

UNIVERSITY OF TWENTE.

Faculty of Science and Technology

Static verification for Snap!

Leon Alberts (s1843796)

4th March 2020

Supervisors

prof. dr. M. Huisman
Dr. R.E. Monti
Dr. Ir. J.F. Broenink

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This research focuses on the implementation of a static verification method in Snap!. Snap! is a visual programming language in which a program can be made by using blocks which will form the program. This language is mainly used by high school students. Therefore an implementation of static verification in this language will be ideal to explain and demonstrate the use of static verification to students at a younger age. In this way students can understand the use of static verification in a language more appealing to them since they already are using it. To implement this static verification, a tool called Boogie [1] will be used. Boogie is an intermediate static verification tool, made by Microsoft research. This tool is meant to be used as a foundation for other static verification tools. The Snap! code will be transformed to Boogie code, which then will be checked by the Boogie verifier. This way, the goal of being able to verify Snap! programs will be achieved.

Contents

1	Introduction	2
2	Background	3
2.1	Snap!	3
2.2	Run time and static verification	3
2.2.1	Conditions	3
2.2.2	Difference between run time and static verification	4
2.2.3	Advantages and Disadvantages	5
2.3	Static verification	6
2.3.1	Cosmetic	7
2.3.2	Design properties	7
2.3.3	Error checking	7
2.3.4	Formal Verification	7
2.4	Hoare Logic	7
2.4.1	Weakest precondition	8
2.5	Automated tools	8
2.6	Boogie	8
3	Implementation	10
3.1	New Snap! Blocks	10
3.2	AST	10
3.3	Snap! to AST	11
3.3.1	Examples	13
3.3.2	Transformation	13
3.3.3	Differences between Snap and Boogie	13
3.4	AST to Boogie	14
3.4.1	Implementing the transformation function	14
3.4.2	Nodes toBoogie()	14
3.4.3	Example	15
4	Examples	17
4.0.1	Example that verifies	17
4.0.2	Example that does not verify	18
5	Conclusion	21
5.1	Further recommendations	21
5.2	Related work	21
	Bibliography	22

Chapter 1

Introduction

Software is more and more present in society nowadays. Moreover, software is used in more important systems than ever. This means that if software in a critique situation would fail, the potential damage it can do is enormous. Fore example if the fail save software in a nuclear reactor would fail, a meltdown can occur due to a wrongly written program. Therefore the validation of software in these systems is important. Verification methods have been made to check these systems. This project will focus on the design of a static verification method for the Snap! programming language.

Software verification can be done mainly in two ways. Static software verification an run time verification. This research is focused on the implementation of static software verification. Static verification can be used without the code being able to run by checking parts of the code with a static verification tool.

Snap! is a graphical programming language mainly used by high school students to learn how to program. This language uses blocks which can be dragged and dropped to create the program. Until now, an implementation of static verification of the Snap! language does not yet exist. With such an implementation, high school students can be shown how verification in software works using a language which they are already familiar with.

To achieve this goal, the snap language needs to be modified. Firstly, there are not any blocks present in the language for verification. Therefore these blocks, such as pre and post condition blocks need to be made. Secondly, there does not exist any blocks for reverencing to an previous instance of an variable, or the result of a function, therefore a *result* and an *old* block need to be made. These blocks are used in reverencing a previous instance of a variable, and the result of a function. This is needed in specifying the verification conditions. The main challenge however, is the translation of the Snap code to a verification language.

For this project, the intermediate verification language Boogie[1] has been chosen. Boogie is an intermediate verification tool, This tool is build by Microsoft research, and is meant for other tools to be build upon. This makes this tool the best choice for this project. To translate the Snap! code to Boogie code, an abstract syntax tree (AST) will be made, which then will be transformed to an Boogie AST using a transform function. This transformed AST will then be translated to Boogie code, which can be verified using the Boogie verifier. In this way it is possible to verify programs made in the Snap! programming language and show the power of static verification to high school students in a way that they can easier understand. This project is done in parallel with an other project which has its main focus on the graphical implementation of the pre and postcondition blocks, and has an implementation of run time verification for the snap program.[2]

Chapter 2

Background

2.1 Snap!

Snap! is a free to use, block based programming language.[3][4] The language mostly aims at high school students to get them interested in programming. Snap is build around blocks, which represent lines of code. By dragging and dropping a block can be placed and one can build a program using these blocks. These blocks are divided in different categories to make the program easier to use. Examples of these categories are: control, containing all the loops if statements and function calls, and variables, containing the blocks to aces and modify variables. These blocks also have different shapes. These shapes correspond with the result a block can give. This way only the right types of blocks can be placed into certain slots. This makes that Snap! does some sort of type checking. Snap! runs using JavaScript and HTML5, making it compatible with almost any web browser.

In Figure 2.1 a simple Snap! program is shown. Every color of a block corresponds to the category the block is in. Yellow blocks for example are control blocks such as If statement, while loops and function calls. This program in particular sets the value of an variable X to 0 and then using a while loop it increases X by one until X has the value of 10. In Snap! almost all variables used can be seen as global variables. Therefore, before this block of code starts, it is possible for X to have a different value than 0 since it can be edited in the past.

Besides Snap! there are other languages which are quite similar to Snap!. One of these languages is Scratch[5]. However, Snap! has quite some extended features that Scratch does not have implemented. Firstly Snap! has a feature called build your own blocks(BYOB) [3]. These blocks can be build using the existing blocks and then can be saved. This means that if a piece of code is used quite often, one can make its own block which can replace this specific part of code. By doing so, one can also create functions. Since Snap! has a call block implemented, a function can be called. This is also an feature which is not present in Scratch. Therefore Snap! is a fully functional programming language in which teenagers can learn how to program like they would in a text based programming language. The advantage however of blocks is that they do not need to learn the syntax of a programming language since almost all the blocks are self explanatory.

2.2 Run time and static verification

2.2.1 Conditions

For the execution of software analyses, conditions to which to behaviour of the code should obey should be specified. The main three of these conditions are pre-conditions, post-conditions and invariants [6] These three conditions together, if they are well specified, could inform the automated tool about what the program should do. However, when they are not fully correctly specified, the code might still behave unexpected in certain cases.

