# Teaching Array Verification Using Snap!

DAMIAN VERHEIJEN, University of Twente, The Netherlands

Working with arrays is a vital skill for any programmer. Functions that use arrays can be complex, making it hard to verify the correctness of the function. Program verification can be used to check the functionality of a program more thoroughly. Providing novice programmers with the knowledge to use verification techniques could greatly improve the quality of their programs. Snap! is a visual programming language tailored to high school students. In this research project, we propose a library for arrays as an extension of VerifiedSnap!. VerifiedSnap! is an extension made for Snap! that allows it to support both runtime and static verification techniques. Exercises to go along with the designed tool were designed to help teach high school students how to use arrays and verification techniques. A small empirical study was carried out to determine how useful the designed tool and exercises would be in teaching students. The results of the study are inconclusive, but suggestions to improve this in the future are described.

Additional Key Words and Phrases: verification, arrays, education

## 1 INTRODUCTION

In the world of programming, arrays are considered one of the fundamental principles. Most programming languages support arrays, and they are used in most programming projects. Arrays have varied uses, therefore a good base understanding of how to use arrays safely is vital.

Mistakes when working with arrays can be hard to spot at times. Using verification techniques, programmers can confirm that a program is working as expected, and find bugs in complex programs [2]. Since arrays are so widely used, it is important that programmers are able to verify programs in which arrays have been used. For this reason, it is important to focus on teaching programmers specifically about the use of program verification in regard to arrays.

One approach for program verification is called Design-by-Contract (DbC) [8]. For DbC a program is split in multiple subroutines or functions. Each subroutine is assigned specifications that are used to define the requirements the subroutine needs to fulfil. In DbC specifications are defined using annotations for the given function or subroutine.

There are four ways to describe annotations that specify the behaviour a function needs to follow when applying DbC. There are pre-conditions, which are used to describe rules that need to be true before the function is executed. Then there are post-conditions that define what effect the function had, this can be both based on the output of the function or the current state of the program.

In addition to these, you can also have annotations within the function. Loop invariants are annotations connected with a loop within a function. At no point during the execution of the loop can these annotations be violated.

*TScIT 37, July 8, 2022, Enschede, The Netherlands*

There are also asserts. An assert statement can be put at any location within the function. Asserts can be used to check at a point within a function a certain rule is still followed. Big functions sometimes have multiple subroutines, so at certain points a programmer can put asserts to check the state of the program at key points during the execution. Checking the state like this can help assure that the program is still on the right track and make it easier to find out where a program went wrong.

When the program has been written, it needs to be confirmed that the program indeed satisfy the assigned specifications. This can be done in two ways. One of these ways is by verifying the program during runtime. Hereby, test cases defined by the programmer are used. If, at any point during these tests, one of the specifications are violated, the test fails, indicating that the program has a mistake. Note, that runtime verification does not verify all possible states in the program. Meaning, that if the right test cases are not used, bugs can still be missed.

Alternatively, deductive verification techniques can be used, also referred to as static verification. With deductive verification, logic is used to deduce all possible states the program can reach. Through this, a situation can be discovered where a state can be reached that does not satisfy all defined specifications, if such a state exists.

The existence of program verification techniques will not help programmers in spotting errors if they do not know how to use them. Teaching this to novice programmers could greatly enhance the quality of the programs they develop.

Huisman et al. have created VerifiedSnap! to help teach high school student about DbC [6]. They created an extension on the already existing program Snap!, which is a visual programming language [4]. VerifiedSnap! adds support for runtime verification and static verification into Snap!.

