A Playful Programming Products Vs. Programming Concepts Matrix

Allsopp, Benjamin Brink

Published in:
Proceedings of the 11th European Conference on Game-Based Learning

Publication date:
2017

Document Version
Accepted author manuscript, peer reviewed version

A Playful Programming Products Vs. Programming Concepts Matrix

Benjamin Brink Allsopp
Department of Learning and Philosophy, Aalborg University, Copenhagen, Denmark
ben@learning.aau.dk

Abstract: A number of Danish primary schools are involved in pilot studies where 1st to 9th grade students work with Scratch and Lego MindStorms in STEM subjects. These games may become part of the curriculum at these schools. Recent research identifies a category of games and toys that support learning to computer program: playful programming. This research also describes a project to bring together different stakeholders (developers, educators, parents and researchers) with a common vocabulary for describing, developing, teaching with and comparing these playful programming products and develops a model of supported programming concepts that can be used to differentiate these products. The model includes programming concepts like operators, procedures, encapsulation, variables and events, but also identifies numerous specialization and sub-specializations of these concepts. This paper aims to use this model to provide educators and researchers involved in pilot studies with an overview of which programming concepts various playful programming products exercise (a playful programming products vs. programming concepts matrix). We also add additional concept specializations and expand on the descriptions of concepts included in the model. Finally we consider how this work can be used to support a crowed-sourced playful programming research program.

Keywords: Playful programming, Programming Concepts, Block programming, Concept specialization maps, Product comparison matrix

1. Introduction

There is growing interest in equipping current and future school children with the skills that will serve them in an increasingly digital society, and much resent research focuses on developing computational thinking (e.g. Guzdial & Soloway, 2008; Resnick et al, 2009; Repenning, Webb & Ioannidou, 2010; Brennan & Resnick, 2012). However learning programming is difficult and requires complex abstractions, overview and mathematical insight (Misfeldt & Ejsing-Duun, 2015). We can fear that teaching programming as we teach other subjects at school may not generate sufficient interest and confidence for most children. One growing option is to leave the traditional school environment. This is seen in coding clubs like Coding Pirates where parents, volunteers and children come together in programming related activities that the children have considerable autonomy over. These environments typically have one adult dedicated to only a few children and seem to be growing in popularity, but it is unclear how scalable this approach is.

A different approach, which is sometimes practiced at coding clubs as well as in schools and at home is to use programming games that allow much of the learning to be guided by the game. A number of Danish primary schools are involved in pilot studies where 1st to 9th grade students work with Scratch and Lego MindStorms in STEM subjects (Jensen, Hanghøj & Misfeldt 2016) and Allsopp & Ejsing-Duun (2016) describe a growing number of: "computer games, robotic toys, programming apps and other software products (and even board games) … that attempt to turn learning to program into play." There are two assumptions in this approach: playful programming products support perseverance and motivation, and playful programming products actually teach programming. Perseverance and motivation are hard to maintain when learning programming, but are precisely what many games seem to support. Likewise, the playful programming products all promise to teach programming, and spending any time with these products seems to confirm this. However, educators need to know more specifically what these products teach. Brennan & Resnick (2012) identify three aspects of computational thinking that they argue are supported by Scratch: computational concepts, computational practices and computational perspectives. However there are dozens of similar products, and educators need to be able to choose between them based on more specific understandings of what they teach. The primary aim of this article is to address this need by comparing nine different playful programming products using a fine-grained list of programming concepts.

2. Prior Work

There is extensive literature on teaching programming to children and adults (e.g. Robins, Rountree & Rountree 2003), but much less on using games to teach programming. This article relies heavily on one article

that explores this: Allsopp & Ejsing-Duun (2016) coin the term *playful programming* to describe the use of games in learning to program and *playful programming products* to describe the growing number of games, toys, and other software that promise to make learning programming more playful. They acknowledge this promise, but also acknowledge a lack of clarity on what the products can do and what their underlying didactic and pedagogical strategies are. More specifically for this article they point to a lack of clarity on the aspects of computational thinking that they aim to teach, and what specific parts of these aspects they exercise. They argue that this lack of clarity creates problems for educators wishing to choose a product and having to rely on the developers' own descriptions of the products which may be exaggerated or otherwise hard to compare with other products. They see addressing this problem as part of a larger project with the goal of "bringing together different stakeholders (developers, educators, parents, learners and researchers) with a common vocabulary for describing, developing, teaching with and comparing products" (Allsopp & Ejsing-Duun 2016). Not only will this help stakeholders choose products, but hopefully it will also support developers to understand how they can better support users in learning to program.

