CBIE 2023
CONGRESSO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO
Uma escola para o futuro: Tecnologia e conectividade a serviço da educação

Featuring Layers of Abstraction as a Modeling Resource in an Educational Game-Based Learning Platform [*]

Braz Araujo da Silva Junior[1], **Júlia Veiga da Silva,**
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,jvsilva,simone.costa,lfoss}@inf.ufpel.edu.br

Abstract. *This paper presents a feature supporting the modeling, management and visualization of layers of abstraction in an educational game engine. Computer science education has conquered significant recognition supported by the concept of Computational Thinking, an essential skill set in a world where computing is pervasive. One of its pillars, abstraction, has many facets on its own, and some of them still struggle to find support of educational tools. Attempting to fill this gap, this work proposes a resource that allows grouping elements while modeling games. Abstract hierarchical graph grammars are used as theoretical foundation and practical examples of how it fosters abstraction are given. The processes found to be possible to grasp with it are generalization, refining, modularization, nesting and navigation through layers of abstraction.*

1. Introduction

In recent years, **Computer Science Education (CSE)** has witnessed a paradigm shift that has placed **Computational Thinking (CT)** at the center of modern pedagogical practices. As technology continues to permeate every aspect of our lives, the ability to approach and solve problems through the lens of computing has become an indispensable skill for everyone, not just computer scientists and Information Technology (IT) professionals [Wing 2006]. By offering a structured and systematic approach to problem-solving based on computing fundamentals, CT empowers learners to analyze complex challenges, devise efficient solutions, and make informed decisions that have real-world implications [Avila and da Costa Cavalheiro 2022, Vieira and Zaina 2021].

Despite CT having many definitions slightly varying which abilities it encompasses [Tang et al. 2020], it generally refers to a cognitive process of problem solving using a series of skills and concepts of **Computer Science (CS)** [Tikva and Tambouris 2021]. It provides a foundational framework for understanding the fundamental principles that underlie CS and harnesses the power of abstraction as a key mental tool. Abstraction, one of the core pillars of CT [Wing 2008], allows learners to create simplified representations of complex systems, enabling them to focus on essential details while suppressing unnecessary intricacies. This cognitive process facilitates the construction of models that capture the essence of a problem, facilitating effective problem-solving across a broad range of domains [Mirolo et al. 2022]. As such,

[*]This work was carried out with the support of CAPES - Brazil - Financing Code 001, the SACCI Network, SMED/Pelotas, PREC and PRPPG / UFPel.

abstraction serves as a bridge between the real world and the digital realm, empowering students to reason about problems and devise solutions in ways that leverage the strengths of computational systems.

The importance of abstraction in CT cannot be overstated, as mentioned in a **Systematic Literature Review (SLR)** on abstraction, "it has been the most significant component of CT in empirical studies measuring learners' CT development" [Ezeamuzie et al. 2022]. However, it is safe to assume that developing this skill requires deliberate and strategic efforts in educational settings. As students grapple with increasingly intricate problems, they must be equipped with proper tools and resources to cultivate their ability to construct and navigate multiple layers of abstraction.

In this paper, we shed light on the critical skill of abstraction in CT and present an effort to respond the pressing need for specialized tools and resources to foster its development [Mirolo et al. 2022]. We present a feature that enables and emphasizes modeling layers of abstraction in an educational game-based learning platform targeting **Kindergarten to 12th grade (K-12)** education, but suitable for all ages. Throughout the paper we explore the following research questions:

1. What features or structures could support the modeling and management of layers of abstraction in GrameStation?
2. How could these features foster CT? That is, how could these features induce or influence learners to solve problems using strategies and abilities aligned with CT?

The rest of this paper is organized as follows. Section 2 discusses related work, giving an overview of how layers of abstraction have been approached in the literature. Section 3 describes the strategy, approach, theoretical foundation and tools we used. Section 4 presents the feature and examples using it. Section 5 concludes the paper with suggestions for future research.

2. Related Work

A recent overview [Mirolo et al. 2022] and a SLR [Ezeamuzie et al. 2022] on abstraction in CSE showed that the term "abstraction" has multiple characterizations and is tied to various topics in CS, as: problem formulation; extracting similarities; ignoring non-essential features; decomposition; computational abstractions; programming languages and paradigms; generalisation and parametrization; procedural and data abstraction; information hiding; and abstraction layers.

The overview also mentions that abstraction is rarely explicitly approached, as expected from a very broad term. Instead, it is expected to emerge from various tasks, mainly revolving around modeling and programming, which is how it has been mainly approached [Kakavas and Ugolini 2019, Mirolo et al. 2022]. Frameworks explicitly targeting abstraction have been presented in the literature [Armoni 2013, Qian and Choi 2022]. But they are theoretical guidances for educators to create tools/conduct activities in a way abstraction is favoured, rather than practical tools to be used by the learners.