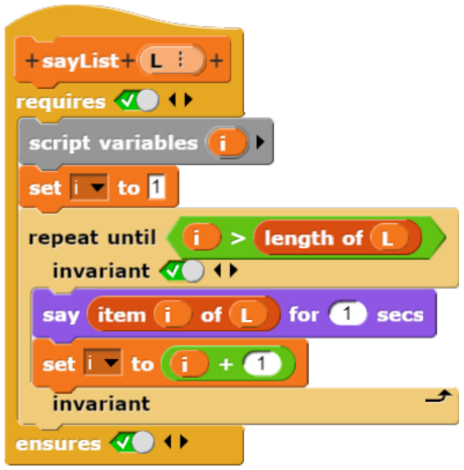
Snap! does not support arrays and therefore VerifiedSnap! can not be used to teach about array verification. Through this paper, a library supporting arrays was designed as an extension of VerifiedSnap! building upon the previous work of Huisman et al. In order to do so, a data type for arrays was added to Snap! and support for the verification of this in VerifiedSnap!.

In combination with the tool, a lesson plan was designed to teach students how to think about the bounds of their arrays and how to verify their implementation using DbC. This is done through exercises that make use of the extended version of VerifiedSnap!. The exercises should highlight problem statements surrounding arrays students can be faced with, so they can learn how to catch and handle these.

From the creation of this lesson plan, the following research questions were formulated:

**RQ1**: What are the main mistakes students need to be aware of when dealing with arrays.

**RQ2**: How effective can an array library as an extension of VerifiedSnap! be used to teach students about correctly programming arrays.



```
+sayList+(L ⋮)+
requires ✓ ◂▸
  script variables (i) ▸
  set i ▾ to 1
  repeat until (i) > length of (L)
    invariant ✓ ◂▸
    say item (i) of (L) for (1) secs
    set i ▾ to (i) + (1)
  invariant
ensures ✓ ◂▸
```

Fig. 1. Example of a BYOB block in VerifiedSnap!

## 2 BACKGROUND

Snap! is a reimplementation of Scratch, both are visual programming languages. Scratch was created as a tool to get children enthusiastic about programming by making it visually appealing [10]. Scratch is mainly targeted at children below high school age, though it is used by other age groups as well.

Scratch, and therefore Snap!, uses draggable blocks as function definitions. Programmers can drag these blocks around to the main program screen, snapping them together to create a program. Dragging around blocks instead of writing code might be more familiar to computer users, making it less daunting to be introduced to programming.

Snap! was developed with the aim to be used to teach programming to high school or college students. Snap! is similar to Scratch, but tailored to be more suitable for high school students. Snap! has additional features over Scratch, which makes it more useful to teach programming fundamentals [3].

Blocks in Snap! can have one of three types. A command block carries out an action, the shape of a command block has a missing rectangle at the top

. and this is added at the bottom allowing multiple command blocks to be slotted together, allowing a sequence of command blocks to create a program.

There are also reporter blocks, a reporter block calculates a value and returns it. A reporter block can be recognized by the rounded shape

.

Lastly, there are predicate blocks, these blocks return a boolean. Predicate blocks are the only blocks that can be dragged into diamond shaped slots, therefore telling the users that at specific places they can only use predicate blocks. These blocks are very useful for verification too, they are used by the user to describe the annotations. To indicate that predicate blocks are associated with the diamond shaped slots, they are also diamond shaped

.

Snap! has a Build Your Own Block (BYOB) feature not found in Scratch, this allows the user to define their own blocks. Similar to how functions are created in conventional programming languages. The BYOB function of Snap! is used by VerifiedSnap! to implement room for the specifications of the function. Figure 1 shows the implementation of a BYOB block.

When defining BYOB blocks in VerifiedSnap! there are two added sections that are not in Snap!. The *requires* section offers the programmer a slot to add boolean logic to define the pre-conditions. While the *ensures* section offers the same, but for post-conditions. A slot for loop invariants was also added. All these slots can be seen in Figure 1.

Studies have been carried out on the effectiveness of using Scratch in teaching programming skill to programming novices. The study conducted by Quahbi et al. found that a teaching method using Scratch helps in making students more motivated in learning programming [9]. The study conducted by Hijon et al. concluded that incorporating Scratch in a CS1 course, beginner level, significantly improves the understanding and knowledge of the students [5].

