Demonstrating runtime assertion checking using Snap!

Lars van Arkel
University of Twente, Enschede
The Netherlands
l.vanarkel@student.utwente.nl

ABSTRACT

This paper details the design of an extension to the visual programming language Snap! that aims to implement program verification using runtime assertion checking. In order to design this, we have looked at other program verification tools and how Snap! works. Three designs for implementing the verification in the user interface have been suggested, and a prototype has been made that uses one of the design options. This prototype contains preconditions, postconditions, assertions and a few extra Boolean logic blocks. It has also been tested if the verification techniques do work as expected.

Keywords

Program verification, Runtime assertion checking, Snap!, Visual programming language

1. INTRODUCTION

When writing a software system, it is important that you write programs that are correct. This can be verified by proper testing, but it can also be done by using program verification. There exists tools like this for a few programming languages, like Java or C, but these languages do require a moderate level of programming knowledge.
Visual programming languages do not necessarily have this entry barrier. Visual programming languages are languages that do not require the programmer to write code, but instead allows them to build their programs out of blocks. Examples of this are the languages Scratch, the software used to program LEGO Mindstorms and Snap! [9]. Snap! allows people to build programs out of smaller blocks. An advantage that Snap! has over Scratch is the ability to Build Your Own Blocks. These blocks acts as functions that the user can create themselves. Since Snap! is easy to learn, program verification could be integrated into Snap! as a demonstration of the capabilities and uses of verification.
This paper will focus on how Snap! can be extended for program verification, and how it can be used for runtime verification.

2. BACKGROUND

32nd Twente Student Conference on IT Jan. 31st, 2020, Enschede, The Netherlands.

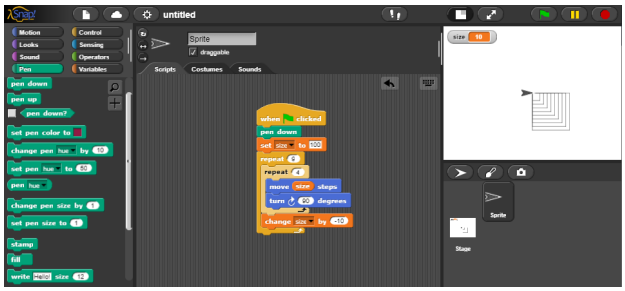

Figure 1. GUI of Snap!

2.1 The language Snap!

Snap! is, as the website states, "a visual, drag-and-drop programming language [9]." If you are already familiar with the language Scratch, Snap! is in fact an extended version of Scratch, built from the ground up. In Snap!, you can build programs from small blocks by dragging them into the code environment and connecting them to other blocks. This can be seen in figure 1. Those blocks can represent functions, values, variables or predicates. Snap! has been made with the purpose of education in mind, making it accessible for new students, as well as containing advanced functions for experienced computer scientists to use. Snap! has programming features such as variables, functions and control flow, but also drawing mechanics using a turtle-like environment [1]. Its main advantage over Scratch is its Build Your Own Block feature, in which you are able to build custom blocks using other blocks. With this feature, you are able to build your own functions that you can use anywhere else in your code. Another feature of Snap! is that lists and functions are considered first-class [10], which means that they can be stored in variables and passed as parameters [9]. The visual aspect of programming with Snap! allows for people who have never programmed before to be able to learn how to program [2].

