Moreno-León et al. 2015], a tool that automatically analyzes Scratch projects and score them in CT competences, such as abstraction, parallelization, flow control and data representation. Complementary tools are also used to ease or deepen the as-

sessment. For instance, traditional paper-pencil tests can be improved with F-ATL, a method that uses fuzzy logic to consider the inherent uncertainty in assessment of teaching-learning [Cardozo et al. 2019]. While for automated tools, tracking student's progress in Scratch is possible with OntoScratch, an ontology to store in a structured way the trajectory of a project according to the Dr. Scratch's metrics of assessment [de Araujo et al. 2020].

2.2. Games and Maker Movement in Education

From all teaching and learning strategies, educational games stand out due to its potential to keep the learner in an optimal engagement [Garris et al. 2002], a state called Flow [Nakamura and Csikszentmihalyi 2009]. It is a zone located between boredom and anxiety, where involvement is maximized [Csikszentmihalyi and Csikszentmihalyi 1992]. This concept is widely used in game design because it is reached when the player is as challenged as it is able to answer it, in turn, balancing the task difficulty against the player's capability is a game design task by nature [Hiremath et al. 2015]. Educational games and game-related environments have considerably increased their number of positive results analyzed and reported within the scientific community, proving themselves as viable educational approaches [Boyle et al. 2016].

Just like the popularization of digital games attracted the interest of education research, the growth of **Do It Yourself (DIY)** hobbyists, tinkerers, engineers, hackers, and artists who creatively design and build projects gave attracted the interest of education research to the so-called Maker Movement. It proposes active learning, turning the student into protagonist of its own education through making, what has shown to: empower youth to engage in new forms of thinking; lead to powerful forms of learning driven by recursive feedback, where people learn from the actions of their creations; be playful and highly tolerant of errors; advocate a growth mindset; and link learning communities [Lee et al. 2011]. GrameStation aims to bring together a game-related environment, a maker context and CS education, as Scratch does, and is probably one of the reasons it is vastly used world-wide [Zhang and Nouri 2019].

2.3. Graph Grammar

A GG describes a system modeling a state of it as a graph (vertices and edges), and events that may alter the current state as a set of graph transformation **rules**. A GG must define how its **state graph** starts, in what is called the **initial graph**. Additionally, a GG may distinguish and restrict its elements by declaring them in a **type graph**. For instance, Figure 1 models the Pacman game as a GG. There is a type graph T declaring the existence of pacmans, ghosts, places (black dots), fruits, counters (white dot) and their relations in this GG. Then, there is an initial graph Ini that shows a pacman, two ghosts and a fruit in a 3x3 grid of places, while a counter at the bottom shows that a fruit has been eaten. Then, there are four rules, $moveP$, $moveG$, eat and $kill$, each being represented by a pair of graphs linked by an arrow carrying the name of the rule.

The pair of graphs representing rules are the **Left-Hand Side (LHS)**, expressing a condition for applying the rule, and the **Right-Hand Side (RHS)**, expressing a consequence of applying the rule. As in $moveP$ (Figure 1), the LHS defines the condition of

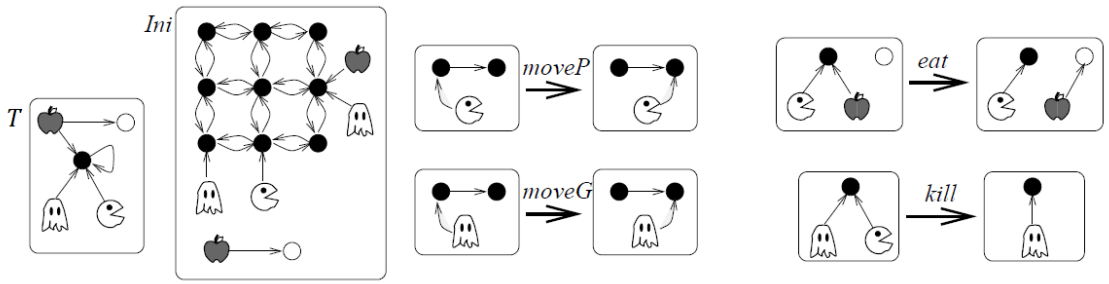

Figure 1. A Pacman graph grammar. Source: Adapted from [Ribeiro 2000]