A study on the grasp programmers had of using the bounds of arrays found that many of the CS2 programmers, lacked the skills to come up with clever ways to work with loop bounds and solve problems related to working with bounds [1].

### 2.1 Current State of VerifiedSnap!

As mentioned in the introduction, VerifiedSnap! supports both runtime and static verification. Custom blocks can be used to define functions within the program. When a user defines a custom block, they can also add pre- and post-conditions, which are used to verify the program. Each time the custom block is run, VerifiedSnap! automatically checks whether the specified conditions are not violated. If any condition is violated, the program will throw an error to indicate to the user which one was violated.

For static verification, VerifiedSnap! uses Boogie. Boogie is an intermediate verification language intended to be used to represent programs in object orientated languages, enabling it to be checked by the software verifier also called Boogie [7].

Static verification is not done within VerifiedSnap!.Instead, a custom block can be exported as a Boogie program. The generated code can be run in Boogie to verify the program.

## 3 DESIGN DECISIONS

In this section, the most important design choices for the VerifiedSnap! extension will be highlighted and the reasoning behind the choices explained.

### 3.1 Array features

While working on this extension, it was realized that there are actually multiple understandings of what an array is. Different programming languages have different implementations of arrays, which makes it hard to pick one correct definition.

When choosing the features for the array block in Snap! it has been considered how relevant each feature was for the students. For each option, it was considered whether being acquainted with it would give the students more of an advantage in the future when working with different programming languages. In the sections below, some of these features have been highlighted and the choice for each feature that was implemented explained.

*Index bounds.* For lists in Snap! it is not checked whether a given index is out of bounds. This means that when an element is queried at an index that does not exist, no error will be thrown. Instead, VerifiedSnap! generates an object that acts like a neutral identity element that has no influence in most operations.

This means that further operations will not fail but instead produce wrong data, meaning that index out of bounds errors with lists are not always clear in more complex code. When teaching about bounds, it would be easier to have explicit errors related to bounds to avoid confusing the high school students.

It should be noted that in most programming languages the bounds of an array are checked, an error will be thrown making the user aware when an index out of bounds happens. However, a notable exception to this can be found in C, where no error is given and instead the value of a wrong memory address is returned, more similar to what is the behaviour of lists in VerifiedSnap!.

Considering the above information, the decision was made to also have our implementation of arrays in VerifiedSnap! check for this.

*Type support.* Lists in VerifiedSnap! allow for multiple different types to be put in a single list. Arrays generally only allow for a single type. In the implementation of arrays, it was decided to follow this convention.

For arrays in Snap!, upon declaration and when updating an element, it is checked whether all elements of the array are of the same type, when this is not the case an error will be shown. Indicating the determined type of the array, based on the first element in the array, and which wrongly typed element was found in the array.

*Counting from zero.* One of the conventions in programming is to start counting at 0. This is something that can be confusing for novice programmers since in most other situations one would start counting at 1. Lists in Snap! start counting at 1, it can be said that this is better. There are many new concepts taught to students, delaying the introduction of 0 based indexing can make it slightly less daunting to learn about indexing to begin with. t

However, when students initially learn about indexing starting with 1, they will have to unlearn this when they move to other programming languages. Confusing them once more and potentially inviting them to make thought errors since they learned to think about array bounds using the 1 indexing. Instead, allowing students to become familiar with counting to 0 early on might be more beneficial down the line, despite it being more confusing initially. Therefore, the decision was made to have arrays in VerifiedSnap! start with index 0.

*Static size.* Depending on the programming language, the size of an array can be determined on initialization (a static size), or it can be determined dynamically. Working with static sized arrays is more tricky, it requires thinking about the size an array needs to be beforehand.