2.1 The programming concepts list

The specific goal with Allsopp & Ejsing-Duun's (2016) article was to create a list of programming concepts that would allow stakeholders to identify differences in what different products support. Previous lists of programming concepts for learning programming exist. For example Brennan & Resnick (2012) provide a list of computational concepts in their broader exploration of computational thinking. However Allsopp & Ejsing-Duun (2016) aim for a more fine-grained list that allows them to identifying subtle differences in what different products support. Their ambition is to achieve enough granularity in the concepts to be able to categorise products without using descriptions of the degree to which, or how, they exercise a programming concept, but simply with a binary marker indicating that they do or do not exercise a specific concept.

At the same time, they did not want to include too many concepts, but to keep the list relevant for comparing playful programming products by grounding it in observations from actual products in the category rather than in more general descriptions of programming languages. There actual observations where from only two products Scratch and Lightbot (both the Web and the 9+ version), and they acknowledge that examining more products may improve on it, but argue for it being relatively stable.

2.2 Concept specialization maps

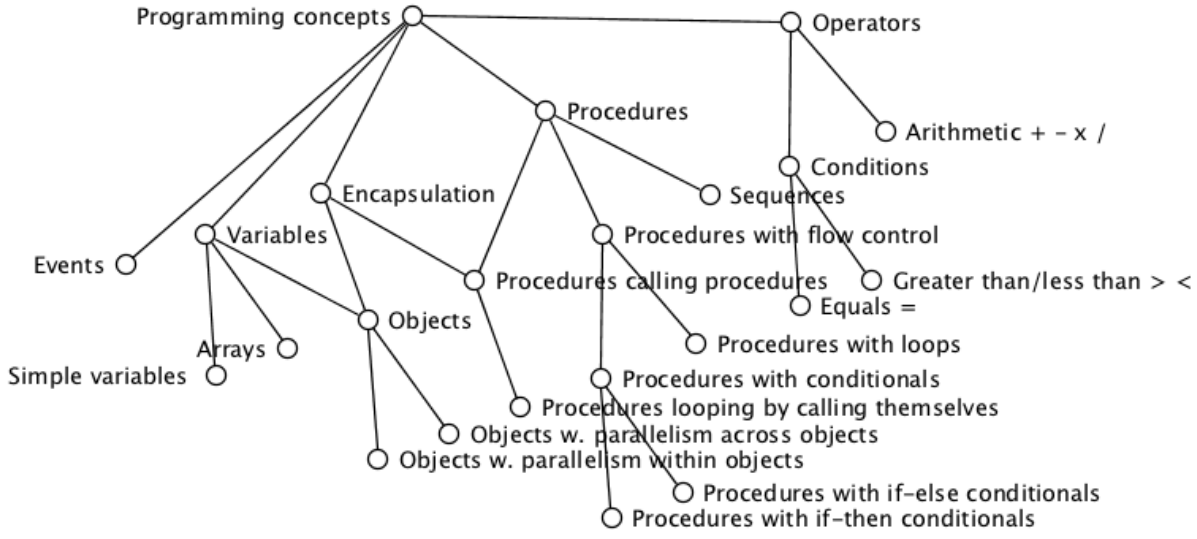

Figure 1. A concept specialisation map (CSM) showing all the programming concepts identified by Allsopp & Ejsing-Duun (2016)