.

having a pacman in a place that has a way to another, while the RHS defines the consequence of removing the pacman from the initial place and putting it into the other. This representation also implies element-wise mappings between graphs (**morphisms**). That is, for each element in one graph, we have to say which element (if one) in the other graph corresponds to it. For instance, saying if the pacman in the LHS is the same pacman in the RHS. If an element is successfully mapped (has a correspondent), it means the rule **preserves** it, as the pacman of $moveP$. If an element is left unmapped (has no correspondent) and is in the LHS, then the rule **deletes** it, as the pacman of $kill$. If unmapped and in the RHS, the rule **creates** it, as the edge from the fruit to the counter of eat.

This pacman is played by applying rules, which is a process that depends on another morphism: the **match**. The match is a total morphism from the LHS to the state graph, meaning that we must find a correspondence for each element. In Ini we could apply $moveP$ mapping any of the three adjacent places and $moveG$ mapping any of the ghosts and their adjacent places. But not eat and $kill$, because there is not a pacman and a fruit/ghost in the same place, so we could not complete a match, since they must respect source and target of the edges.

The base theory for GG makes no distinction between vertices and edges, i.e. it is not sufficient to specify that a vertex is a pacman and another is a ghost, they are all simply vertices. Distinguishing between elements is possible through labeling or typing, which is mapping every element into a label or a type. For typing, an additional component enters the GG: the **type graph**, which is a special graph where each element is considered distinct from the other. Then, every other graph of the GG must map their elements into the type graph (typing morphism) and respect the source/target restrictions their types imposes. For instance, an edge from a black dot can point to another black dot in Ini (Figure 1), but not to a white dot, because in T there is an edge with the black dot type as its source and target (a loop), but no edge with the black dot as source and white dot as target. Again, in the visual representations, the morphisms are implied by the look of the elements, corresponding looks in different graphs imply they are mapped.

We avoided exposing the underlying math in this paper, but the formal definitions that we are following comes from an algebraic approach for GG [Corradini et al. 1997], using Double Pushout for graph transformations, injective matches and the category of typed attributed graphs and total morphisms [Cavalheiro et al. 2017]. These are common arrangements for GG, that can be specified in GG automatic tools present in the literature [Azzi et al. 2018, Taentzer 2003, Rensink 2003]. Our game engine is indeed a

software that allows the user to specify and run GG as those, but it does not offer tools for formal specification and analysis as they do. The core distinction is that GrameStation adapt GG to turn them into games, what we made introducing the concept of **Grame**, short for graph game.

A grame is essentially a GG that contains special elements (**gears**) hidden from players (but not from creators) to control gaming aspects, such as which player is able to play at a given time (managing turns) and which rules each are available for each player. During the execution of grames, the engine offers players a subset of rules (controlled by gears), so the player can select one of them to start mapping a match to apply the rule. Gears can also be created or deleted by rules, which means this control is in the grame creator's hands. Thus, GrameStation approaches GG centring decision making through a dynamic environment, where the users respond on the fly to the state transitions caused by others. While the existing GG tools approaches GG centring formal specification through a single user environment, where users analyze properties that emerge mostly from the automatic execution of its rules and the exploration of the generated state space.