Because of this, it was decided to have the size of an array be set on initialization. It forces students to be more aware of the bounds of their array, and it will still teach them how to reason about arrays even if down the line they are dynamically sized.

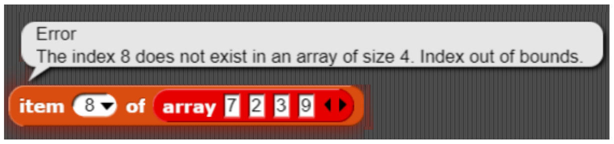

Error
The index 8 does not exist in an array of size 4. Index out of bounds.

Fig. 2. Error message shown when an index out of the range of the array is accessed.

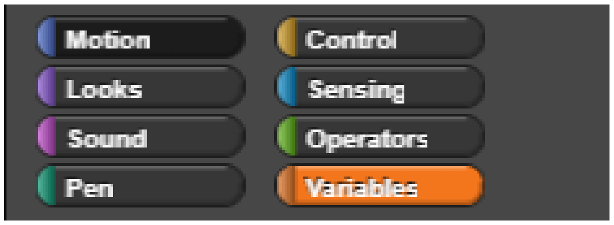


Fig. 3. The colours assigned to some of the block types

## 3.2 Error messages

VerifiedSnap! will be used to teach students basic skills on working with arrays and the verification of programs using them. When learning, it is natural to make mistakes. Receiving proper feedback on their work can make it easier for students to figure out where they went wrong and correct their understandings.

One way to help students in correcting their mistakes is giving clear error messages if their code does not work. If the error message is confusing, it might scare students and prevent them from figuring out what went wrong and what caused the error.

When writing the error messages thrown by arrays, an attempt was made to keep the language within the message simple and as descriptive as possible. And so giving users information they can both understand and use to fix their program.

Figure 2 shows an example of an Index out of bounds error, the message tells the user which index was tried to be accessed and the size of the array, giving a hint on what the range of the index should be. The name of the error is also explicitly mentioned, so the students can learn to recognize the error in the future.

## 3.3 Block colour

The main selling point of Snap! is that it is a visual language, allowing the users to drag around blocks that act as functions to the right spot in order to make programs. In Snap! each block has been assigned a colour. The colour of the block indicates the type of the block. A small overview of types and their colour mapping can be found in figure 3.

The design of the new array blocks needs to be chosen such that it fits in seamlessly within the program. Within Snap! the colour for variables is a light orange, as seen in figure 3. Within Snap! lists can be found in the same location as variables, lists and the function associated with lists have been given a slightly darker orange than the variables.

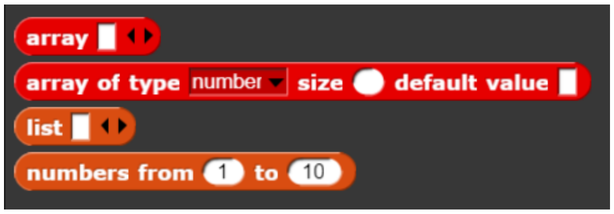


Fig. 4. The colour for lists next to the chosen colour for arrays.

Since arrays are similar to lists in Snap! it was decided to give the array blocks a somewhat similar colour to lists to express that they are similar yet different. As can be seen in figure 4 the chosen colour for array blocks is red. It was deemed similar enough that it fits in with the other colours, while being clearly different to indicate that they are of a different type.

### 3.4 Array functions

Despite the above-mentioned differences between lists and arrays, they are also similar in many ways. This means that arrays and lists have an overlap between some of their default block functions. For example, the *map* block, which applies a given operator to each element of a list, would work exactly the same for an array. At the same time, while there is an overlap between functions, not all default blocks for list can work with arrays.

For example, since the size of an array is static, it is not possible to add a new element to an array so the function block that adds an element to a list can not work for arrays.

The similarity and difference between list and array makes it difficult to determine the best way to add support to arrays through default function blocks.