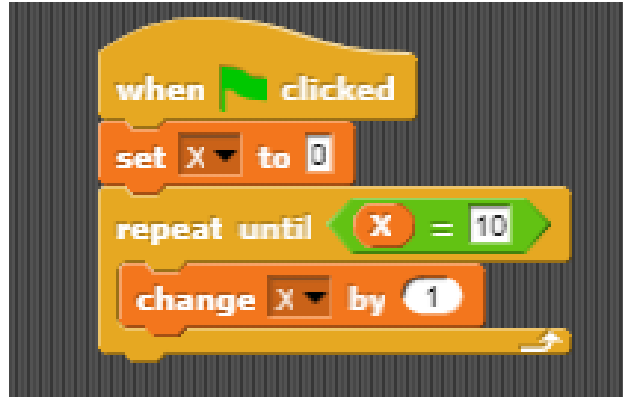


Figure 2.1: An simple snap program:

Preconditions Preconditions can be specified before a function. These conditions tell the tool what should always hold before this function is being called. For example, to make a withdrawal of a bank account, the credit stored on it should be above 0. The automated tool will check all the places that the method is called and verify if all the preconditions are satisfied in all possible cases. If there could exist a way in which the preconditions do not hold, the tool will tell the user in which line this violation may be caused. In this way, the user can fix the error such that the system does not violate the conditions anymore. [6]

Postconditions Postconditions are specified for a function. These conditions tell what should hold after the function is executed. For example, if a function makes a withdrawal of a bank account, the credit stored on the bank account should be lower than the initial value after the function has been executed. When a tool encounters a postcondition it will check the function if there is a input for which this postcondition would not hold. For example, if it is not specified that a negative withdrawal is not allowed, the tool will give an error, since the withdrawal of a negative amount will cause the credit on the account to rise. When a postcondition can be violated the programmer is informed and than can fix the code accordingly. [6]

Invariants An invariant is a statement which should be true during an entire block of code. It is stated at the beginning of the part which is checked, and then it is not allowed to violate the invariant at the boundaries of the code. For example, an invariant could be that the temperature of an simulation never goes lower than zero Kelvin. Then the automated tool will check if there could be any order of functions and inputs in which it could achieve a temperature below the zero Kelvin. If this would be the case, the tool gives an error message and the code could be adjusted. Another way invariants are used is as a loop invariant. In this case the invariant holds for a while or for loop in which the invariant must always be true at the beginning and the end of an iteration. When the loop has been executed, that specific invariant may be violated again. Loop invariants are needed in every loop to describe the behaviour of the loop, this is needed to check the postcondition for the method.

2.2.2 Difference between run time and static verification

In run time verification, as the word implies, the verification is done while the program runs. This is mostly done by checking certain variables at critical points in the code, to check if the values of those variables comply with certain conditions.[7] If this is not the case, the verification method will give an assertion error and terminate the program. In this way, when the verification method stops the program, it is known that the program contains a bug which needs to be fixed. However, if a code only would break by a certain order of events or a specific input value, this would be harder to detect. Since in this case, the program should be ran with all possible different input possibilities.

The code can also be checked using static verification [8]. By static verification, the code would be examined to see how it behaves over the entire set of input values. This way, the code itself is reviewed

```

Function f(x):
    ensures x==0;
    while (x>0):
        x = x - 1;
    return x

```

Figure 2.2: Simple pseudo code

rather than the values of the variables. This can be done either manually, which is time consuming, or by using tools which can do the verification. The disadvantage of these tools is that they are not yet available for all programming languages. For example, if one would have a function which is shown in Figure 2.2. This function would verify with run time verification, as long as the input x is bigger or equal to 0. If the input would be negative, the verifier would conclude that there is a violation of the ensure statement. However, a static verifier can not say that this program is correct since it will check the program for all possible inputs. Only if an precondition is given that x is always non negative it will verify. This example shows the difference between static and run time verification with an simple example. In Table 2.1 below some of the advantages and disadvantages is shown of both run time and static verification. These is be further elaborated on in the next subsection.

	Advantages	Disadvantages
Run time	<ul style="list-style-type: none"> • flexible of what to scan • identifies flaws of the run-time environment 	<ul style="list-style-type: none"> • hard to trace code location • false feeling everything is addressed
Static	<ul style="list-style-type: none"> • Finds exact location of flaws • automated tools scan entire code • finds errors earlier in development 	<ul style="list-style-type: none"> • not find run-time environment flaws • no tools available for every language • More annotation necessary

Table 2.1: Advantages and disadvantages of the two verification methods

2.2.3 Advantages and Disadvantages

In Table 2.1 the advantages and disadvantages of both run time verification and static verification are mentioned. In general neither of the two are better than the other, it depends on the situation which one can be used best. However, due to the limited time for this project, one had to be chosen, and therefore this breakdown in advantages and disadvantages has been made.

Advantages of run time verification

Run time verification is done inside the run time environment. Therefore it takes the run time environment into account when doing the verification. This is due to the fact that run time verification is checking values of variables and variables at a certain moment in time. If in any way the run time environment would make the program do the wrong things, this will be reflected in one of the intermediate results. Then with the right calls to check those variables the error will be caught.

Since the developer can add assertions in the code itself, it is also flexible at what point in the code which variables have to be checked. This makes that only the relevant constants at a certain point have to be checked. However, this may cause the developer to forget to check a part of the program.

Disadvantages of run time verification

When a program gives an assertion error, the developer knows that a pre or post condition is not valid. It is also known which variable causes this to be invalid. However, it is not known which line of code changed the variable in a way it should not have been changed in. This can cause that it takes a lot of time for the developer to find this error.

When a test is done and this test addresses all lines in the entire code, the developer may think that everything has been tested. However, everything has only been tested for one specific set of input values. This means that other input values might still cause the system to behave in an unexpected way. Even when multiple input values are used, one can not be certain that there does not exist any input values in which the system would break.

Advantages of static verification

Since by static analyses the code is analysed, the code does not have to run to be checked. This makes the validation possible before the entire code runs. Therefore it is possible to find errors earlier in development. This makes the bug easier to fix, since there is less code which is dependent on the part which has the bug if it is done earlier in development.

The static verification method is done by checking if the code would run with all possible input values. This can be done in two ways, by a group of professionals which will examine the code closely, or by a tool which will do the examining. Since the code itself is examined, when a bug has been found the exact location is immediately clear. This makes that fixing the error is easy since the place it is created is known.

There exist some automated tools [9] such as Boogie or Dafny which can be used to determine if there are bugs in the code. These tools are faster than professionals checking the code. Some of these tools are open source and free to use.

Disadvantages of static verification

Since the code is checked, the possible flaws of the run time environment is not taken into account, Then if there is a breakdown in the program caused by the run time environment, the static program verification methods will never find it. For this, other forms of verification such as run time verification are required. The automated tools which exist are not yet available for every programming language. This makes that the code has to be parsed into another format to use some of these tools. This requires more work, which can also produce errors. The other option when a tool is not available is to manually check the program code with a group of professionals. However, this is time extensive as well. Since a static verification should be able to verify the code without running it, more annotations are necessary, because it can not simply be checked if a method does not change a certain value but it should be specified before that method. Even though it can seem trivial that that specific method does not change that variable.