3. GrameStation

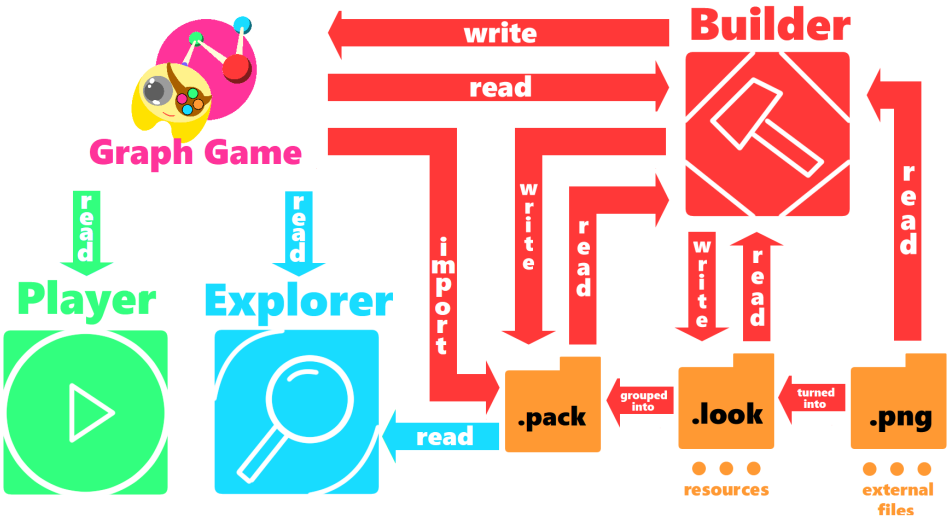

Figure 2. GrameStation modules and overall organization.

.

We have made educational games based on GG, from physical boardgames [Silva Junior et al. 2017] to digital games [Silva Junior et al. 2019a]. These previous experiences in turning GG into games matured into the notion of grames. In turn, a GG framework made in Unity [Technologies 2020] we have developed for running the digital game matured into the GrameStation. The engine is used for both, creating and running games, but it is separated into modules for the respective purposes as seen in Figure 2: the Grame Builder and Grame Player. Additionally, a Grame Explorer module locates and opens the grames, stored in an **Extensible Markup Language (XML)** file defined by a **XML Schema Definition (XSD)** file we specified.

In adition to the GG, grames count with external files such as images and audio effects to be used as **resources**, e.g. for the appearance of the vertices and edge. GrameS-

tation organizes resources in collections (**packs**), and currently supports images using **Looks**, which are XMLs encapsulating a png or a jpg file. If we want to make a grame, the very first thing we shall do is to import pre-available packs or to create a new one. For instance, Figure 3 shows a pack of looks for making a pacman grame.

Figure 3. Looks for the pacman grame.

.

The next step in designing a grame is to define the types that we are going to use. Thus, the type graph fits the role of a declaration area. All grames begin with an empty type graph and an empty initial graph. GrameStation allows the users to create new types by requesting them a kind (vertex or edge), a name, a look and a color (source and target are also requested if its an edge). As shown in Figure 4, for a pacman grame we filled the type graph declaring: a Pacman, a Ghost, a Fruit and a Place. Additionally, we added two vertices to represent Victory and Loss. As a grame is a GG, relations stand out, so we also declared that: Pacman isAt a Place; Ghost isHaunting a Place; Place may have a wayTo another Place; and a Place hasA a Fruit.

Figure 4. Type graph for the pacman grame.

.

With everything declared, we proceed to fill the initial graph. GrameStation allows the users to create new elements in the initial graph by instantiating the ones defined in the type graph, requesting only a type and a name (source and target are also requested if its an edge). Continuing our example, Figure 5 shows the initial graph of the pacman grame. There are our hero pacman, our enemy ghost and our goal fruit at (isAt, isHaunting, hasA, respectively) different linked (P01, P02, P12, P23) places (P0, P1, P2, P3).

At last, we specify the rules. GrameStation allows the creator to add as many rules as they want. And to add an element to a rule, it will request if it is deleted, preserved

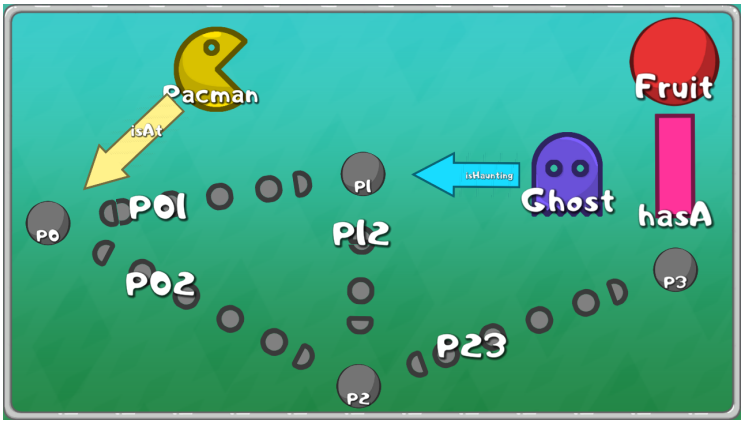

Figure 5. Initial graph for the pacman grame.