On top of that, here we are focusing a specific aspect of abstraction: **layers of abstraction**, which refers to the hierarchical organization of complex systems or problems into multiple levels of representation, with each level hiding unnecessary details and exposing only the essential elements needed for a particular purpose. This approach helps manage the inherent complexity of systems and enables problem solvers to reason at various levels of granularity.

To the best of our knowledge, tools directed towards working with layers of abstraction in K-12 education are scarce. Arguably, visual programming languages and other CSE modeling/programming environments are able to handle some form of layered abstraction, but as a minor or implicit feature that does not receive much attention. Beyond programming, activities introducing notions of layers of abstraction approach it by showing multiple forms of representation of the same problem. For instance, in a chemistry class on the natural carbon cycle, students are shown a real-world macroscopic phenomenon; the scientific notation of the chemical reactions happening there; a computer-generated simulation of the phenomenon; and the simulation's source code. [Gautam et al. 2020].

3. Methods

Given the lack of tools offering a explicit and central way to work with layers of abstraction, our goal was to properly fill this gap: something very visual, where the abstraction and its layers would be an obvious focus. At the same time, we wanted to avoid isolating the activity from the rest of CT, in an effort to show how this specific skill can contribute to general problem solving in practice (not just in a "controlled environment" designed just to teach that). This lead us to the decision of integrating an abstraction-focused feature into an existing platform that already explores CT. GrameStation, an educational game modeling/specification engine based on **Graph Grammars (GG)** fits well this role, since: its development is already in the context of education (primarily designed for K-12 education, suitable for all ages); game-based learning provides engagement and room for complex projects where abstraction could really shine; graphs – used everywhere in the platform – are a visual data structure; and the platform was developed to foster CT.

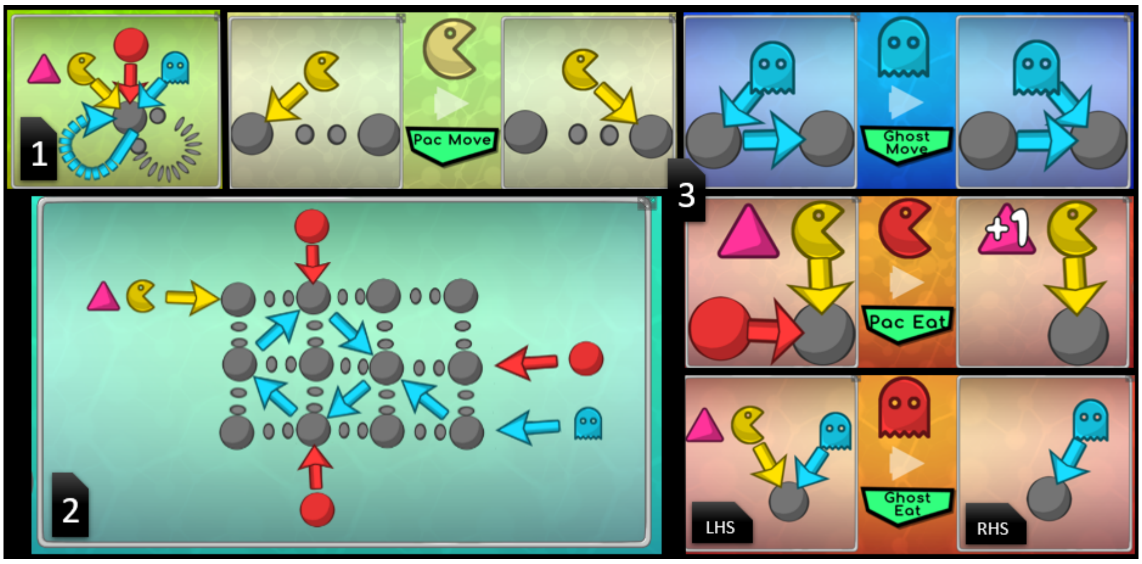

Figure 1. Graph Grammar of a Pac-Man game.

The whole engine is based on GG, a formal language of specification and verification of systems, which we will briefly introduce the core notions[1]. Basically a Graph Transformation System (GTS) equipped with a initial state and a typing mechanism, a GG describes a system as (see Figure 1): 1- a type graph that represents its universe (the types of things that can be found within it); 2- an initial graph that represents its first state; and 3- a set of graph rewriting rules that represents how states can change. Those rules map graphs element-wise to define which elements will be preserved, created or deleted by the rule application. They are represented by two graphs: Left-Hand Side (LHS), indicating a condition; and Right-Hand Side (RHS), indicating a consequence.