As can be read from the advantages and disadvantages section, both methods have their flaws and their advantages. Which method is best depends mostly on the part that needs to be analysed. The best thing to do would be to validate program with both methods. Early in development test the code with static program verification since this is the earlier verification method which is possible when the code does not yet run. When this detects an error in the code it will be easier to repair it since there is less code written which builds on the error and the tool will indicate exactly which line or lines are broken. Later in development, when the program is able to run, it is advised also to use the run time verification. Then validate some specific cases in which the code might act different or can be abused. For this project it has been chosen to implement the static verification. This because this is the most thorough way of testing. Another project which is done in parallel with this project will focus on the run time verification.[2] The next section will in depth explain how static verification works, and what has to be done to provide static verification support for a concrete programming language.

2.3 Static verification

Static verification, or static analysis is, as earlier explained, the verification of a piece of code without executing the code itself. Static verification can be done in many different ways. For example, if someone would read code, and point at some small errors, that person would be doing static verification. If someone would read his own code and correct an error, he would be doing static verification as well. But when a code would be fed to an automated static verification device this would be static verification which would be time efficient. But as can be seen, there exist many methods in which static verification can be performed.

In this part of the report, a few different kinds of static analysis will be explained. Also will be explained how this kind of analyses can be done. These kinds of static analyses are: cosmetic, design properties, error checking and formal proof.

2.3.1 Cosmetic

Cosmetic properties of code are those of the syntax of the code. This verification is mainly done by the IDE environment in which the developer is working. Therefore, a static verification tools main purpose is not to find an error in the syntax of the code. However, when an error is made in a keyword or a semicolon is forgotten at the end of a line, the static verification tool will inform the user in some way that the syntax that was typed is not conform the standard notation of the language written in. This kind of verification is performed by the IDE mainly and does not need an external tool to be checked, but when a mistake has been made in the syntax the tool will detect in nonetheless.

2.3.2 Design properties

Design properties are choices by the developer which influence the behaviour of the code. This can greatly influence the efficiency of the code. For example, when we want to do a multiplication of five and six, and the programmer decides to design a complicated for loop which in the end only adds five times six to an intermediate result, the program would run slower than when the normal multiplication is done. These errors can be found by rereading the code and think about the design choices made and if they are most efficient. When a tool would be used to check the code, that tool will tell that it is correct since it does produce the right output. However, when the code is checked by professionals by hand, they will find this error.

2.3.3 Error checking

The error checking part is the part that could be checked by an external checking method. Error checking is checking if the code could give an unexpected output. This may be due to unconventional naming of the variables, such that it might use a wrong variable, changing a variable which should not be changed, or give an input which is not allowed for a certain method. A tool can be used to check the code or parts of the code for such mistakes. This tool can then tell in which line the mistake was made and the programmer can then adjust his code accordingly.

2.3.4 Formal Verification

Formal verification is the proof if a program always would produce the right output [10]. This is a very hard thing to do since code runs in an unpredictable environment, and an unlimited number of inputs can be given. The way formal verification works is to rewrite the code to a logical mathematical expression, and check that with the mathematical expression that the program should execute. If those two mathematical expressions are logically equivalent, the program should always give the correct output. Formal verification is heavily based on the ideas of Hoare [11] and Dijkstra [12]. These ideas are explained in more detail in the next section.

2.4 Hoare Logic

Hoare triples are sets of conditions and functions that describe the behaviour of a function. These were first described by Hoare in his report in 1969 [11]. These triples form the foundation on which modern day program verification has been build. These triples are written as $\{P\}S\{Q\}$ in which P is the precondition, S is the function and Q is the postcondition. These triples should be read as: if S is executed initially satisfying P , then the result should be satisfying Q . When a situation occurs in which $\{P\}S1\{Q\}$ and $\{Q\}S2\{R\}$ we can combine these two statements into one statement that says that: $\{P\}S1;S2\{R\}$, meaning that if $S1$ is run with input P , and its output Q is used as input for $S2$, we should get output R from $S2$. We can do this since the output of $S1$ is the input of $S2$. These principles are still nowadays used to verify software and are the building blocks of the tools which are used nowadays.

2.4.1 Weakest precondition

The weakest preconditions was an idea by Dijkstra [12]. in the Hoare logic, for a postcondition there exist many preconditions. There exist only one weakest precondition. The notation of the weakest precondition we will use is: $wp(S, Q)$ where S is the function and Q is the postcondition. The weakest preconditions should always satisfy to the Hoare logic so we can say that it should hold that $\{wp(S, Q)\}S\{Q\}$. When specifying a precondition for a function, it is always best to try to apply the weakest precondition for which the program still satisfies the postconditions. This makes it so that the code can run for all possible inputs which do not violate the postcondition. However, finding the weakest precondition can be difficult in cases where functions tend to get more complex.

2.5 Automated tools

For the functional checking in static verification, some automated tools exist, such as JML [13], which is made based on Java. But also tools exists which are called intermediate verification tools. One of these tools is Boogie [14], intermediate verification tools are made in such a way, that it is meant for other tools to be build with Boogie as a foundation. These automated tools will check if the code does what it is expected to do. To check if the code does what it has to do, conditions need to be specified to the tool. These conditions can be expressions about a program state in first order logic. For example, the value of x should not ever increase. Or the guest staying in room one can never be null. The tool then checks the code if there could be any way to violate that statement. If there is it will give an error and tell the user in which line this violation may be caused. This method however, makes the functioning of the tool extremely dependent on the conditions that are given to the tool. If it is not specified that something is not allowed, the code will not check for it. In the next part of this section, the way these conditions work will be explained in more depth.

```
var x;
var y;

function n(x)
requires x > 0;
ensures x > 0;
ensures y==old(y);
{
    return m(x)
}

function m(x)
requires x > 0;
ensures x > 0;
ensures y==old(y);
{
    return x
}
```

Figure 2.3: pseudo-code of conditions

Extensive amount of annotations An automated tool needs quite extensive conditions to be able to check the code. This is since the code can not be executed for static analyses and therefore needs to be more specified. For example if we have the pseudo-code written in Figure 2.3. As can be seen, the function $n(x)$ returns the value of $m(x)$. So when we want to check $n(x)$ with the given pre and post conditions, we need to know the behaviour of $m(x)$. To get this without running the code, the conditions of $m(x)$ need to be specified as well. Even though it seems obvious $m(x)$ would not change the value of y , but the tool can not see that. When it wants to check $n(x)$. To make sure that $n(x)$ satisfies the condition that $y == @old(y)$. We must know if $m(x)$ would change y . And the only way for the tool to know this is by making it an ensure condition.