Of particular interest is the way Allsopp & Ejsing-Duun (2016) analyse their observations using a new visual notation called *concept specialization maps* (CSMs), which is exemplified in Figure 1. Because the goal of the current article is not just to use the list, but if possible improve it, it is important to understand these maps. CSMs are designed to reduce ambiguity between concepts by spatially arranging them to better see when they

overlap, and when they differ. More specifically, CSMs are directed graphs with nodes and arcs that point between nodes. They are acyclic, or partially ordered, in that no arcs go *upstream*. Each arc points from one concept, which is considered more general, to a concept, which is considered more specific. These representations where used to visualize tentative understandings of the relations between programming concepts and where modified iteratively as new programming concepts were identified. Figure 1 shows the CSM developed by Allsopp & Ejsing-Duun (2016), in a version that consolidates all the changes identified in their article. This acyclic graph was flattened to become their list of programming concepts.

3. Our Approach

In this article we adopt an inclusive understanding of playful programming products as introduced in Allsopp & Ejsing-Duun (2016), and use their list of identified programming concept. Here we identify the concepts that each product exercises and register these observations in a matrix connecting programming concepts to each product. When forced to, we update the model and reconsider previous products against updates to the list. However, although we update the model iteratively when identifying new concepts, for reasons of clarity when presenting our findings we will describe the populating of the matrix and the updating of the map separately.

With only one exception all the playful programming products were installed and/or accessed online. These, also with only a few exceptions, were played to completion. In the few situations where a product was not played, or not played to completion, we were able to view online videos of all levels of the games. While working through the products in this way we considered each new programming challenge and code block against the concepts list. In the vast majority of situations the matrix could be updated without needing to reflect on the list or the underling concept specialisation map. On a few occasions however, a playful programming product exercised something that seemed like a programming concept, yet could not be satisfactorily recognised as one of the existing concepts in the list. This could either be because it seemed to do more or less, or both at the same time, than an existing concept. In these situations it was necessary to introduce a concept, but not simply by adding it to the list. We first needed to examine the CSM to find an appropriate place for it, and if necessary reconsider the existing concepts and their relations. Working with the CSM map forced us to understand each concept as part of a coherent understanding of all the concepts together. This way we avoid arbitrarily defined concepts that promote a specific product.

3.1 Method

As in Allsopp & Ejsing-Duun (2016), our methodological foundation for this work is inspired by Lakatos' (1976) philosophical work on mathematical proofs. We invite social falsification by opening up our observations and analysis to peer scrutiny. Specifically the CSM and the comparison matrix play the same role as a mathematical notation in a mathematical proof; they provide transparency into, and aid the interrogation of, our reasoning.

4. More Products

In the following we report on observations when examining six new playful programming products. In choosing the products we choose to look at two of three clusters identified by Allsopp & Ejsing-Duun (2016). These where navigation games and visuospatial authoring tools which both can be called block programming. The third cluster consists of environments that provide support or motivation for regular text based programming. Besides this, the products were casually chosen and by no means exhaustive of the category. The individual assessments of which products exercise which concepts are presented in the completed matrix shown in Figure 2. In this figure we see three visuospatial authoring tools (Scratch, Microsoft Block Editor and Hopscotch) followed by six navigation games (Lightbot (Web and 9+), Kodable, Lego Bits & Bricks, Coding Pirates and Cargobot). The programming concepts do not completely match those identified by Allsopp & Ejsing-Duun (2016), but the small modifications will be discussed under *Updating the map*, but before this we discuss the new products in turn.