In order to bring layers of abstraction to this GG environment, we developed the concept of "wrappers", that behaves similarly to a subgraph. The underlying theoretical framework used comes from the abstract **Hierarchical Graph Grammars (HGG)** [Busatto et al. 2005], which provides an hierarchical organization on top of a base graph by using two additional graphs. One graph defines the relationship between packages, which are elements that will organize the elements of the base graph. And the other graph indicates in which package each base element is contained. For instance, Figure 2 shows a hierarchical graph: the base graph on the left defines a snake, a bear and a bird in a grassy land adjacent to a beach and a mountain; the hierarchy graph on the middle defines a package of animals and another of lands; and the coupling graph on the right defines the containment relations between base elements and hierarchy packages.

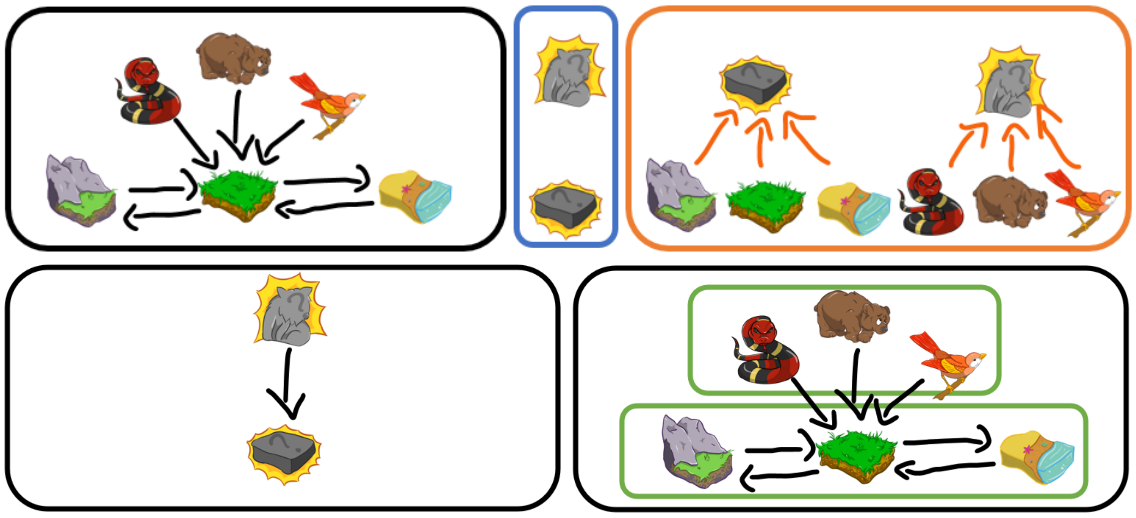

Figure 2. Example of hierarchical graph as three graphs (top) and their simplified representation (bottom) as one, either collapsed (left) or expanded (right).

Throughout the paper we will discuss some desired topics on object orientation. While we acknowledge the existence of **Object-Oriented Graph Grammars** [Ferreira 2005, Ferreira et al. 2007], we didn't want to bound all activities to object orientation nor rework the whole engine, adapting it to a different theory. The HGG

[1]All GG notions presented in this paper refer to the specific approach used as theoretical foundation to the engine and the proposed feature. That is, some statements may not hold true for other approaches to GG. For interested readers, we refer to [Silva Junior et al. 2021] to learn more about the platform and to [Ehrig et al. 1991] for the mentioned GG approach.

approach was chosen because it allowed us to add the grouping information without compromising all definitions from the theoretical framework GrameStation already relied on. However, it felt overwhelming to always show the user two additional graphs just for that. Therefore, we hid them, embedding the information provided by the HGG into the visualization of the base graph. We implemented the "packages" from the theory as "wrappers" (see Figure 2), that are shown as regular vertices while collapsed (left) and regions behind their content while expanded (right).

4. Wrappers

In this section we will discuss how exactly wrappers may be used to foster abstraction skills, modeling using different layers of abstraction and how they facilitate the visualization of all that. As an explanatory support, we will use an educational game to offer practical examples. The game was chosen due to the convenience of being designed as a GG, we refer to [Silva Junior et al. 2017] for readers wanting further details, here we will summarize the essentials. It is a turn-based strategy game were animals are trying to revitalize a recently deforested jungle. The players may apply simple rules representing the growth cycle of plants and their dispersion being carried by the animals. There are multiple goal states randomly drawn and assigned to each player at the start of each game. The player that manages to make the jungle look like their goal first wins the game.

4.1. Generalizing

The game features four animals: a bird, a snake, a bear and a wolf. They do not have different behaviors in the game, which means they are functionally the same kind of thing. That is why they are modeled as a single type, animal. The different visual representation and names of the instances are just conventions to make the game more visually diverse, they are not a real distinction for the GG.