2.2 Program verification

There are multiple ways to verify the correctness of a program. The most commonly used one is to test your programs, but sometimes this takes quite a lot of time to test only basic features. Another solution is to use program verification with design-by-contract. With program verification, individual functions can be specified with certain rules that can be verified by a program. Design-by-contract has two basic features that this extension will implement; preconditions and postconditions [6]. There are more features used in design by contract, more of them can be found in the section on JML, but in this paper we will only focus on the preconditions and postconditions. As the name implies, preconditions and postconditions tell

something about the state of the program before and after the execution of a specific part of a program. Usually this is limited to a single function. These conditions are Boolean predicates that explain that a program requires a specific state before the function executes, and that it can ensure that the program has a specific state, or a specific range of states. Often verification tools use the terms 'requires' and 'ensures' for the preconditions and postconditions[8]. How to use these conditions in verification can be divided in two methods, static checking and dynamic checking. Static checking checks if a program complies to the rules using mathematical proof techniques. It verifies the program without executing it. Dynamic verification, also called runtime assertion checking, checks the specification of the class at run-time. This means that every time a function is executed, the specification is checked. For the implementation of program verification, we will be focusing on dynamic verification. This is done so that the program can give accurate feedback on what mistake the user has made, so when an assertion is violated, the input of the function that violates the precondition or postcondition can be displayed to the user so that they can check what went wrong. Although static verification can prove for all inputs that a function does not violate the preconditions, according to Maurica et al.[5], static verification is not always capable of verifying all programs, while run-time assertion checking is capable, since it verifies at run-time.

3. REQUIREMENTS OF THE EXTENSION

The extension will need to adhere to the following requirements. These requirements make sure that the extension can be used effectively, and works as a program verification software.

3.1 The look of the user interface

The advantage of Snap! is that it is visually intuitive. The blocks that Snap! uses are distinct in their use. For example, commands have a small indent at the top, and a small part sticking out at the bottom, like a puzzle piece. These two pieces fit together, so it gives a visual representation that these parts can be combined together. The same applies to different types of variables, with Boolean predicates having arrow-shaped borders. Since Snap! is used for education, adding an extension that resembles the UI from Snap! would lower the entry barrier of prior knowledge.

The extension should fit in with these design choices. This means that the preconditions and postconditions should resemble the blocks from Snap! or otherwise resemble the UI of Snap! itself. Any errors thrown, such as a violation of the verification, should also be reported using the error system of Snap!.

3.2 Program verification features

Since the extension is a demo, it does not require all program verification features. However it will require the basic features. The feature requirements of this extension should at least be the validation of preconditions and postconditions. These conditions are the most basic features of program verification, and can demonstrate the effects of specifying a function.

3.3 Execution of program verification

This extension will focus on program verification by runtime assertion checking. Therefore, the previously mentioned program verification features will be checked at runtime. A requirement of this is that when an assertion fails, the user is notified that the assertion fails, the program will terminate, and the user is given information of the state of the program, namely the variables that violate the condition.

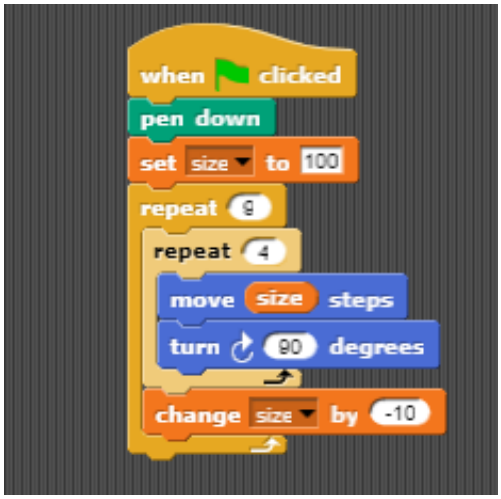

Figure 2. Blocks of Snap!

4. EXISTING PRODUCTS

Although there is not yet a tool for program verification for Snap, there exist program verification tools for other programming languages. In this section we will describe how Snap! creates blocks and executes them. We will look specifically where the blocks are defined, and how they are executed when their block is called. We will also show how program verification tools are used in other programming languages. To do this, we focus on how the user can specify their programs, and how these specifications are checked. The examples we will look at are the Java Modelling Language and the Eiffel programming language.