As can be seen from this example, for a simple check of two simple functions, quite a lot of conditions need to be specified. This causes that static analyses is a lot of work to check the code. When the functions start to get more complex, a lot of pre and post conditions are needed for the tool to be able to check the code. Therefore, by using static verification, it is quite easy to forget to specify an obvious specification which will cause a well running program to fail the verification test.

In the next section, an automated tool called Boogie will be elaborated on since this tool will be used to perform the static verification of the Snap! program.

2.6 Boogie

Boogie is an intermediate static program verifier which has a programming language with build in specification constructs such as requires, ensures and

```

1 //function that increases i until it reaches a given value
2 //return value should be equal to the input value, for any positive integer
3 //for negative input values the program will not work
4
5
6 procedure F(n: int) returns (r: int)
7   requires n >= 0;
8   ensures r == n;
9 {
10  var i :int;
11  i := 0;
12  while (i < n)
13    invariant i <= n;
14  {
15    i := i + 1;
16  }
17  r := i;
18 }

```

Figure 2.4: Boogie code with conditions

invariants. Also the old, result, forall and exist keywords are implemented. It being an intermediate static verifier means that the tool is made to be used as an verifier to build on for other languages. This makes the boogie tool ideal to use in this project. The tool is developed by Microsoft. The source code for Boogie is open source, making it ideal to work with for this project.

As mentioned above, Boogie is an intermediate static program verifier, which nowadays is mainly used as a foundation for other static verification languages such as Dafny and Spec# [1] [15]. Boogie has its own programming language, which is object-based, imperative and sequential, and supports generic classes and dynamic allocation [1]. Boogie has also built-in preconditions, postconditions and invariants. Boogie programs are statically verified for total correctness, which means that the program should both, satisfy the pre and post conditions, and the program should terminate. The Boogie tool is used to generate first-order verification conditions that are passed to a logic reasoning engine called Z3 [1].

To show the syntax of Boogie code, an example will be used, this example can be found in Figure 2.4. As can be seen, the keywords for the conditions are: requires (line 7), ensures (line 8) and invariant (line 13). This code keeps increasing the value of an integer i (line 15), until it reaches the value of the input integer n (line 12). Then it will return the value of i, which then should be equal to the original input value. However, this will not work if n was negative. Therefore a requires statement is needed to specify that negative inputs should not happen (line 7). Also it needs a loop invariant which states that i can never be bigger than n in this loop (line 13). If one of the requires or ensures would have been wrong or forgotten, the Boogie tool will give an error stating that one of the ensures might not hold. It also tells which postcondition can not be ensured.

Chapter 3

Implementation

This chapter is about the implementation of Static verification in Snap!. Here all new additions to snap will be discussed in depth and there will be an explanation on how every implementation works.

3.1 New Snap! Blocks

Before the Assignment, Snap did not have an implementation of pre and post conditions. Therefore for this project these blocks will be added. For now, these blocks will be implemented as normal code blocks, which do not have an implementation in the working code, such that they can be placed without the code being unable to run, but they will be implemented in the translation to boogie. In the future it would be better to embed the ensure and require blocks, as well as the invariant blocks inside of the function declaration block, but implementing this will exceed the scope of this assignment. However, there is an assignment being [2] done in which this will be implemented this way, and combining these two implementations may lead to a better UI for the implementation of the static verification in Snap!.

Since Snap! does not yet have an implementation for the verification, some new blocks have been implemented in order to make Snap! code have an verification implementation. Firstly the pre and postconditions need to be made. For this an ensure and require block have been made. They both have an boolean expression input block. Next to the pre and post conditions an invariant block had to be made. This invariant also has an boolean input block and can be used as an loop invariant or as an invariant for an entire method. It has been chosen to only use one invariant block for these two types of invariants, since the syntax of both invariants are the same in Boogie. Therefore it makes sense to only use one block which can be used in both cases. For the proper use of these blocks, some other new blocks have been made. The *old* block has been made and can be used to access the state of a variable before the function has been called. A *result* block and implies block also have been made to make the creation of statements for the pre and post conditions more extensive.

3.2 AST

As explained in the Snap! section, Snap! is written in HTML and JavaScript. To transform the blocks of Snap! to Boogie code, we need to use the underlying JavaScript code. To achieve this, the abstract syntax tree of the code can be used. Then, the JavaScript code can write the corresponding blocks of JavaScript code as Boogie code in a file. This file then need to be read by the Boogie interpreter. This will then finally return if the code violates any of the conditions.

In Figure 3.1 an example of an Abstract syntax tree is shown. Such a tree can be made from a snap program. For example, this tree has been made from the snap program which can be found in the figure on the top right.

An AST is the representation of the code in its original language. In this case the AST therefore is a representation of the Snap! code. When this code would be transformed to Boogie code, the AST needs to be transformed to an AST which can generate the Boogie code. Therefore a transformation method should

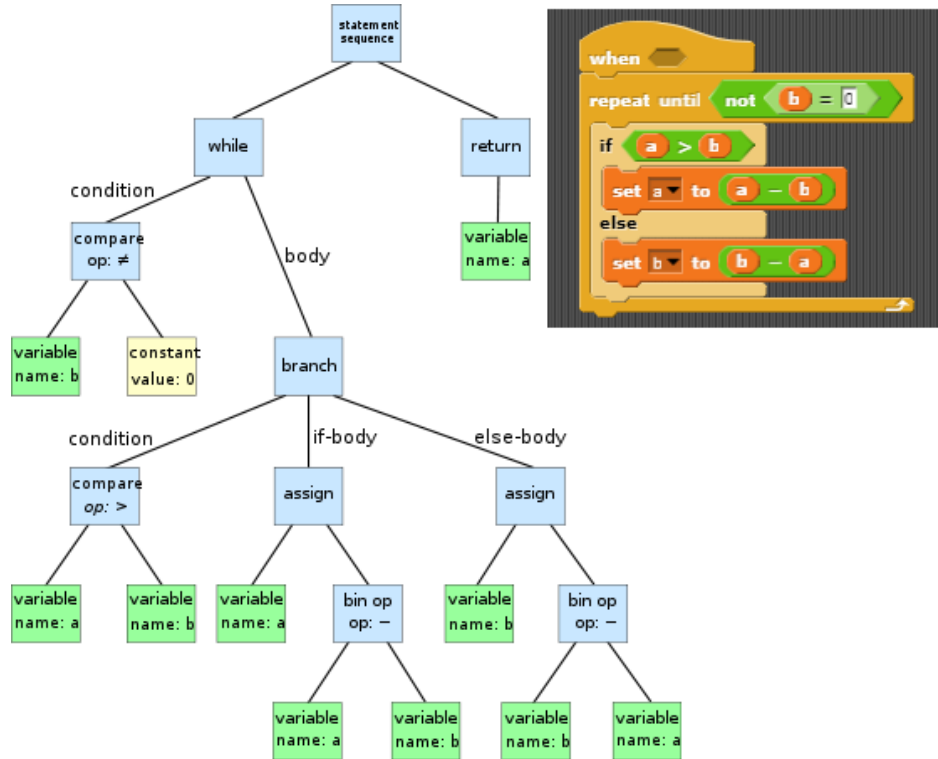


Figure 3.1: An abstract syntax tree for the Euclidean algorithm:

be called on the tree. When the tree is transformed, the tree can be converted back to code, but then it will generate the Boogie code.

In the next sections the implementation of this process will be explained.

3.3 Snap! to AST

First the Snap! blocks are transformed to an AST. The AST is a recursive tree in which all code is represented, and only by looking at the AST, it should be clear what the original Snap! code looked like. For this implementation, different kind of nodes have been made. All these nodes inherited from a parent node, which has all the basic methods that a node needs. The other nodes that are used are : Snap! node, Method node, Verification node, Code node, Loop node, Nested node and Block node. In figure 3.2 the hierarchy of the nodes is shown. The arrow in the image points from a node type to all its possible children types. For example, a method node can have verification nodes and code nodes as children. All nodes shown in the image will be explained below.

Snap! node The Snap! node is the top node of the AST. This node represents the entire Snap! code. All the methods in this code will be children of this Snap! node. This Snap! node also contains a list of all the variables used in all the methods, because all variables used in Snap! methods can be seen as global variables. Therefore they need to be declared on top.

Method node The method node is a child of the Snap! node. This method node contains the hat block of a method. The hat block in Snap! is a block which indicates the start of a method. This method node has two children, a verification node and a code node.

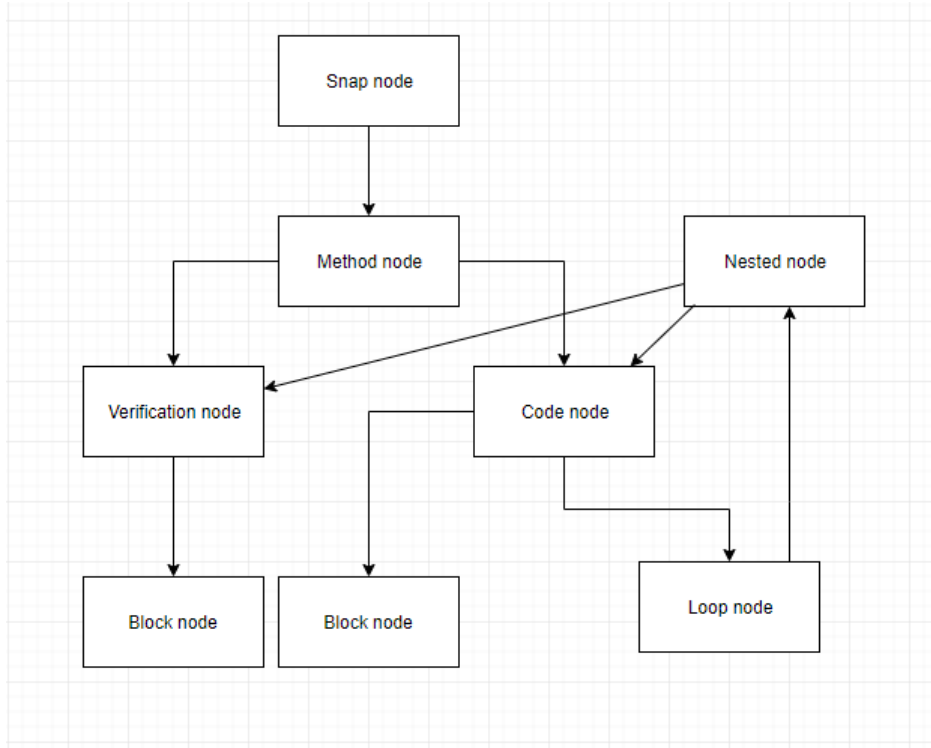


Figure 3.2: The hierarchy of the nodes:

Nested node The nested node will hold a nested block. It is a child of the loop node. The nested node works similar to the method node and will have all blocks in the nested block as children. The only difference is that it does not contain a hat block and thus is not a method on its own. This nested node therefore hold the entire body of the loop node. It has been chosen to be a separate node from the loop node because it functions more as the method node. However, since it is not a method but a nested block it could not be a method block either. Therefore a separate node is introduced.

Verification node The verification node is a child of the method node or the nested node. this verification node will have block nodes as children. But it will only have verification block nodes as children. Therefore the verification node will filter out all the verification Snap! blocks and create block nodes as a child for every verification block in the method or nested block.

Code node The code node is a child of the method node or nested node. This code node will have block nodes as children. But it will only have working code block nodes as children since the verification blocks are filtered out by the verification node.

Block node The block node is a node which is a child of a code node or a verification node. This node will hold a single Snap! block which does not contain a nested block. So it can hold both verification blocks and working code blocks.

Loop node The loop node is a special type of block node, which is made especially for blocks which have one or more nested blocks in them. For example, while loops, for loops and if statements. This loop node has as children, a condition to branch, which is a block node, and the nested block nodes which will hold the nested blocks.