.

or deleted by the rule, as well as type and name. In our example (shown in Figure 6), the same $moveP$ from the pacman GG we introduced in section 2 (Figure 1) is modeled in the GrameStation. This rule preserves a pacman, two places (CurPos,FutPos) and the way between them (wayTo), while replacing its positioning edge that target a place (delete wasAt) to one that targets the other (create isGoingTo), effectively moving the pacman. All the others rules can be defined just like this one.

Figure 6. Rule $moveP$ for the pacman grame.

.

The specified rules can be selected on the top of the screen in the Grame Player during the execution. If so, their LHS and RHS will be showed and a match can be set by clicking the LHS elements and their corresponding elements in the board (state graph) in sequence. If any illegal mapping occurs, such as mapping a pacman into a ghost, a cross will appear signalizing the error and the current match will be cancelled. Grames are designed to support parallel, asynchronous rule applications, result of multiple players playing at the same time. But, as GrameStation does not offer support for online play yet, an adaptation to allow multiplayer grames is temporarily held as default: once a rule is applied by a player in a computer, this computer will become the next player (they switch turns). This is all controlled by gears and could be changed by expert users. Gears bring a whole new level of freedom to model games, being possible to make real-time grames, rules that pass the turn backwards or changes the player order and much more. But for beginners, they should be completely ignored, as we ignored in our pacman grame. By

default, all is set up to be a simple turn-based that passes the turn when any rule is applied.

Internally, Grame Builder works through a parser reading and executing command lines, but a grame designer is not supposed to code. A decision tree system, named the **definer**, is called every time someone wants to build or edit anything in the module. It asks the user for consecutive choices that build up a complete definition (a command line), as result, this constructs a graph linking the new choices with the previous. An example of the definer is shown in Figure 7: a sequence of expansions in a decision tree to build the type for pacman. From left to right, top to bottom, first we select that we are building a vertex, then we enter its name, choose the package of the look (there is only the basic set available), choose the look, choose the package of the color, then the color.