4.1 How Snap! creates and executes blocks

Snap! is an open source program written as a web application in JavaScript. The code of Snap! can be found on its GitHub page [7]. In the following section, we will describe the basics of how Snap! works, and how modifications to the system could be made. The multiple references to code refer to the source code of Snap! on the GitHub page, with the correct line numbers.

How normal blocks are created

Blocks in Snap! are created and executed in two steps. First a definition of the block is created. This definition can be found in the objects.js file in the function initBlocks, which initialises the blocks field in the SpriteMorph prototype. This field contains an object with all block definitions (objects.js, line 183-1415). These definitions contain the type of the block, which is either a hat, command, reporter or predicate. It also contains the category the block is in, as well as the text on the block. This text includes inputs, so for example the text on the add block, named reportSum, is "%n + %n". The %n represents a numeric input. This input will be shown as a round slot on the block. A complete list of these formats can be found in the documentation of the BlockMorph (blocks.js line 2376). If a block is added in this way, it will show up in the block editor in the correct category. It will however not execute, since the code that executes the block needs to be added. The code that executes the blocks can be added in the threads.js file. In the Process

```
/*@ requires a != null
  @       && (\forall int i;
  @               0 < i && i < a.length;
  @               a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
  // ...
}
```

Figure 3. An example of verification with JML

prototype, the function evaluateBlock executes the block. In this function, a selector is used to call the function. In our previous example of the sum block, the function reportSum is used to evaluate the function (threads.js, line 3316). This function is stored in the Process prototype.

How custom blocks are created and executed

In Snap, the definition of custom blocks are stored in the byob.js file, short for build your own blocks. In this file there are a few important classes. The first one is CustomBlockDefinition. This class contains the data of the custom block, like the category of the block, and the body of the block. This definition is updated every time the apply or OK button is pressed in the block editor. When updating the definition, the contents of the hat are stored in the CustomBlockDefinition.
When a custom block is executed in threads.js, it is done inside the evaluateCustomBlock function in the Process prototype. This function uses the custom block definition to execute the sequence of blocks contained in the custom block.

4.2 Existing program verification tools

Java Modelling language

The Java Modelling Language (JML) is a specification language written for Java. JML itself is not a tool, but rather a notation for Java that can be used by other tools. JML has support for preconditions and postconditions, which can be seen in Figure 3, but also other options, such as invariants and pure flags [8]. Invariants are conditions that hold true for all examples, and if a method is flagged pure, it means that the method does not have any side-effects.
JML code can be compiled with the JML compiler (jmlc), which is an extension to the Java compiler. The code generated from this compiler can be executed with the JML Runtime Assertion Checker (jmlrac) [4]. This checker executes the Java code, and performs a runtime assertion checker of the preconditions and postconditions and the other JML labels.
JML also adds some statements used only by JML. These consists of extra Boolean logic in the form of implies statements, and also some keywords as \return and \old(var), which gives the return value of the function and the value of the variable before the function executed respectively.

Eiffel programming language and design-by-contract

The Eiffel programming language is a language created by Bertrand Meyer in 1986, which aimed to create reliable software systems. One of the techniques it uses for that purpose is design-by-contract. Design-by-contract is part of the language itself. Preconditions and postconditions can be placed at functions by using keywords at the beginning or end of the function. Figure 4 shows how the structure of how to place these keywords in your code. In Eiffel, the precondition is written before the code, while

```
routine_name (argument declarations) is
            -- Header comment
require
            Precondition
do
            Routine body, i.e. instructions
ensure
            Postcondition
end
```

Figure 4. The structure of an Eiffel function with specifications [6]

the postcondition is placed after the code. This differs from JML, where the user can put postconditions next to preconditions above the code. Eiffel's way of placing the contract implies to the user that the code is ran at the end of the program.