If they were modeled as different things, each of their own type, we would quickly realize some inconvenience trying to model common behavior between them. For instance, the rule that moves an animal from a square to another would have to be designed four times, each using one of the animals. That is the reason why modeling different types for things that do not behave differently is considered bad design, but that is a common mistake our intuition leads us to do while modeling. Additionally, this inconvenience is inescapable if we are after things that are supposed to behave differently but also to have some common behavior.

To deal with that problem, we shall generalize the elements. Wrappers can be used to do that in a very intuitive way (see Figure 3): an Animal would be a wrapper containing either a Bird, a Bear, a Snake or a Wolf (1). Then, instead of four rules (2), we would need to have only one, using the Animal wrapper (3). The important thing to note here is that we are still able to make specific rules for each type of animal, which wouldn't be achievable if they were all the same type.

For instance, let's say the bird should be able to Fly (see Figure 4-1), which would be a special move allowing it to go from a square to another regardless of the connections between them (the original movement rule requires connected squares). If the animals were all of the same type, we would have no way to refer specifically to the Bird, then either all would be able to fly, or none would. With the wrappers, we can let only the Bird fly, and all animals (including the Bird) move.

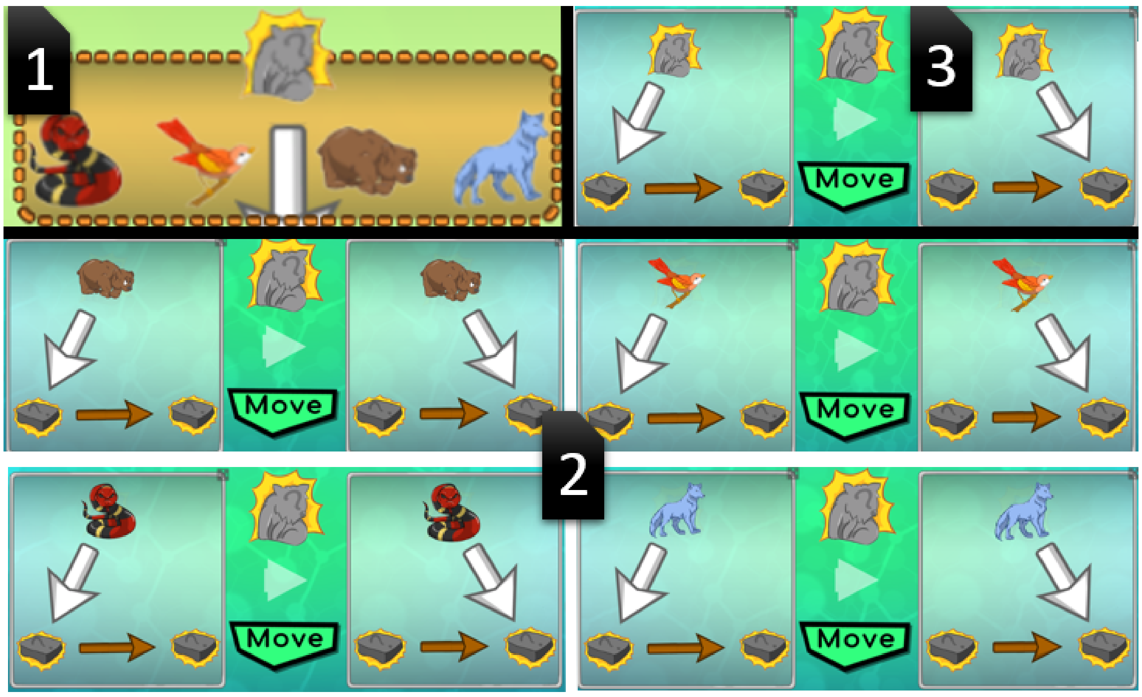

Figure 3. Generalization from Bird/Bear/Snake/Wolf to Animal.

So, here we properly differentiated and generalized the original four animals of the game, allowing them to show both, general behavior to share amongst all of them, and specific behavior to further characterize each.

4.2. Refining

The other way round, breaking an abstract concept into more concrete ones, is also possible and useful. We will harness the opportunity to illustrate how relations (edges) can also be approached in different layers of abstraction.

We created a specialized behavior for the Bird, but not a corresponding specialized relation. A Bear may not be able to reach all squares a Bird can, but if they are at the same square, they stand the same kind of relation to that square (IsAt). Differentiating them could further enrich (and increase complexity of) the game. For instance, the original rule to Gather fruits from trees could be made more strict, allowing only animals that are above the ground to reach and gather the fruits.

This effect is reached in Figure 4 (see 2 and 3), turning the IsAt (white) edge into a wrapper containing edges as OnTheGround (green) and AboveTheGround (blue). Then Fly and Gather could use the specific edge AboveTheGround relation; while Move use the generic wrapper IsAt (see Figure 3-3). This way Move could still be applied on both, those OnTheGround and those AboveTheGround, as nothing had changed, since it would require just the wrapper to match.