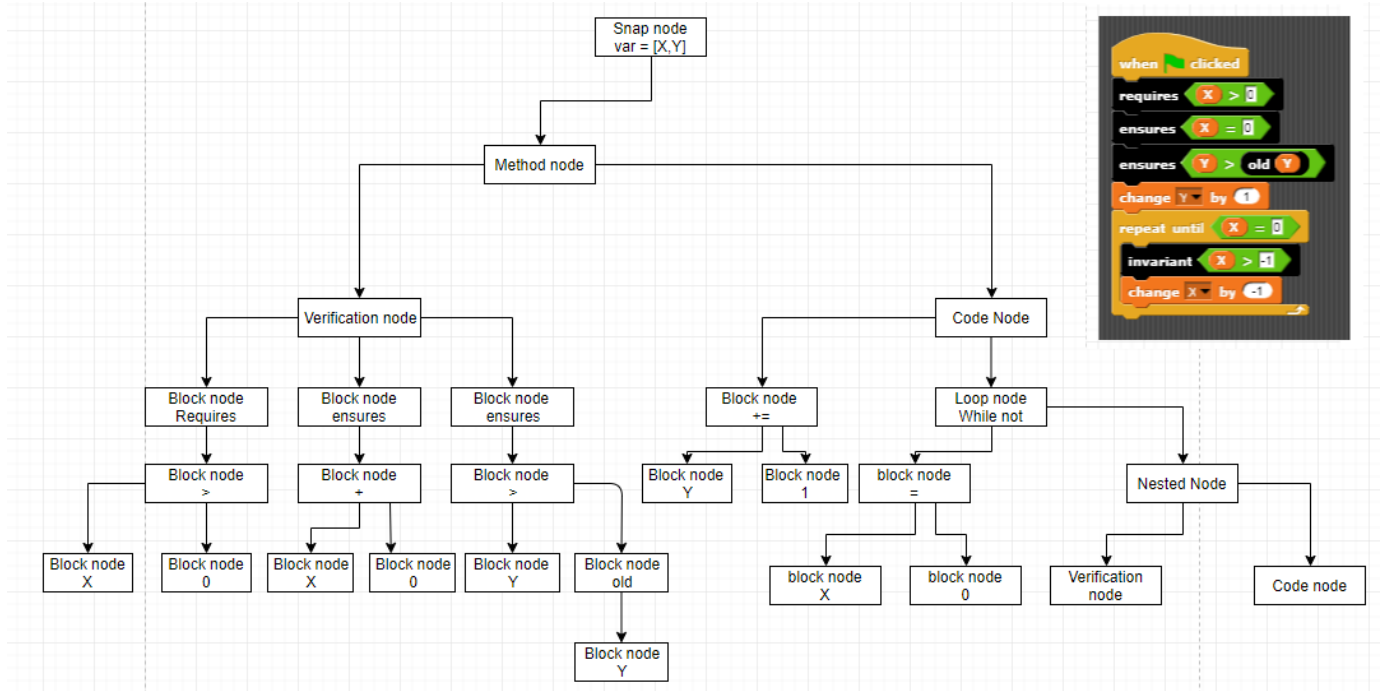


Figure 3.3: An abstract syntax tree for the corresponding snap program:

3.3.1 Examples

In figure 3.3 an example can be found in how a block of Snap! code would transform to a AST. The arrows point from a parent node to its children. This program increases Y always with one, and it will decrease X until it is 0. For X this only works if X initially is non negative, therefore this has been given as a precondition. First the Snap! node is made, which will store all the variables accessed in this program, in this case X and Y . Then a method node is created, this one can store a top block for a method. In this case the top block with the green flag is stored in there. The method block creates a verification node and a code node. The verification node will go through the entire code checking for verification blocks. Then it finds the two ensures and the one requires blocks. These then are created as block nodes as children of the verification node. Then the input of those blocks, for example the $>$ in the requires is a child of the requires block. For the code node, it will have all the non verification blocks as children. So it will find the change Y block, which is stored as a block node as a child of the code node. Then it also finds the while loop, which is stored as a loop node. The blocks inside of the loop node will be children of the nested node. Such an AST can be build from every possible Snap! program, and will be used as a Snap! representation of the program. The next step is to transform the AST in such a way that it is an Boogie AST. In other words, the difference between Boogie and Snap! needs to be found, and a transform function has to be made to transform the Snap! AST to an Boogie AST.

3.3.2 Transformation

In this part the differences between snap and boogie conventions will be explained. This is needed to implement the transform method. After these differences are found, the transform method called on the AST will be explained.

3.3.3 Differences between Snap and Boogie

There are many differences between snap and Boogie. However, most of these differences are not visible in the AST since they are non important for the transformation. Therefore the only points that this part will

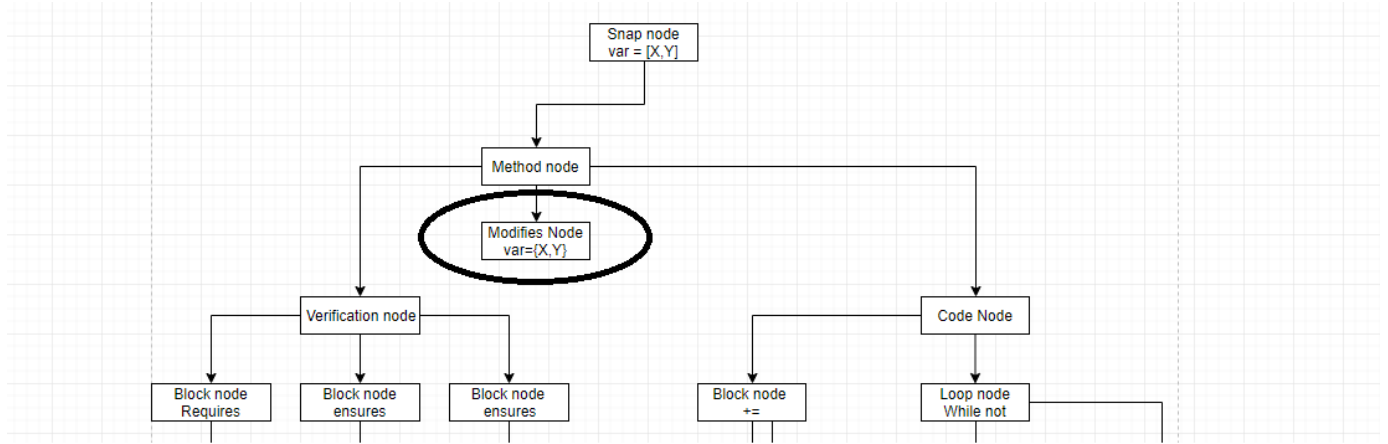


Figure 3.4: An abstract syntax tree transformed to boogie

focus on are the differences which will have an impact on what the AST will look like.

Global variables

In Snap, all variables which are declared can be seen as a global variable. This makes that, to edit a variable, it does not have to be given to the function. Therefore, all variables are stored as a list in the Snap node on top of the AST. This way, we can declare all variables in the start of the Boogie translation, such that Boogie then also will recognise the variables when they are called in a function. When such a variable is edited in a function, Boogie needs to know that that function modifies that variable. For this part the AST does need to be transformed in such a way that all methods will have an extra child, called the modifies child. This child will contain all the variables which this method modifies.

Verification

In Snap, verification was not an option before this project. While Boogie's main purpose is to verify code. Therefore a big difference is in how the verification is implemented. For this project it is chosen that the verification blocks can be placed anywhere in the method, and that the verification node will filter them out and place them on the right place in the tree. Therefore this difference is already implemented in the Snap AST and does not need an transformation function.