5. ARCHITECTURE OF THE EXTENSION

5.1 The user interface

The user interface of the extension is the first thing that users see. Before users are able to execute the features necessary for program verification, they will first need to use the user interface to build the preconditions and postconditions. As stated in the requirements, the UI needs to fit with the blocks of Snap!. Even then, there can be multiple ways of implementing this. When developing the extension, we have found 3 ways of implementing the preconditions and postconditions into Snap!. For these solutions, I have found a few advantages and disadvantages that will be explained in the next sections. All these solutions use the same wording and pattern, which are implemented in different parts of the program.
In the extension, we have named the preconditions requires, and the postconditions ensures. We have done this, since other validation tools use the same naming scheme [8]. Since preconditions and postconditions require the input of a Boolean, the input for the requires and ensures blocks can be limited to allowing just predicates. Snap! supports these predicates with the diamond shaped slots. If a condition is violated, we can throw an error message using Snap!'s standard error handler. If Snap! encounters and exception, the error message will be displayed on the UI. When this happens, the block that is being executed will glow with a red border, and the error message will be displayed in a speech bubble.

5.1.1 Inline conditions

The first solution would be to use inline conditions. These are blocks that can be put anywhere in the program, and are divided into requires and ensures. This would be more in the form of assertions [8]. These assertions can be placed anywhere in the code, but the precondition is executed before a function call, and the postcondition afterwards. This means that every time a custom block is saved, the required and ensures blocks are separated from the program. These blocks would be represented like Figure 5.

Advantages.

The advantage of inline conditions is that the blocks can be placed together to the code. This can be good, but it can also be confusing to the user, since there are no restrictions on putting blocks anywhere, such as an ensures

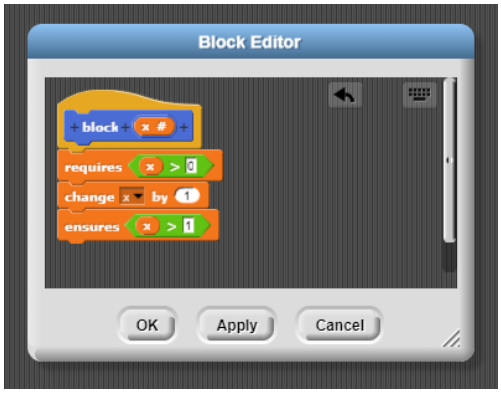

Figure 5. Inline conditions

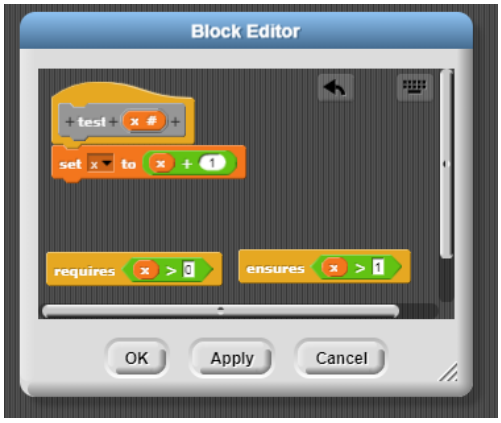

Figure 6. Separate blocks

block after a requires block. Another advantage is that each requires and ensures statement can be put in their own block. There is no need to nest 'and' statements for each requires statement. In this way, if a statement is violated, the error message can trigger on the correct block.

Disadvantages.

The main disadvantage of the inline conditions is that the ability to place the blocks anywhere in the code might be confusing to the user. Since requires should always be executed before the code, and ensures always at the end of the function, users could think that the verification blocks are executed at that location of the code. This is not the case, since otherwise they would just be assert statements. Another difficulty of this implementation would be the execution of the blocks. When executing the custom block, the blocks first need to be separated into verification blocks and blocks that are part of the program. When implementing this into Snap!, extracting the requires and ensures blocks could prove to be hard to process.