It is possible to only edit the compatible list blocks to also work with arrays. However, if it is not clear that only some block work of arrays and some do not, the students might make assumptions that they can use all blocks for arrays in the same way as they can use them for lists and then be surprised that it does not work.

A way to avoid this issue is to make different blocks for arrays, making it clear that they are different blocks. This would allow students to better understand that blocks of a certain type only work for that type. However, since lists and arrays have an overlap in base functions, it can be a bit counterintuitive that the same function block has two obviously separated implementations.

As explained above, both options to deal with this issue have advantages and downsides, there is no clear better option. So since it was simpler to change existing blocks than adding new ones and because we wanted to keep the block list minimal, it was decided to change the existing blocks in the prototype.

## 4 EXERCISES

For designing the exercises, it was explored what sort of mistakes can occur when working with arrays. Virtually all mistakes that could be found were mistakes made when working with arrays seemed to be related to index out of error. In particular, students seem to struggle with Off-By-One errors. Off-by-One errors occur when a programmer makes a mistake with the bounds of an array and therefore misses one element of the array or tries out one element too many [11].

Because of this, it was decided to focus on understanding array bounds through the exercises. In order to decide which exercise would be used for the lesson plan at the end, multiple example programs were made and considered.

For an example program to be chosen for an exercise, it was considered how relevant and useful it was to teaching the basics of arrays and verification of the functions using arrays.

For the first exercise, fig 1, it was decided to have a simple program printing the elements of the inputted array. This exercise is meant as an introductory exercise showing off the difference between lists and arrays and showing the students an example of how to iterate over an array.

For the following exercise, it was chosen to use an example program that calculates the average of an array to introduce pre-conditions. Calculating the average can only be done over numbers, so introducing a post-condition for checking that the given array is indeed a number array can be more intuitive for the students, allowing them to understand better what the use of post-conditions are.

Further example programs were chosen because they calculated something easily verifiable later on, allowing the introduction of post-conditions without too complicated logic. One of these program gets the smallest number of an array and the other returns the sum of the inputted array.

Each exercise repeats the learning goal of the previous exercise to continue exposing the students to what they learned before allowing them to get more experience with the new concepts and therefore hopefully gaining a better understanding.

## 5 BOOGIE

As mentioned in previous sections, there are two type of verification methods. Run time verification and static verification. Run time verification happens in Snap! itself. The conditions for runtime verification are based on the existing predicate blocks, so implementing working arrays in Snap! is enough on its own to work. Static verification is done through Boogie, in order to allow verification through Boogie the VerifiedSnap! code need to be parsed into Boogie code. Through the previous work from Huisman et al. there is an existing framework for this [6]. In order to enable parsing, it needs to be defined for each block how it should be translated into the Boogie language. Since there was no time to define the Boogie representation for all default blocks in VerifiedSnap! some example programs had to be changed to only have blocks that already had an implementation. For students to learn about the static verification, it is not necessary that the Boogie parsing allows for all possible type of arrays. Therefore, it has been decided to only support arrays of numbers and booleans.

## 6 STUDY

In the study, we assess the usefulness of the extension as a tool to teach about arrays and the effectiveness of the exercises. This has been done through a small empirical study with a small group of people who have little to no programming experience using the prototype that was designed. We use the outcome of this study to reflect on the design decisions we made for the tool and the didactic effectiveness of the exercises we prepared for the students. Ideally, the study would have been done using high school students who already have had basic lessons in programming and verification through VerifiedSnap!, so they would be familiar with basic programming concepts, verification and VerifiedSnap! itself. However, a study like that was out of the scope for this project, and instead it was decided to carry out the study with people that have no prior programming experience, to try and simulate a teaching experience.

When the study was initially planned, the choice was between choosing participants with no prior programming experience, or students from computer science who already have understood these concept and have an affinity to programming. It was decided to choose the participants with no experience, as they were closer to the desired target group of high school students.