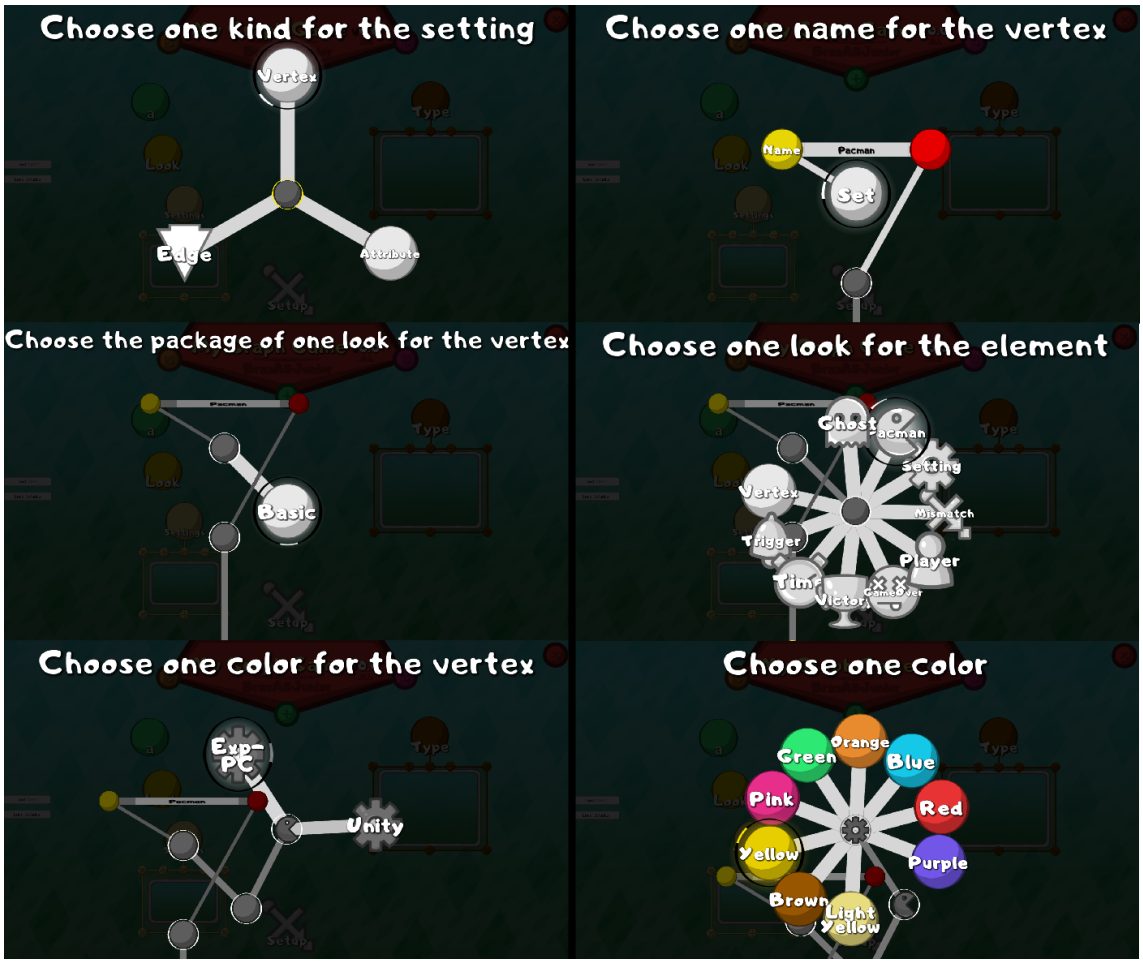

Figure 7. GrameStation's definer sequence for creating a pacman type.

.

4. Discussions and Conclusions

There are a number differences between specifying and programming. Specification is much more problem-focused, much closer to the theoretical foundations of CS. While programming is much closer the the applications of CS. In a complete workflow, specification is the stage that confronts the problem, it is where the problem is conceived, modeled and abstractly solved. Programming comes after, it receives the problem already

modeled from the specification and then turns it into a concrete solution for the real word. The specification stage has being neglected in CS education, students try to understand and model the problem on the fly while already programming. As a result, the CT movement rises and shouts for problem-solving skills to fill this gap. The GrameStation is an effort to revive the specification stage and automatically bring specifications to the real world (final product execution).

More than that, some of the skills often claimed for by the CT movement are related to GG aspects: they raise the **level of abstraction** by being able to get completely rid of texts; and by being declarative, where you just specify key graphs (a condition and a consequence), rather than sequencing orders to describe how to achieve them; the idea of matching is one of the meanings of **abstraction** – selecting only the necessary elements for an event of interest and temporarily ignoring the rest; matching is also pretty straight forward a **pattern recognition** process; rules are implications, "**conditionals**" – if LHS, then RHS; types are **generalizations** of elements, while rules are generalizations of behaviors of subgraphs; it is naturally a **parallel** model; it is based on well-known **data structures** – graphs; and the match being required for rule application implies that GG execution is compulsorily an **evaluation** task, you cannot disassociate an operation from examining its context.

Therefore, we conclude that CS education should be approached further than just coding, relying on programming as a wider concept that systematizes problem-solving while consulting the information processes and phenomena studied by CS. And since problem-solving becomes essential in this approach, CS education shall foster CT skills to support it. We should also address to the many challenges newcomers face, such as syntax difficulties and lack of motivation. Accordingly, we propose GrameStation[1], a platform to use graphs for creating games, what classifies it as a latest generation programming language, LCD, VPL, game-related environment and aligns with the maker movement practices. At last, we cannot neglect CT assessment, this area still needs further research and GrameStation might be very helpful by automatizing data gathering and allowing CT assessment through the association with GG concepts.