5.1.2 *Separate blocks*

The second solution was to have the blocks as separate, non-connectable blocks. These blocks can be placed anywhere in the block editor, but cannot be connected to other blocks. Therefore, they have no connectable puzzle pieces. This should inform users that this block cannot be connected to other block. Since this block does not have rounded edges like a reporter [3], users should distinguish this from a reporter, which returns a value. This block can be selected from the editor, and can thus be placed multiple times in the block editor, allowing for multiple requires blocks, or none at all if there are no specifications for the method. This option would look like Figure 6.

Advantages.

The main advantage of the separate blocks is that, compared to the inline conditions, the blocks are distinct from other blocks that are used to execute code. With separate blocks, the blocks cannot be connected to other blocks. This makes it so the separate blocks are distinct from the main execution path. Users are also free to put the requires and ensures blocks anywhere in the code. Another advantage would be that this solution is easier to implement into the code of Snap! compared to the inline blocks. Since the blocks are not nested somewhere in the program, they can be extracted from the block editor, and put in the correct order.

Disadvantages.

A disadvantage of this solution is that the blocks could not be as intuitive as expected, since Snap! has no blocks without a flat top and bottom. Also, the blocks are only used when making custom blocks, and should therefore only execute in the block editor. If this was used in the script tab, the blocks should not be executed. A solution to this would be to not execute the block in the script tab, like the reporter block, or another way would be to throw an error if the verification blocks are executed outside of the custom block editor.

5.1.3 *Extended hat*

The third solution is to change the hat block in the block editor. This hat block normally contains the information of the custom block, such as the parameters and the text of the block, but also contains the starting puzzle piece of the program. The solution is to add the requires and ensures predicates as part of the hat block itself, with the requires and ensures block put around the code. This is done by a c-shaped ring, which is also used in if-statements and loops to put blocks in the body. To use multiple predicates to be used for requires and ensures, a multi-argument option can be used. This is used by Snap! for lists with any type, but can be also be used for a specific type, such as lists, functions and Booleans, which is the type that we need. Since the code is put in the c-shaped ring and we do not need to execute code after the ensures, there is no need for a puzzle piece after the ensures. This means that we could have a flat bottom, similar to the report block and loop forever. Combining these features can give us something that looks like Figure 7.

Advantages.

The main advantage of the extended hat is that it is harder to confuse the meaning of the solution compared to the other solutions. With the inline conditions, the location of the conditions does not change the execution of the code, but can confuse the user. With the extended hat being a rigid structure, the location of the conditions do not change. The requires block is always above the code, and the ensures block always below. As the hat has a flat bottom, the only location to place blocks is inside the c-shaped ring. This further adds to guiding the user to place their code.
Since we use a multi-argument for the predicates, it is simple to add and remove predicate spots by clicking on the arrows. This means that multiple different predicates do

Figure 7. Extended hat

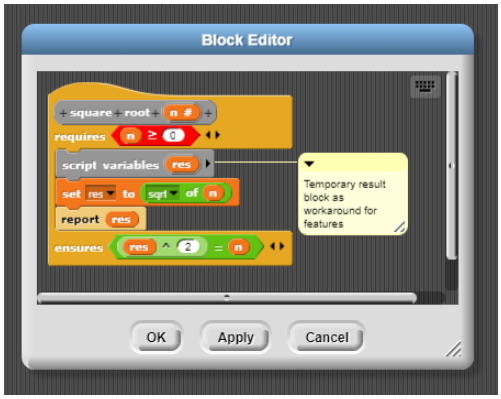

Figure 8. how a custom block is made with the extension

not need to be put inside 'and' statements. This could also mean that the violation of a precondition or a postcondition can be more accurately displayed to the user.

Disadvantages.

A disadvantage of the extended hat is the reduced space for the program. Although the c-shaped ring changes size when more pieces of code are added, the ensures block does take up some space when creating the program. As the preconditions and postconditions are part of the hat, it is also not possible to remove them when making a custom block. Even if there are no preconditions or postconditions to be specified, they do appear on the window. Even changing the amount of predicates does keep the text.