For the study, 7 participants were approached, each of them had no prior programming experience. Each of the participants was given the VerifiedSnap! environment alongside with the exercises. Then the participants were instructed to try and solve the exercises, they were encouraged to ask questions if they got stuck.

Instead of having multiple participants work on the assignment simultaneously, it was decided to have separate sessions for each participant. This mainly was to compensate for the lack of programming experience, giving them ample room to get concepts explained to them if they got stuck. In addition to this, it also allowed the understanding of the exercises and concepts to be observed as the participant worked on the exercises.

After working on the exercises, the participants were given a small questionnaire in which they were asked about how well they thought they understood the topics addressed in the exercises and whether they thought the exercises were clear.

## 7 STUDY RESULTS

While performing the study, it became very apparent that the chosen target group of people with no or very little programming experience was not suitable for this research. The gap in the required programming knowledge and the actual understanding of the participants was too big to reconcile.

Most participants struggled to understand the basic foundations of programming, things like variables and functions were completely foreign to them. Many of the participants did not understand the use of the introductory exercise, not understanding why the elements were printing or why you would want this to begin with.

Since the participants did not understand the basics of programming, the study was not very productive for evaluating the quality of the lesson plan. Since the participants did not grasp the basics, it was virtually impossible for them to work on the exercises related to verification.

As a result of this, none of the participants managed to finish all the exercises, with most of them giving up after the second exercise, after already getting many hints.

While working on the exercises, most of the participants failed to understand what the use of lists were and how the indexing can be used. When, after this, they were introduced to arrays, the difference between a starting index of 0 instead of 1 seemed to increase the confusion.

Most of the participants seemed to not understand what the iterator variable was for, or why it would need to increase within loops. They did not seem to understand that it would keep track of the location where the program currently was in the array and when faced with an off-by-one error they often changed it so that the iterator would no longer increase since they saw a -1 thinking that was how it needed to be fixed.

## 8 CONCLUSION

Throughout this research, we attempted to create a lesson plan to teach high school students about program verification when working with arrays by creating an extension of the visual programming language Snap!. The program can be found on https://gitlab.utwente.nl/fmt-student-projects/snapverifiedwitharrays. In this repository the exercises can also be found inside the subfolder *lessons* and then *arrays*.

For this, we explored common made mistakes when working with arrays. Most mistakes can be boiled down to mistakes when working with the bounds of the array. Using this conclusion, we designed some exercises to highlight the use of arrays and the bounds, alongside exercises to highlight the difference between lists and the implemented arrays.

To evaluate the effectiveness of the extension alongside with the exercises, which together created the lesson plan, a study was carried out giving participants with no prior programming experience the exercises to solve.

During the study, it was discovered that the choice of target group for the study was wrong. The lack of knowledge was too much for the study to be effective. Therefore, due to the experienced issue with the study, there is no definitive answer on how effective the designed lesson plan is.

In hindsight, it would have been better to choose computer science students who already understand the concepts taught in the lesson plan. Computer science students could simulate how the exercises would be received by high school students. However, it was not feasible to carry out an alternative study within the allotted time.

## 9 LIMITATIONS OF VERIFIEDSNAP

VerifiedSnap! in its current state, it is not finished. This section goes over the main features lacking in VerifiedSnap.

Foremost, VerifiedSnap! currently is unable to convert all code blocks into Boogie code. While the program does tell the user that it does not support the blocks when they are trying to compile their BYOB into Boogie. Adding Boogie support to the remaining blocks would allow creating more versatile programs and verify them using Boogie. https://www.overleaf.com/project/62a9f4e0d818f34a4d830fba Additionally, it is not possible to run Boogie code through VerifiedSnap!, instead the generated code will be downloaded to the user computer where they can run it. Unfortunately, the process of installing Boogie is complicated, and it needs to be run from the command line. Since the main goal for VerifiedSnap! is to be intuitive and user-friendly, having an extended program that is the opposite seems counterproductive. In the future, a way needs to be found to integrate Boogie more into VerifiedSnap! or at least make it more friendly.