3.4 AST to Boogie

3.4.1 Implementing the transformation function

As can be seen in Figure 3.4, the only thing the transformation function needs to do is to detect the variables edited within every method. Then it will add a node to the children of the method node called modifies node. This modifies node will hold all the variables that are edited in the method as its children. This because in Boogie all the variables which are modified by a method should be mentioned in the method itself. when it is transformed to a Boogie AST, the only change of the previously shown AST can be seen in the circle of Figure 3.4.

3.4.2 Nodes toBoogie()

For going from the AST to a Boogie file, the main challenge is to translate the block nodes to the right Boogie translation and having all the code in the right order in the Boogie file. To do this, first translations were made for the blocks. some of these translations are shown in tables 4.1, 4.2. These tables were made

using the Boogie manual [14]. In this section, for each node, it is explained how the translation toBoogie() works. In the nodes, the method which will translate the AST toBoogie() code is called toBoogie().

Snap! node This node has stored all variables and is the first node on which the toBoogie() method will be called. This node will first construct all variables on top of the Boogie file, before calling the toBoogie() method on all the children, which are all the methods.

Method node The methods toBoogie() method will first declare the method. But before the body of the method starts it will call the toBoogie() method on its first two children, since those are the verification node and the modifies node, which both need to be stated before the curly bracket opens. After the first two children it will open the body and then call his third child toBoogie() method, which is the code node. and lastly it will close the bracket.

Verification node The verification nodes toBoogie() method will print the toBoogie() method of all its children, which are the verification block nodes.

Modifies node The verification nodes toBoogie() method will print the toBoogie() method of all its children, which are the variables which are edited in the corresponding method.

Code node The verification nodes toBoogie() method will print the toBoogie() method of all its children, which are the block nodes and eventual loop nodes.

Block node The block nodes toBoogie() method will transform the block to the Boogie syntax, and will call the toBoogie() method on his children if he has any. The translation of some of the blocks can be found in 4.1,4.2.

Loop node The loop nodes toBoogie() method will first print itself, before printing the Boolean to branch on, and then it will call the toBoogie() method on the nested node children. For example, the while loop block will first print the "while", then the first child, which can be " $X > 1$ ". and then it will call the toBoogie() method on the nested node.

Nested node The nested nodes toBoogie() method works almost the same as the Method node toBoogie() method, except that it does not declare the method. But it is responsible for the curly brackets around the nested block and it is responsible for handling both a verification node and a code node, to make sure that loop invariants appear where they should.

3.4.3 Example

When we run the toBoogie() method on the AST from Figure 3.4, we get the Boogie program seen in Figure 3.4.3. When this code is ran, Boogie can verify the code and the result is that this code does not contain any errors. As can be seen in the Figure, all elements of the original Snap! program in Figure 3.3 are translated, for example, the repeat until is translated to the while loop in line 12. and the invariant of that loop can be seen in line 13. Also the variables stored in the Snap! node of the AST are declared before the method runs. These can be seen in line 2 and 3. The modifies node, which was added in the transformation is responsible for the lines 8 and 9. In the next sections more examples will be done, to show that this translation works for Snap! programs, and to show that it also detects when a program does contain an error.

```
1 //this is the text file to verify the snap code
2 var Y : int;
3 var X : int;
4 procedure f()
5   requires (X > 0);
6   ensures (X == 0);
7   ensures (Y > old(Y));
8   modifies Y;
9   modifies X;
10 {
11   Y := Y + 1;
12   while (!(X == 0))
13     invariant (X > -1);
14   {
15     X := X + -1;
16   }
17 }
18
```

Boogie code:

Chapter 4

Examples

4.0.1 Example that verifies

In this section some examples will be shown, these examples will help with further clarifying how the verification of Snap! programs works and how the Boogie verifier responds to those Snap! programs. These examples are also made to show that, as long as the block from Table 4.1 are used, the output of the verifier is correct. In this section two examples will be shown. One example which does verify. This example can be used to show that the verifier indeed does work as expected. Then the second example will not verify, to show that the verifier does indeed show when a program does not satisfy the set conditions.

The first example can be found in Figure 4.1. This program does the following: if the value of X would be above 5, it will get decreased in the repeat until loop until X is 5. If X is between five and zero, X would be decreased to zero in the second repeat until loop. If X would be negative, this program would not terminate. Therefore a requires is given stated that X can not be negative. In the Boogie code the translation of this can be seen in line 4. Then this method ensures that if the original value of $X > 5$, then $X = 5$. This is the second ensures block in the Snap! code, and can be seen in line 6. The first ensure block states that if $!(X < 5)$ then X will be 0. If this example is verified using the Boogie verifier, the program will be verified with zero violations. This is also the expected result since this will hold for all cases where X is non negative, and the precondition of this function states that X is non negative.

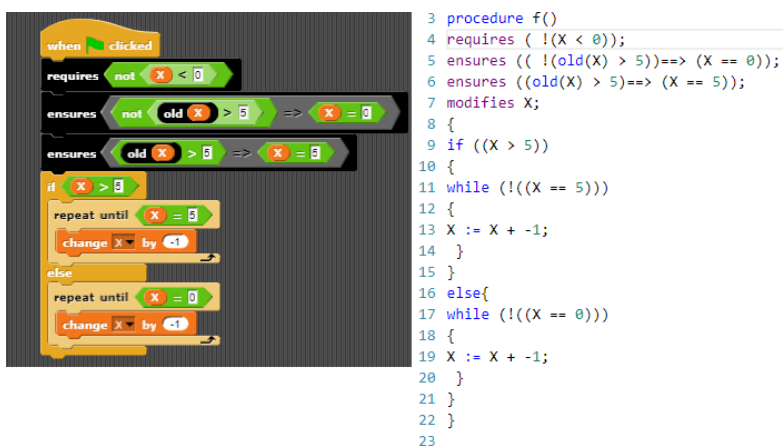


Figure 4.1: Example 1 with Snap! code left and Boogie code right:

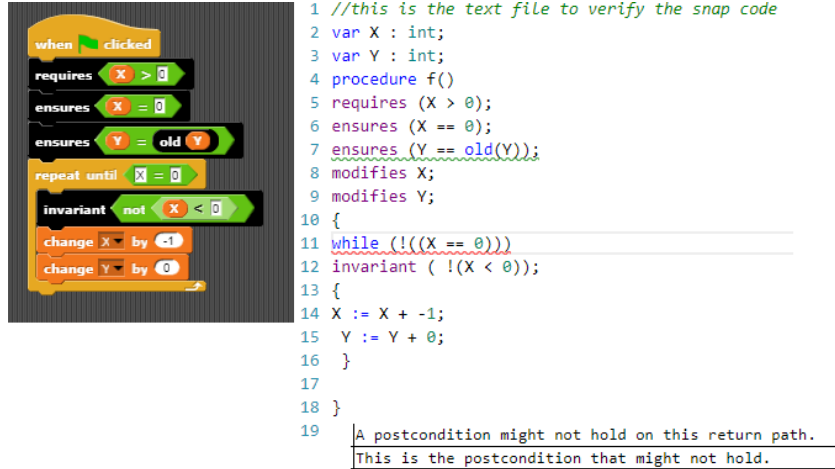


Figure 4.2: Example 2 with Snap! code left and Boogie code right:

4.0.2 Example that does not verify

The second example can be found in Figure 4.2. This program decreases the value of X to 0 and keeps the value of y unchanged. This is also stated in the post conditions. However, this program does not verify. This has to do with the loop invariant. The loop invariant does not state that Y is not changed in the loop. Therefore, the program does not know if $Y = \text{old}(y)$ as stated in line 7 of the Boogie code. As can be seen from the figure, this line is also green underlined meaning that this post condition does not hold. The red line underneath the while loop indicates that this is the return path where this postcondition can be violated. This example also shows that the verification of the snap program works as it is intended.




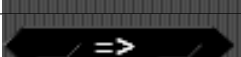






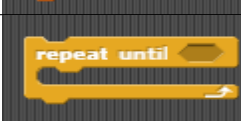
Snap Block	explanation
	<ul style="list-style-type: none"> Is the main block that starts a function
	<ul style="list-style-type: none"> the empty space will hold a Boolean
	<ul style="list-style-type: none"> the empty space will hold a Boolean
	<ul style="list-style-type: none"> the empty spaces will hold a Boolean
	<ul style="list-style-type: none"> the empty space will hold a variable
	<ul style="list-style-type: none"> the empty space will hold a Boolean
	<ul style="list-style-type: none"> the blocks between the if and else statements will be called childblocks. the empty space will hold a Boolean
	<ul style="list-style-type: none"> the blocks between the if and else statements will be called childblocks. the empty space will hold a Boolean
	<ul style="list-style-type: none"> Dependent on the type of x the int can also be a double or float. If x is set to another value, the 0 will be changed into that value
	<ul style="list-style-type: none"> assuming we change var x.
	<ul style="list-style-type: none"> this is the while not loop.

Table 4.1: implemented Snap! blocks












Snap Block	explanation
	<ul style="list-style-type: none"> the empty space will hold a Boolean
	<ul style="list-style-type: none"> the empty spaces will hold a Boolean
	<ul style="list-style-type: none"> the empty spaces will hold a Boolean
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable
	<ul style="list-style-type: none"> the empty spaces will hold a variable

Table 4.2: implemented Snap! Blocks

Chapter 5

Conclusion

This report shows that it is possible to make static verification support for the Snap! programming language. It was achieved by creating new blocks in the Snap! language. These blocks allowed for an implementation which is following the core principles of Snap! of building code with blocks. These blocks allowed for the implementation of static verification since these blocks functioned as pre and post conditions. A intermediate static verification tool called Boogie is used to build the static verification upon. First by translating the Snap! code to an AST, which still represents the Snap! code. Then, using an transform function this AST is transformed to an Boogie AST. From this AST the boogie code can be easily generated by making an method for each node which will print the right syntax to a Boogie file. Then the Boogie verifier can check whether the Snap code satisfies the given pre and post conditions. This way a user can check if the snap code satisfies specified conditions, and Snap! code can be verified. This can be used in demonstrations to high school students about how static verification works and give them some insight in where it is used. The source code of this project can be found in [16]

5.1 Further recommendations

For further work it would be advised to automate the input into Boogie. Now the Snap! code is translated to Boogie code, which than manually needs to be opened by the user in an Boogie verifier. If this process goes automatically, the user would directly know if an program would be correct or not. Also the user interface can be changed by embedding the pre and post conditions into the hat block of a snap method. this way it is more intuitive to use these conditions and they can not be misplaced. The next step would be to translate the error generated by the Boogie verifier to an Snap! error which tells in what block exactly the mistake has been made. If that is done the verification extension does not need to open a separate Boogie program to verify the code but it would look like it is done entirely in Snap, suppressing the Boogie part entirely to the back end.

5.2 Related work

The work in this project is closely related to the work Lars van Arkel did for his bachelor thesis.[2] There the run time verification was implemented in snap, and an intuitive user interface has been made for the pre and post conditions. In the end it would be advised to replace the pre and post condition blocks used in this project by the ones he designed.

Bibliography

- [1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [2] Lars van Arkel. Demonstrating runtime assertion checking using snap! 02 2020.
- [3] Jens Mönig Brian Harvey. Snap manual. <https://snap.berkeley.edu/SnapManual.pdf>, 11 2019.
- [4] Snap! <https://snap.berkeley.edu/snap/snap.html>. Accessed: 04-03-2020.
- [5] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and et al. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009.
- [6] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electronic Notes in Theoretical Computer Science*, 55(2):255 – 276, 2001. RV’2001, Runtime Verification (in connection with CAV ’01).
- [7] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 01 2013.
- [8] What is Static Analysis. <https://blog.ndepend.com/static-analysis-explanation-everyone/>. Accessed: 2019-11-28.
- [9] F. Wedyan, D. Alrmuny, and J. M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *2009 International Conference on Software Testing Verification and Validation*, pages 141–150, April 2009.
- [10] Michael I. Schwartzbach. Lecture notes on static analysis. <https://cs.au.dk/~amoeller/spa/spa.pdf>, 11 2019.
- [11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
- [12] Willem P. de Roever. Dijkstra’s predicate transformer, non-determinism, recursion and termination. In *MFCS*, 1976.
- [13] José Sánchez and Gary T. Leavens. Static verification of ptolemyrely programs using openjml. In *Proceedings of the 13th Workshop on Foundations of Aspect-Oriented Languages*, FOAL ’14, page 13–18, New York, NY, USA, 2014. Association for Computing Machinery.
- [14] K. Rustan M. Leino. This is boogie 2. June 2008.
- [15] Paqui Lucio. A tutorial on using dafny to construct verified software. <https://arxiv.org/pdf/1701.04481.pdf>, 11 2019.
- [16] Leon Alberts. Snap with static verification. <https://github.com/leonaff/StaticSnap>, 2020.