So, here we properly broke down a generic relation into two specifics, allowing it to have some of its behaviors specialized, while preserving the rest as no changes were made. This is one of the most useful features of working with layers of abstractions, we can make changes in one without compromising the others.

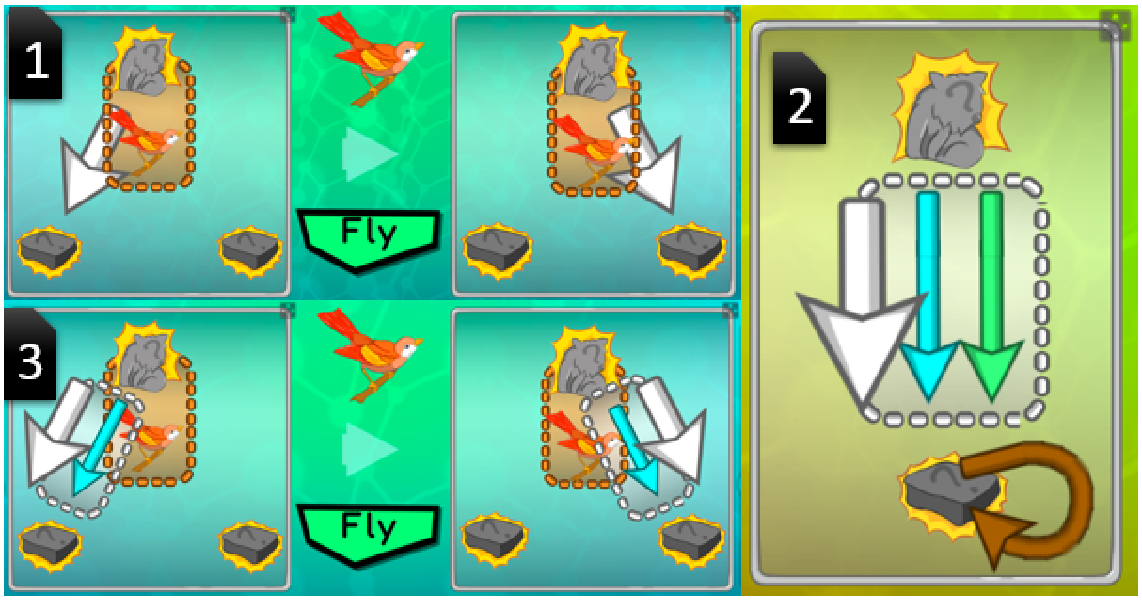

Figure 4. Refining of the edge IsAt in the Fly rule.

4.3. Object Orientation

Those generalizations and refinements may superficially simulate some characteristics of **Object Orientation Programming (OOP)**. There isn't a real inheritance or polymorphism, since an element must always be mapped to another of the exact same type (not to any descendant type/child class as in OOP). However, if we see a wrapper and its content as a form of "composite" type, they are able to produce similar behavior in GGs. That takes advantage of the fact that matches require some elements, but have no saying on the context of them[2]. That is, a general rule can require a wrapper regardless of its content.

Simulating that in GGs require some attention to avoid faulty behaviors like deleting the content of the wrapper, but not the wrapper, or the other way round. This would be analogous to stripping a child class off its superclass properties or somehow instantiating an abstract superclass.

4.4. Modularizing

Another use for wrappers is as containers, which can be used to modularize the project, either just for organization and better visualization or being functional elements themselves. For instance, we could organize the squares together with their seeds, plants and trees into places (see Figure 6-1 and 2). That would allow not just a better visualization, especially in big projects, but using the place as an element itself, regardless of its content.

For instance, we could add the sun to the game, illuminating one place at time, and adding this as a requirement for the rule to make plants Grow into trees. The difference of using places instead of squares for this is that we could allow some places to have different content, or even change their content mid game, while keeping the sun mechanic intact.

[2]That may vary for each Graph Grammar approach. In the one we have been using, the algebraic approach using Double-Pushout for rule applications, it is not entirely truth. There is a context-based restriction of not being able to delete elements that would leave dangling edges (without a source or target).

Wrappers can also be **nested** as long as they don't form a cycle. That is, there is no limit of how many layers of abstraction you could create with them. Nesting is generally only justified if the system gets too big or complex, which is not the case of our little game, here we forced nesting just to exemplify.