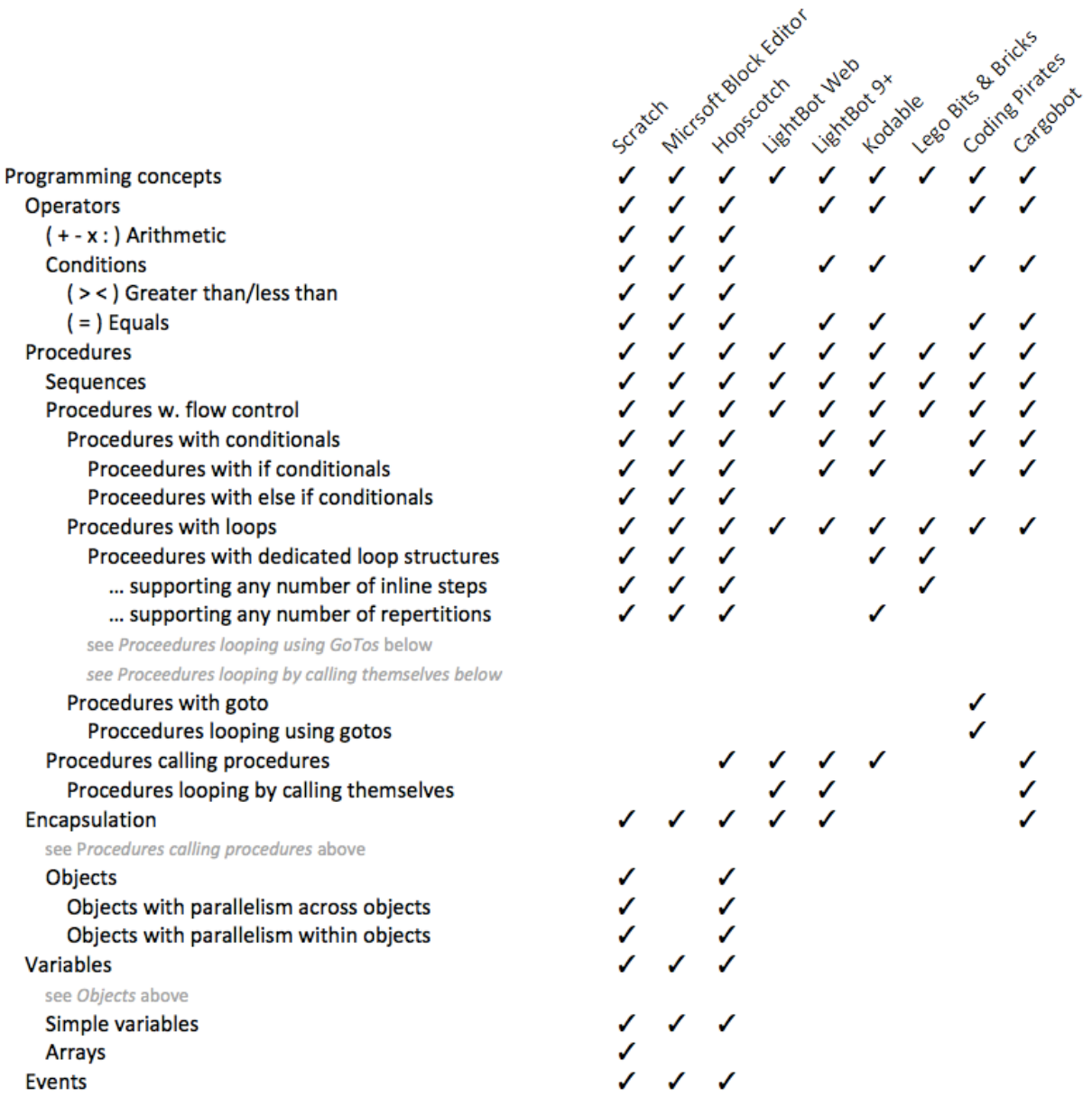

Figure 2. The completed products vs. concepts matrix

4.1 Visuospatial authoring tools

The examined visuospatial authoring tools all resemble Scratch in that the user chooses blocks from extensive, pallets and places them onto a canvas that can support more than one procedure. The steps in a procedure are arranged from top to bottom.

4.1.1 Microsoft Block Editor

We explored Microsoft Block Editor (MBE) as provided as a programming interface for the BBC Micro:bit; a pocket sized embedded device for children to build internet of things systems. The Micro:bit approach is interesting because it forces the programmer to think beyond developing typical screen interactions and explore alternative connections between the digital and the non-digital. MBE thus includes blocks representing the various input/output capabilities of the hardware. These include accessing an array of LED lights, a compass, an accelerometer and more, including a number of ports to connect to other electronic hardware. These I/O blocks have names like “show LEDs”, “compass heading” and “on shake”, and clearly differ from those available in the standard Scratch environment, however when these blocks are ignored and the remaining blocks are considered in terms of the programming concepts exercised, it is almost indistinguishable

from Scratch except for in two ways. MBE does not support any blocks which allow the defining of arrays, and MBE (at least in the micro:bit version considered here) does not seem to allow the creating of objects, beyond the default object which is the micro:bit device.