Finally, when using verification techniques for checking arrays, there can be situations where the type of an array matters. Currently, in VerifiedSnap! there is no way to query the type of an array. It is possible to get the type by checking the type of the first element, since all elements in an array are the same, but it could be more intuitive to have a dedicated block for this.

## 10 FUTURE WORK

Building upon this paper, there are still questions that need to be answered and studies that can be carried out.

First of all, the study that was carried out did not help in answering the question since it was the wrong target group. Therefore, the same study should be carried out with a more suitable target group, potentially Computer Science students.

Additionally, as mentioned before, there was no room within the scope of this project for a large scaled study with high school students on the effectiveness of the designed program. Carrying out such a study in the future would be necessary, though it might be beneficial to first address the limitations mentioned in the previous section.

One of the questions that was encountered while developing the extensions for arrays was whether to use separate functions for array or to adapt the existing functions to work with arrays. It could be desirable to explore in future research which of these options will help students learn to understand how to use arrays better and how they differ from lists.

What was noticed in the study was that it seemed to be confusing people that lists and arrays started with different indexes. It might be better to keep this consistent between lists and arrays in VerifiedSnap!. Starting with indices from 1 might be more intuitive for novice programmers, but starting with 0 is the norm in programming. It could be explored which of these option it is better to start with when introducing novice programmers to the notion of lists or arrays.

## REFERENCES

[1] David Ginat. “On novice loop boundaries and range conceptions”. In: *Computer Science Education* 14.3 (2004). ISSN: 17445175. DOI: 10.1080/0899340042000302709.

[2] Stijn de Gouw et al. “OpenJDK’s Java.utils.Collection.sort() Is Broken: The good, the bad and the worst case”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 9206. 2015. DOI: 10.1007/978-3-319-21690-4_16.

[3] Brian Harvey and Mönig Jens. *SNAP! Reference Manual.* 2020. URL: https://snap.berkeley.edu/snap/help/SnapManual.pdf.

[4] Brian Harvey et al. “SNAP! (Build Your Own Blocks) (Abstract Only)”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education.* SIGCSE ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 759. ISBN: 9781450318686. DOI: 10.1145/2445196.2445507. URL: https://doi.org/10.1145/2445196.2445507.

[5] Raquel Hijón-Neira et al. “A guided scratch visual execution environment to introduce programming concepts to cs1 students”. In: *Information (Switzerland)* 12.9 (2021). ISSN: 20782489. DOI: 10.3390/info12090378.

[6] Marieke Huisman and Raúl E Monti. “Teaching Design by Contract using Snap!” In: *2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG).* 2021, pp. 1–5. DOI: 10.1109/SEENG53126.2021.00007. URL: https://ieeexplore.ieee.org/document/9474640.

[7] Rustan Leino. “This is boogie 2”. In: *Manuscript KRML* June (2008).

[8] Bertrand Meyer. “Applying “Design by Contract””. In: *Computer* 25.10 (1992). ISSN: 00189162. DOI: 10.1109/2.161279.

[9] Ibrahim Ouahbi et al. “Learning Basic Programming Concepts by Creating Games with Scratch Programming Environment”. In: *Procedia - Social and Behavioral Sciences* 191 (2015). ISSN: 18770428. DOI: 10.1016/j.sbspro.2015.04.224.

[10] Mitchel Resnick et al. “Scratch: Programming for all”. In: *Communications of the ACM* 52.11 (2009). ISSN: 00010782. DOI: 10.1145/1592761.1592779.

[11] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. “A miss is as good as a mile: Off-by-one errors and arrays in an introductory programming course”. In: *ACE 2020 - Proceedings of the 22nd Australasian Computing Education Conference, Held in conjunction with Australasian Computer Science Week.* 2020. DOI: 10.1145/3373165.3373169.