Future work involving GrameStation includes adding: pedagogical agents to guide newcomers; attributes (from attributed graph grammars theory); negative application conditions (NAC); and online/network support. As well as conducting several empirical studies conducting various activities to test it as an instrument for: developing CT; teaching GG; teaching specification; and teaching parallel processing.

References

Armoni, M. (2016). Computer science, computational thinking, programming, coding: the anomalies of transitivity in k-12 computer science education. *ACM Inroads*, 7(4):24–27.

Azzi, G. G., Bezerra, J. S., Ribeiro, L., Costa, A., Rodrigues, L. M., and Machado, R. (2018). The verigraph system for graph transformation. In *Graph Transformation, Specifications, and Nets*, pages 160–178. Springer, Cham.

[1] `https://wp.ufpel.edu.br/pensamentocomputacional/gramestation-pt/`

Boyle, E. A., Hainey, T., Connolly, T. M., Gray, G., Earp, J., Ott, M., Lim, T., Ninaus, M., Ribeiro, C., and Pereira, J. (2016). An update to the systematic literature review of empirical evidence of the impacts and outcomes of computer games and serious games. *Computers & Education*, 94:178–192.

Brennan, K. and Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, volume 1, page 25.

Cardozo, A., Gayer, C., Cavalheiro, S., Foss, L., Du Bois, A., and Reiser, R. (2019). Flexible assessment in digital teaching-learning processes: Case studies via computational thinking. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 429.

Cavalheiro, S. A. d. C., Foss, L., and Ribeiro, L. (2017). Theorem proving graph grammars with attributes and negative application conditions. *Theoretical Computer Science*, 686:25–77.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Löwe, M. (1997). Algebraic approaches to graph transformation–part i: Basic concepts and double pushout approach. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 163–245. World Scientific, Singapore.

Csikszentmihalyi, M. and Csikszentmihalyi, I. S. (1992). *Optimal experience: Psychological studies of flow in consciousness*. Cambridge university press, Cambridge.

de Araujo, N., Primo, T. T., and Pernas, A. M. (2020). Ontoscratch: ontologias para a avaliação do ensino de pensamento computacional através do scratch. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1823–1832. SBC.

de Lima Sousa, L., Farias, E. J., and de Carvalho, W. V. (2020). Programação em blocos aplicada no ensino do pensamento computacional: Um mapeamento sistemático. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1513–1522. SBC.

Denning, P. J. and Tedre, M. (2019). *Computational Thinking*. Mit Press.

Denning, P. J. and Tedre, M. (2021). Computational thinking: A disciplinary perspective. *Informatics in Education*.

Gaggioli, A. (2017). The no-code revolution may unlock citizens' creative potential. *Cyberpsychology, Behavior, and Social Networking*, 20(8):508–509.

Garris, R., Ahlers, R., and Driskell, J. E. (2002). Games, motivation, and learning: A research and practice model. *Simulation & gaming*, 33(4):441–467.

Hiremath, S. V., Chen, W., Wang, W., Foldes, S., Yang, Y., Tyler-Kabara, E. C., Collinger, J. L., and Boninger, M. L. (2015). Brain computer interface learning for systems based on electrocorticography and intracortical microelectrode arrays. *Frontiers in integrative neuroscience*, 9:40.

Hu, Y., Chen, C.-H., and Su, C.-Y. (2021). Exploring the effectiveness and moderators of block-based visual programming on student learning: A meta-analysis. *Journal of Educational Computing Research*, 58(8):1467–1493.

ISTE and CSTA (2011). *Computational Thinking leadership toolkit*. 1 edition.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L. (2011). Computational thinking for youth in practice. *Acm Inroads*, 2(1):32–37.

Lindberg, R. S., Laine, T. H., and Haaranen, L. (2019). Gamifying programming education in k-12: A review of programming curricula in seven countries and programming games. *British Journal of Educational Technology*, 50(4):1979–1995.