4.1.2 Hopscotch

Hopscotch is designed for the iPhone and iPad and was a pleasure to use, arranging blocks neatly in the canvas with a significantly simpler interface than Scratch. However Hopscotch supports almost all the same programming concepts as Scratch. Like MBE is does not support arrays, but it allows the creation of objects with parallelism within and across them. It differs from both Scratch and MBE by allowing the *encapsulation* of procedures, and *procedures calling procedures*.

4.2 Navigation games

All of the navigation games consist of adding blocks left to right in an instruction area to control the movements of an avatar (normally through a maze). None of these games exercise *variables* or *events* as programming concepts.

4.2.1 Kodable

Kodable is an extensive collection of learning materials, which promises a smooth transition into JavaScript code. However we will only look at what is exercised in an included navigation game. Where all other navigation games considered here require one block to move each step in the maze, a single arrow block in Kodable will make the avatar continue to move in a direction until it is stopped (for example by a wall). This makes it important to introduce conditionals earlier to interrupt these movements. Like Lightbot 9+ conditionals can be triggered by coloured squares in the maze. One way this is different from more advanced conditionals, seen in for example Scratch, is what conditions are supported. Kodable only supports the (=) *equals* condition triggered when the current squares colour is *equal* to a specific colour. Another way that its conditionals are different to Scratch, is that it does not support *if else* conditionals. Kodable supports *procedures with loops*; more specifically *procedures with dedicated loop structures*. These are loop structures *supporting any number of repetitions*, but are restricted to looping through only two blocks of code. Finally Kodable, like Lightbot, supports *procedures calling procedures*, which are called functions and are represented with a block labelled "{}", however unlike Lightbot it does not allow procedures to call themselves.

4.2.2 Lego Bits & Bricks

Lego Bits & Bricks is very similar to Lightbot in its scenario of a bot needing to navigate a maze. We completed all 16 levels and our overall impression is that the game was fun and challenging, but this may have more to do with the designs of the mazes than to do with the programming concepts exercised; the mazes would also be fun to solve by moving the bot directly. Unlike the previous navigation games, this game requires each placed block to be tagged with a digit (1 to 9) for the number of times it is to be repeated. The ways that different games implement a movement block are not considered programming concepts and Bits & Bricks does not support many coding concepts. It supports *procedures with flow control*, but no conditional flow control; in the game play, crossing bridges or taking portals in the maze depends on the bot stepping on coloured squares or pulling levers, but there is no option to specify conditional steps. Without conditionals there is also no need for conditions, or operators in general. Bits & Bricks does however support *procedures with loops* and even *procedures with dedicated loop structures*. This dedicated loop structure is shown in Figure 3. The structure is different from Kodable's dedicated loop structure in that it supports any number of blocks to be inserted in the loop *(… supporting any number of inline steps*). It is however also limited in that, like other blocks in Bits & Bricks, it takes a digit tag and only supports 1 to 9 reparations.

Figure 3. Bits & Bricks' dedicated loop structure

4.2.3 Coding Pirates

The game, Coding Pirates, is developed by the Coding Pirates code club. We experienced our initial signing in to Coding Pirates as clunky and perhaps even buggy. We have not considered the game's custom maps, multiplayer options, or level editor, but we played all levels. At first it resembles Lightbot with simple blocks representing a single move. It also resembles Lightbot 9+ with respect to how it exercises *procedures with conditionals*. Different blocks in the instructions check if the avatar (this time a pirate) is standing on something (sand, grass etc.), sees something (tree, rock etc.) or is holding something (a key, treasure), before continuing the instructions or moving down to an alternative instruction. Although there are many different things that can be checked these correspond to the *(=) equals* condition. An example condition could be read as: "what the pirate is standing on equals sand". Like other navigation games the user is limited to *procedures with if conditionals*. Where coding pirate really stands out is in how it supports *procedures with loops*. Coding Pirates provides a block with a chain icon on it. This could suggest a dedicated loop structure, however it is more than this. By placing a chain block the instruction area and then clicking on it the user is able to select any other block in the specification. A directed arc is drawn from the chain block to the selected block to indicate that it will be executed next. This is a different and more general type of *procedure w. flow control* than those with loops. It resembles the goto statements seen in older programming languages and thus Coding Pirates exercises *procedures with gotos*. Only when we use this block to point to a block earlier in the specification do we have procedures with loops, specifically *procedures looping using gotos*. With the goto block and simple conditions Coding Pirates supported some really interesting coding challenges.