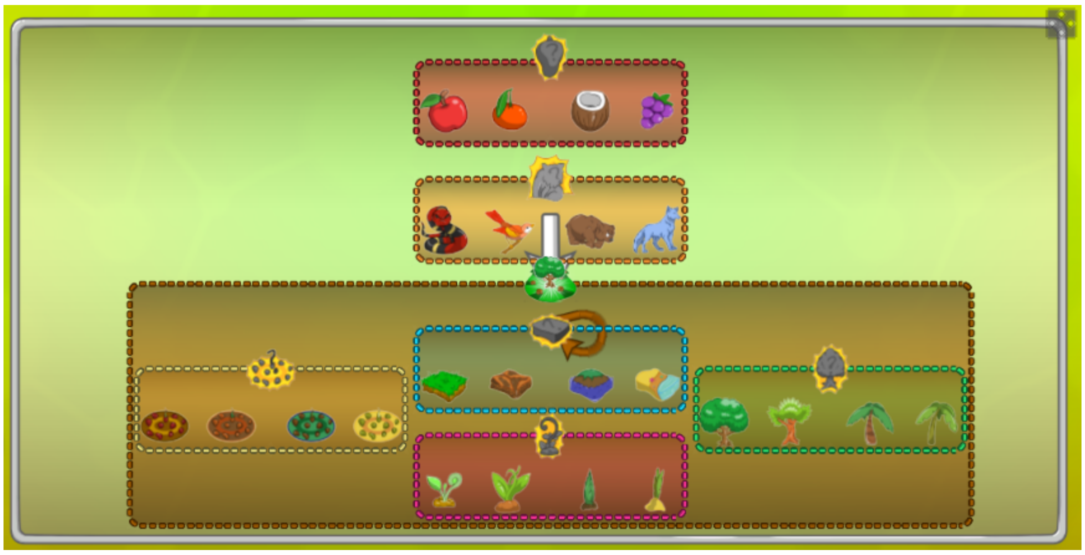

Figure 5. Type graph using wrappers for each element.

As Figure 5 depicts, if we considered each different representation of the original game as a type and then generalized them (as we did for the animals), we would have a wrapper for each element. On top of that, we could consider the "Place" suggestion we mentioned, which would be a wrapper containing multiple wrappers (see also Figure 6-1).

4.5. Navigating

The greatest advantage of wrappers is the clear visualization they offer to the modeler over the designed abstractions and its levels. In the platform, the default is a **free view**, where you can collapse or expand any wrapper at will, regardless of their layer. But we decided to empower and highlight the use of layers with the **layered view**, allowing the user to navigate through them seeing only the respective representations on each level (expanded wrappers are hidden, showing their content only).

That can be seen in Figure 6), where: (1) shows the free view with all wrappers expanded; and the rest show layered views, (2) with the highest layer of abstraction; (3) with the middle layer; (4) with the lowest; and (5) shows how the rule Fly (Figure 4-3) is much cleaner under the layered view. Noteworthy, this Fly rule seen through the highest layer looks exactly like Move (Figure 3-3), since one is a specialization of the other.

To deliver this control over the layers in a fun and engaging way, we used the "Abstractometer", that shows how many layers of abstraction are present in your project and can be used to navigate through them by clicking in the respective section. Figure 7 shows the usual interface of the game builder module of GrameStation, used to design the games as GG, featuring: (1) a component area to switch between rules, type and initial graphs; (2) a main display that shows relevant info as name, icon and ID of the object currently selected, and action buttons to add, edit or erase elements; (3) the work area, where the graphs/rules are displayed; and the new addition, the abstractometer.