Luo, Y., Liang, P., Wang, C., Shahin, M., and Zhan, J. (2021). Characteristics and challenges of low-code development: The practitioners' perspective. *arXiv preprint arXiv:2107.07482*.

Martin, L. (2015). The promise of the maker movement for education. *Journal of Pre-College Engineering Education Research (J-PEER)*, 5(1):4.

Medeiros, R. P., Ramalho, G. L., and Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90.

Moreno-León, J., Robles, G., and Román-González, M. (2015). Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *Revista de Educación a Distancia (RED)*, 15(46):1–26.

Nakamura, J. and Csikszentmihalyi, M. (2009). Flow theory and research. *Handbook of positive psychology*, pages 195–206.

Oliveira, G., Assunção, O., and Prates, R. (2019). Strategies to introduce computational thinking to children: An analysis based on cultural viewpoint metaphors. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 547.

Rensink, A. (2003). The groove simulator: A tool for state space generation. In *Proceedings of International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485, New York, NY, USA. Springer.

Resnick, M. et al. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67.

Ribeiro, L. (2000). Métodos formais de especificação: gramáticas de grafos. *VIII Escola de Informática da SBC-Sul*, pages 1–33.

Román-González, M. (2015). Computational thinking test: design guidelines and content validation. In *Proceedings of 7th Annual International Conference on Education and New Learning Technologies (EDULEARN 2015)*, pages 2436–2444.

Selby, C. and Woollard, J. (2013). Computational thinking the developing definition. *University of Southampton (E-prints)*.

Silva Junior, B. A. (2017). A última árvore, utilizando gramática de grafos em um jogo educacional para explorar o pensamento computacional. 2017. Thesis (Bachelor's) – Bachelor of Science in Computer Engineering, Center for Technological Development, Federal University of Pelotas, Rio Grande do Sul, 2017.

Silva Junior, B. A., Cavalheiro, S. A. C., and Foss, L. (2018). Uma análise de um jogo educacional sob a ótica do pensamento computacional. In *Simpósio Brasileiro de Informática na Educação-SBIE*, volume 29, pages 595–604.

Silva Junior, B. A., Cavalheiro, S. A. C., and Foss, L. (2019a). Revisitando um jogo educacional para desenvolver o pensamento computacional com gramática de grafos. In *Simpósio Brasileiro de Informática na Educação-SBIE*, volume 30, pages 863–872.

Silva Junior, B. A., Cavalheiro, S. A. d. C., and Foss, L. (2017). A última árvore: exercitando o pensamento computacional por meio de um jogo educacional baseado em gramática de grafos. In *Simpósio Brasileiro de Informática na Educação-SBIE*, volume 28, pages 735–744. Porto Alegre: SBC.

Silva Junior, B. A., Cavalheiro, S. A. d. C., and Foss, L. (2019b). Métodos formais na educação básica: Operando gramática de grafos em um jogo educacional. In *Workshop-Escola de Informática Teórica-WEIT)*, volume 5, pages 178–186. Passo Fundo: UPF.

Taentzer, G. (2003). Agg: A graph transformation environment for modeling and validation of software. In *Proceedings of International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 446–453, New York, NY, USA. Springer.

Tang, X., Yin, Y., Lin, Q., Hadad, R., and Zhai, X. (2020). Assessing computational thinking: A systematic review of empirical studies. *Computers & Education*, 148:103798.

Technologies, U. (2020). Unity real-time development platform — 3d, 2d, vr and ar engine. Unity Technologies, 2020. Available at: `https://unity.com/`. Accessed: 2020-02-17.

Tsai, C.-Y. (2019). Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior*, 95:224–232.

Tsukamoto, H. et al. (2016). Textual vs. visual programming languages in programming education for primary schoolchildren. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–7. IEEE.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., and Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25(1):127–147.

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

Zhang, L. and Nouri, J. (2019). A systematic review of learning by computational thinking through scratch in k-9. *Computers & Education*, page 103607.