4.2.4 Cargo-Bot

Cargobot is an iPad game that is similar to the other navigation games in that the user drags blocks onto an instruction area. However it differs from the navigation games in that users do not specify an avatar's path through a maze (going forward and turning left or right). Instead they control a crane that can only move left or right along a rail and pick up or drop off crates, which are placed in fixed positions under the rail. We played a number of screens of Cargo-Bot, but then watched a fast-forwarded video recording of the completion of all levels (see link in references) to identify additional blocks and how they were used. Despite their differences Cargo-Bot resembles Lightbot 9+ in how relatively few types of blocks exercise many general programming concepts. Like Lightbot, *procedures calling procedures* are supported with a numbered block directing execution to a numbered alternative instruction area. Again like Lightbot, *procedures with loops* are supported as *procedures looping by calling themselves*. *Conditionals* are also supported similarly to in Lightbot 9+ as *procedures with if conditions*. Here a block may be tagged with a colour so that it only executes if the crate held by the crane has the same colour. The conditions supporting these conditionals are, like the other navigation games, limited to the *(=) equals* concept. Finally, it is worth noting that like Lego Bits and Bites, the game's challenges often preceded decisions on applying programming concepts. Both games required thinking that would also be necessary even if the user controlled the bot or crane directly with say a joystick.

5. Updating the CSM

Some of the new products exercised programming concepts where not identified by Allsopp & Ejsing-Duun (2016). Updating the concept list requires understanding how these new concepts relate to existing concepts, and this has depended on understanding how to update the CSM to accommodate the required concept. There are three kinds of modifications: changes to the naming of existing concepts, the adding and connecting of a new concepts and the changing of specialisation relations between existing concepts. The changing of names has been relatively limited and focused on using more conventional terminology. For example the term "if else" replaces the term "if-else". One simple example of the adding and connecting of a new concept occurred when we observed that Coding Pirates supported a block that exercised the creating of *procedures with gotos*. This is considered a different way of supporting *procedures with flow control* than both *procedures with loops* and *procedures with conditionals*. This could therefor be added to the CMS as a specialisation of *procedures with flow control*. Beyond this change, all other changes to the CMS involve specialisations to *procedures with loops*. All changes are emphasised in the CMS shown in Figure 4, the remaining of which are discussed below.