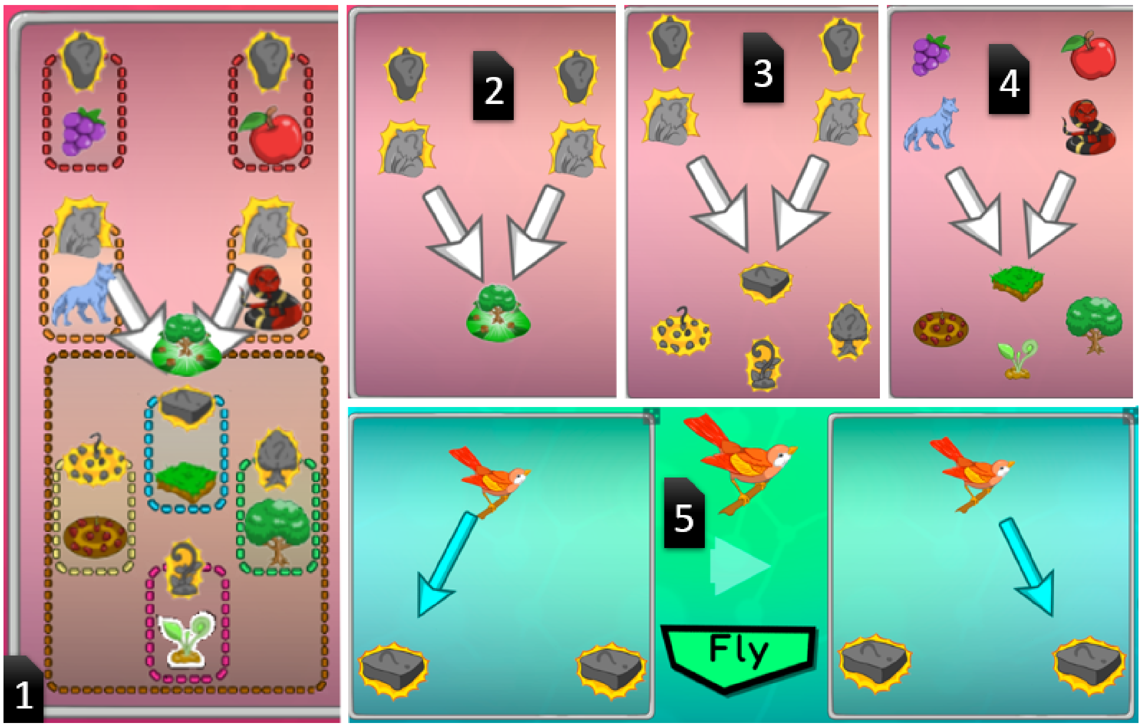

Figure 6. Free and layered views of a graph and rule.

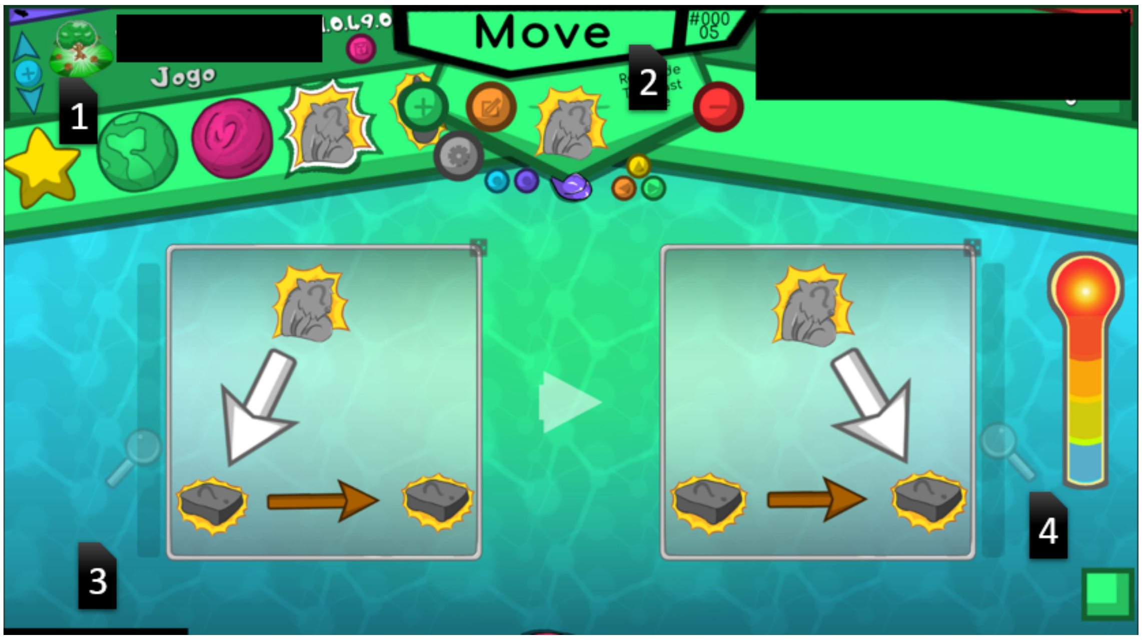

Figure 7. Interface of GrameStation with the Abstractometer.

5. Conclusion

In this paper we designed wrappers and proposed various ways in which their features could support the modeling and management of layers of abstraction in GrameStation. Given the current gaps [Mirolo et al. 2022], making available the tools to work with layers of abstraction in a more explicit manner is a valuable contribution to the CT community. And as GrameStation is a general purpose platform that allows the creation of any game that can be modeled as a GG, researchers and educators can build activities on various application domains counting with the layers of abstraction support.

Theoretically, everything achievable with the wrappers is also achievable somehow using basic GG resources. For instance, instead of the wrapper, we could model Animal as a type of vertex and Bird/Bear/Snake/Wolf as types of edges, or other vertices connected to Animal. But **wrappers** help to ensure consistency and **provide a intuitive**, organized and explicit **way to model** those **abstractions**.

What is appealing about wrappers is that they are not just an additional resource that would be approached only when layers of abstraction is the subject or learning objective of the activity. They actually solve an inconvenience that is quite often found when modeling GG: common behaviors among different types. Because GG requires strictly same-type matches[3], when a novice models a GG, they are likely to go through the following path: making different types; wanting them all to do the same thing; then realizing that would require several similar rules, one for each type. That we see as **a natural way** to induce the learner **to understand** the reason and importance of **generalization**.