5.1.4 Solution chosen

After analysing the 3 different solutions, we have decided to choose the extended hat. We have quickly eliminated the inline conditions, as we deemed it to be not intuitive for the user that the preconditions are always executed at the start of the code, even if the blocks are not put at the top. Changing this to it being executed at the location would also not be an option, since this would remove the entire precondition and postcondition aspect of the program, instead changing it to assertions.
When deciding between the separate blocks and extended hat, we looked at how the Eiffel programming language handles preconditions and postconditions. In Eiffel, the conditions are placed before and after the code [6], like the extended hat. As with the Eiffel language, the preconditions and postconditions are added to the function specification itself. If the user does not want or need any preconditions or postconditions, the sections for these conditions still exist, but the user does not need to put anything in it.

5.2 Implementation of the extension

To demonstrate the extension, a prototype is made with most features working. The code of this prototype is, like Snap! itself open-source and can be accessed on the GitHub page [11]

Implementation in the Block Editor

To implement the extended hat, the current hat is modified to incorporate the requires and ensures blocks. In this new solution, the body of the block is stored in the C-shaped slot, instead of as a child of the hat block. The require and ensures blocks are added with a MultiArg list. With a MultiArg, the user can select how many predicates they need. Along with that, some changes were also made to store the body of the block in the CustomBlockDefinition. Before, the body of the definition was extracted by finding the children of the custom hat, but now the definition uses the children of the C-shaped slot. The precondition and postcondition are stored in the CustomBlockDefinition as separate fields. When editing an existing block, the conditions and body of the block are put in the right place when reopening the block editor. This implementation can be seen in Figure 8

Execution of verification

The execution of program verification is implemented to work similar to normal execution. When a custom block is evaluated, a new stack frame is created and the contents of the custom block are extracted and put in an array. The extension adds preconditions and postconditions to this array, with the precondition in the front and the postcondition at the back of the array. This works for most blocks, unless if a report block is used to quit the function. In order to solve that problem, the ensures block is also called when reporting from a function. As this does not interfere too much with the regular execution order of the program, the verification is also able to verify recursive functions, and execute the preconditions and the postconditions of each recursive step. If a precondition or postcondition is violated, an error is thrown with the error message system that Snap! already uses, so this works for the requirement of the look and feel of Snap!.

Verification blocks in the Block Editor

The extension also has a few blocks added besides the extended hat. Some of these blocks were added to the code of Snap!, while other blocks were custom made and can be found in the same file as the tests. The blocks added to Snap! are mainly some logic that we felt missing from Snap!, which are the less than or equal and greater than or equal blocks. Snap! did not have 'or equal' blocks before. Other blocks added could be used when writing program verification, such as the implies predicate, or an assert statement, so that preconditions and postconditions are not the only option to verify the program.
Besides blocks added to the code, the testing file also contains some blocks that are used for program verification, but were made in the custom block editor. These blocks could have also been added to the code, but we decided to create them in the custom block editor as it would be easier to make them while creating the test than it was to implement it in Snap!. In a later version, these blocks could also be added to the code.
The blocks that were custom made were a 'for all' block,

which maps a predicate over a list and checks if all values are true, a function that checks if a list is sorted, and a loop invariant. This loop invariant is a modification of the for-loop, and uses the assert statement that is executed at the beginning of the loop and after each iteration of the for loop.

6. TESTING OF THE EXTENSION

As the purpose of the tool is to check if the user has written code that works properly, the tool also needs to function correctly when given code that works and code that does not work. To do this, we have made a test project containing multiple custom blocks that were given preconditions and postconditions. In the examples folder of the GitHub repository [11], the test file can be found. This file also contains the extra custom blocks discussed in the previous chapter. These blocks can be found in the verification tab, while the blocks used for testing can be found in the variables tab, coloured grey.