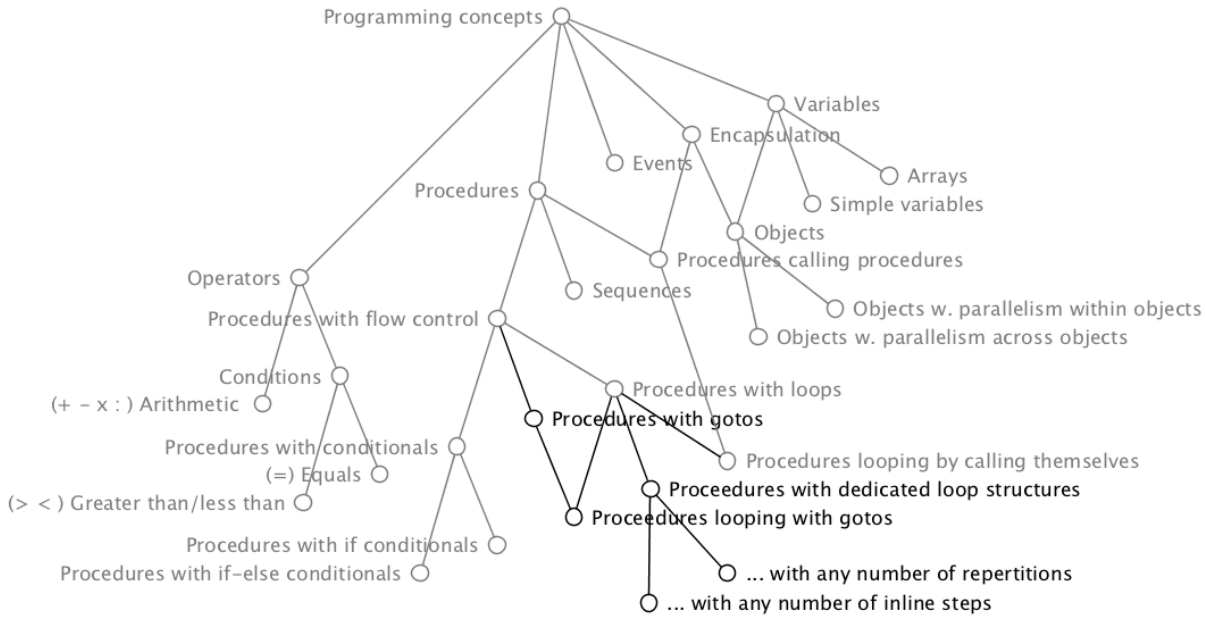

Figure 4. The updated CSM with the new specialisation and programming concepts emphasized

5.1 Specialisations to procedures with loops

Procedures looping with gotos seem to be a way to implement *procedures with loops*, however it is not simply a specialisation of what was understood by the concept that Allsopp & Ejsing-Duun (2016) identified from Scratch and called "procedures with loops". Scratch supports special loop blocks that enclose the repeated code, while gotos allow jumping to any step in a procedure. Looping with gotos is not a special case of what Scratch does. What is then the best way of understanding *procedures looping with gotos*, as a specialisation of *procedures with loops* or not?

It is in situations like this where concept specialisation maps can show their strength. It is possible that in normal conversation or writing we would either ignore the dilemma or offer some sort of loose explanation about how looping with gotos is similar to, but not strictly speaking the same as procedures with loops. While in CSMs we are forced to either connect or not connect two concepts, and we must rethink our assumptions until we can do this. Upon deeper reflection and with the representational support of the CSM a satisfactory precisioning was found. It turns out that Scratch's way of supporting loops, while perhaps the most obvious or common in popular programming languages can be seen as a specialisation of a more generic concept, which can keep the position of *procedures with loops* and even keep its name. A new concept, *procedures with dedicated loop structures* is drawn as a specialisation of the original concept and describes what is special about Scratch, the other visuospatial authoring tools, and the navigation games: Bits and Bricks and Kodable.

Now it is meaningful to draw procedures looping with gotos as both a specialisation of *procedures with gotos*, and *procedures with loops*. It is just a different specialisation of the latter than *procedures with dedicated loop structures*. The introduction of the *procedures with dedicated loop structures* specialization also allows a similar drawing of a concept as a specialisation of two other concepts. It is now possible to draw a specialisation arc from procedures with loops to procedures looping by calling themselves. Previously this arc would not have been appropriate because is would have suggested that the looping seen in LightBot was a special type of the looping seen in Scratch.

Finally it was necessary to distinguish between two different types of dedicated loop structures. Those seen in the editors and the more restricted ones seen in Bits and Bricks and Kodable. Here it was possible to distinguish the editors by both exercising procedures with *dedicated loop structures supporting any number of inline steps* and *procedures with dedicated loop structures supporting any number of repetitions*. The dedicated loop structures seen in Bits and Bits and Kodable can be distinguished from the visuospatial authoring tools by only exercising one of these. They can be distinguished from each other in that they differ on which specialisation they exercise.

6. Discussion

The usefulness of the matrix has not been explored empirically. We can currently only ask as researchers if we find it convincing that educators will use it. This can be split into two parts:

a. Would we as researchers, who have read the above article be able to differentiate between two products?
b. Would ordinary educators pressured with a choice between two playful products to use in class be able to do the same?