Furthermore, as information in a graph is represented by shapes in a space, we naturally try to manage their positions to fit the screen. As a graph grows, fitting all elements in the screen becomes harder and harder, eventually impossible without overlaps. Wrappers allow us to compact (hide) multiple elements into a single container, simplifying the view of the whole. That we see as **a visual way to induce the learner to** understand the reason and importance of **abstraction of data** and **modularization**.

We discussed the capabilities of wrappers to foster abstraction layers and related skills (generalization, refinement, modularization, visualization and navigation). However, we recall it is a tool within a game engine, not a fully fetched activity on its own. It enables the creation of a wide range of activities around those topics: from putting the students to model entirely new games relying on the support of wrappers to organize and modularize it; to making them play strategically pre-made games where navigating between layers of abstraction is crucial. Creating such activities and making them available is part of our future work, but also possible for anyone that downloads the platform[4].

Once these activities are at hand, the primary focus will be on empirically validating this proposal. We plan to conduct experiments in different educational settings to determine its effectiveness of fostering the ability: to model considering different layers of abstraction; refine; generalize; and modularize. We are also considering expanding a series of pedagogical agents [Veletsianos and Russell 2014] that have been proposed for the platform [da Silva et al. 2021], with one to guide users on how to properly use layers of abstraction, while keeping this interaction more humane, personalized and engaging.

[3]This can vary depending on the GG approached, but holds true for the one used in the platform.

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

References

Armoni, M. (2013). On teaching abstraction in cs to novices. *Journal of Computers in Mathematics and Science Teaching*, 32(3):265–284.

Avila, C. M. O. and da Costa Cavalheiro, S. A. (2022). Practical guide for designing activities that integrate curricular content, computational thinking and constructionist theory. In *Proceedings of the XXXIII Simpósio Brasileiro de Informática na Educação*, pages 208–219. SBC.

Busatto, G., Kreowski, H.-J., and Kuske, S. (2005). Abstract hierarchical graph transformation. *Mathematical Structures in Computer Science*, 15(4):773–819.

da Silva, J. V., da Silva Junior, B. A., Foss, L., and da Costa Cavalheiro, S. A. (2021). Gramers: Agentes pedagógicos para uma plataforma de jogos baseada em gramática de grafos. In *Proceedings of the VI Workshop-Escola de Informática Teórica*, pages 80–87. SBC.

Ehrig, H., Korff, M., and Löwe, M. (1991). Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *Graph Grammars and Their Application to Computer Science: 4th International Workshop Bremen, Germany, March 5–9, 1990 Proceedings 4*, pages 24–37. Springer.

Ezeamuzie, N. O., Leung, J. S., and Ting, F. S. (2022). Unleashing the potential of abstraction from cloud of computational thinking: A systematic review of literature. *Journal of Educational Computing Research*, 60(4):877–905.

Ferreira, A. P. L. (2005). *Object-oriented graph grammars*. PhD thesis, Universidade Federal do Rio Grande do Sul.

Ferreira, A. P. L., Foss, L., and Ribeiro, L. (2007). Formal verification of object-oriented graph grammars specifications. *Electronic Notes in Theoretical Computer Science*, 175(4):101–114.

Gautam, A., Bortz, W., and Tatar, D. (2020). Abstraction through multiple representations in an integrated computational thinking environment. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 393–399.

Kakavas, P. and Ugolini, F. C. (2019). Computational thinking in primary education: A systematic literature review. *Research on Education and Media*, 11(2):64–94.

Mirolo, C., Izu, C., Lonati, V., and Scapin, E. (2022). Abstraction in computer science education: An overview. *Informatics in Education*, 20(4):615–639.

Qian, Y. and Choi, I. (2022). Tracing the essence: ways to develop abstraction in computational thinking. *Educational technology research and development*, pages 1–24.

Silva Junior, B. A., Cavalheiro, S. A. C., and Foss, L. (2021). GrameStation: Specifying games with graphs. In *Proceedings of the XXXII Simpósio Brasileiro de Informática na Educação*, pages 499–511, Porto Alegre, RS, Brasil. SBC.

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.

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.

Tikva, C. and Tambouris, E. (2021). Mapping computational thinking through programming in k-12 education: A conceptual model based on a systematic literature review. *Computers & Education*, 162:104083.

Veletsianos, G. and Russell, G. S. (2014). Pedagogical agents. *Handbook of research on educational communications and technology*, pages 759–769.

Vieira, J. M. F. and Zaina, L. A. (2021). Learning trajectories visualizations of students data on the computational thinking context. In *Proceedings of the XXXII Simpósio Brasileiro de Informática na Educação*, pages 705–717. SBC.

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 A: Mathematical, Physical and Engineering Sciences*, 366(1881):3717–3725.