The tests range from easy variable manipulation to check if the verification works at all, to more complex algorithms such as sorting of lists. Tests with variables are done with the increment and square root blocks, while more complex algorithms such as binary search and 2 sorting algorithms are implemented. Also the custom loop invariant block is tested. Using these tests, some bugs were discovered that we were able to fix. For instance, there were problems when calling recursive functions. The top case of the recursive calls seemed to give a null pointer exception in the code, but we were able to solve this, and further testing also showed that the verification worked in recursion. Another problem that was found with the tests was that the report block made sure the ensures block was called, so a workaround for that was found.

7. CONCLUSIONS

Initially, we set out to find a way to add program verification to a visual programming language. We first looked at other program verification tools, how program verification was implemented and what features existed. We also looked at how Snap! worked, and how other programs have adapted Snap! for other purposes, such as 3D drawing and interacting with another application.

When designing the application, a few requirements were set out. The first requirement was that the implementation should still feel like Snap! We have done this by looking at a few designs of possible ways to add in preconditions and postconditions, while still only using the blocks and features from Snap!. The slots for the precondition and postcondition have the shape of a predicate, so the user knows what kind of block to put in. Even if a condition is violated, we use the standard error handling system of Snap!.

As for the program verification features, we have successfully implemented preconditions and postconditions. When deciding on how we represent this on the user interface, we have chosen to follow the programming language Eiffel and placed the preconditions before the code, and the postconditions after the code. Compared to the other solutions we came up with, we felt this option to be the most easy to follow. Besides the preconditions, we also added some more features such as the assert command and some Boolean logic

The last requirement, the execution of the verification, has also been fulfilled. The execution of the program verification has been implemented and tested. The testing has mainly focused on testing that the verification correctly identifies if a precondition has been violated, and that it does not give an error if no violations occur. These tests range from small programs to more complex sorting algorithms.

We think that this extension is useful in demonstrating how to use program verification. As the language Snap! is not especially useful in professional development, the main use of this extension would be in education. A few ideas will be explained in the following section.

Future work

The prototype could have a few more improvements. Firstly, there are still some bugs with features such as block variables which can be fixed. Also, the custom blocks that were made in the testing demo like the 'for all' and 'sorted' functions could be added to the code. More program verification features like JML's \old and \result keywords could also be added.

As shortly mentioned in the last section, a major application of the extension would be in education. Program verification could be added to a curriculum that already uses some form of visual programming. This could either be incorporated into another curriculum, or could be used when creating a new programming course. The inclusion of new features could also be done by looking at how to teach program verification.

8. REFERENCES

[1] R. Goldman, S. Schaefer, and T. Ju. Turtle geometry in computer graphics and computer-aided design. Computer-Aided Design, 36(14):1471–1482, 2004.

[2] B. Harvey and J. Mönig. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. Proc. Constructionism, pages 1–10, 2010.

[3] B. Harvey and J. Mönig. Snap! reference guide. https://snap.berkeley.edu/SnapManual.pdf.

[4] G. T. Leavens and Y. Cheon. Design by contract with JML, 2006.

[5] F. Maurica, D. R. Cok, and J. Signoles. Runtime assertion checking and static verification: Collaborative partners. In T. Margaria and B. Steffen, editors, Leveraging Applications of Formal Methods, Verification and Validation. Verification, pages 75–91, Cham, 2018. Springer International Publishing.

[6] B. Meyer. Applying design by contract. Computer, 25(10):40–51, 1992.

[7] J. Mönig and B. Harvey. Snap! source code. https://github.com/jmoenig/Snap.

[8] E. Poll. Introduction to JML, tool-supported specification for Java. https://www.cs.ru.nl/E.Poll/talks/jml_basic.pdf.

[9] B. Romagosa. About Snap! https://snap.berkeley.edu/about.

[10] B. Romagosa i Carrasquer. The Snap! Programming System, pages 1–10. Springer International Publishing, Cham, 2019.

[11] L. van Arkel. Verification tool Prototype. https://github.com/LvanArkel/SnapVerified.