The first question seems unproblematic; an understanding of the CMS provides an understanding of the different programming concepts and the matrix allows us to immediately compare two or more products with respect to which programming concepts they support. The second question is more problematic. The programming concepts are disambiguated by their name and their relation to other programming concepts in the CSM. However beyond the descriptions in this article and Allsopp & Ejsing-Duun (2016), the concepts have not been provided with textual description. This will probably be necessary if educators are to be able to understand the available programming concepts in a product.

6.1 Looking ahead

The above could be addressed with the publishing of these descriptions; however, looking ahead, the writing of these descriptions should ideally be an on-going process that may be more suited to a wiki format than print or e-journal publication. The idea of opening up this work to online collaboration ties in well with the broader goals for the playful programming project Allsopp & Ejsing-Duun (2016) where the ambition is to bring together developers, educators, parents, learners and researchers. This bringing together of practitioners need not just be for them to consume research, but for them to be the participants in further research.

A Lakatos understanding of science (involving social falsification) requires the opening up of claims to scrutiny (Lakatos 1976), and the maintenance of the right sort of community to provide that social scrutiny. In a subject like mathematics this has consisted of various mathematical notations that make reasoning errors more visible, and of communities of mathematicians (connected through universities and journals) that are able to scrutinise and respond to this reasoning. In a playful programming research program, both the scrutiny and the community can be supported online with purpose built infrastructure allowing a form of crowed-sourced science. In the specific current case of identifying programming concepts and gathering observations of individual products exercising individual programming concepts, interactive online versions of the CSM and products vs. concepts matrix can be an important part of that infrastructure. Imagine if the clicking on any concept node or specialisation arc between concept in the CSM, or any cell in the matrix, opened up an online threaded discussion forum dedicated to that token. Any practitioner could add highly focused evidence and analysis to the on-going understanding of playful programming products and the programming concepts that they exercise.

7. Conclusion

This article has continued work initiated by Allsopp & Ejsing-Duun (2016). It has examined (or reexamined) nine playful programming products to determine the underlining programming concepts that they exercise, and mapped the resulting observations in a products vs. concepts matrix. It has used concept specialization maps (CSMs) to analyze and further develop a model of the interrelations between programming concepts needed to differentiate playful programming products and to update the list of programming concepts. The list is showing signs of stability in that the changes to the CMS due to examining the additional products are significantly smaller than the changes due to examining the original two products. More work is required to make the matrix accessible and comprehensible to educators needing to make choices between playful programming products, but it is argued that this is not only doable, but that the CMS and matrix can be part of an interactive infrastructure for supporting crowd-sourced science into playful programming products.

References:

Allsopp, B. B. and Ejsing-Duun, S., 2016. Programming Concepts in Playful Programming Products. In *Proceedings of the 10th European Conference on Games Based Learning*: ECGBL 2016, (p. 1).

Brennan, K. and Resnick, M., 2012, April. 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.
Guzdial, M. and Soloway, E., 2002. Teaching the Nintendo generation to program. In *Communications of the ACM*, *45*(4), pp.17-21.
Jensen, E. O., Hanghøj, T., and Misfeldt, M., 2016. Game Design and Development as Mathematical Activities: Proposing a Framework. In *Proceedings of the 10th European Conference on Games Based Learning: ECGBL* 2016 (p. 296).
Lakatos, I., 1976. *Proofs and Refutations*. Cambridge University Press.
Repenning, A., Webb, D. and Ioannidou, A., 2010, March. Scalable game design and the development of a checklist for getting computational thinking into public schools. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 265-269). ACM.
Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y., 2009. Scratch: programming for all. *Communications of the ACM*, *52*(11), pp.60-67.
Robins, R., Rountree, J. and Rountree N., 2003. "Learning and Teaching Programming: A Review and Discussion". *Computer Science Education, vol. 13, no 2*, Routledge.

Andy G (2012) Cargo-bot full walkthrough, YouTube
https://www.youtube.com/watch?v=r_DRHuKCEMU&ab_channel=AndyG