



# Teaching Design by Contract Using Snap!

Marieke Huisman<sup>✉</sup> and Raúl E. Monti<sup>✉</sup>

Formal Methods and Tools, University of Twente, Enschede, The Netherlands  
{m.huisman,r.e.monti}@utwente.nl

**Abstract.** With the progress in deductive program verification research, new tools and techniques have become available to support design-by-contract reasoning about non-trivial programs written in widely-used programming languages. However, deductive program verification remains an activity for experts, with ample experience in programming, specification and verification. We would like to change this situation, by developing program verification techniques that are available to a larger audience. In this paper, we present how we developed program verification support for **Snap!**. **Snap!** is a visual programming language, aiming in particular at high school students. We added specification language constructs in a similar visual style, designed to make the intended semantics clear from the look and feel of the specification constructs. We provide support both for static and dynamic verification of **Snap!** programs. Special attention is given to the error messaging, to also make this as intuitive as possible. Finally, we outline how program verification in **Snap!** could be introduced to high school students in a classroom situation.

**Keywords:** Verification · Software · Education

## 1 Introduction

Research in deductive program verification has made substantial progress over the last years: tools and techniques have been developed to reason about non-trivial programs written in widely-used programming languages, the level of automation has substantially increased, and bugs in widely-used libraries have been found [9, 24, 28]. However, the use of deductive verification techniques remains the field of expert users, and substantial programming knowledge is necessary to appreciate the benefits of these techniques.

We feel that it is important to change this situation, and to make deductive program verification techniques accessible to novice programmers, because specifying the intended behaviour of a program explicitly (including the assumptions that it is making on its environment) is something that programmers should learn about from the beginning, as an integral part of the process leading from design to implementation. Therefore, we feel that it is important that the *Design-by-Contract* approach [23] (DbC), which lies at the core of deductive program verification is taught in first year Computer Science curricula, as is

done already at several institutes, see e.g. CMU’s *Principles of Imperative Computation* freshmen course, which introduces *Design-by-Contract* combined with run-time checking [7, 26]. In this paper, we take this even further, and make the *Design-by-Contract* idea accessible to high school students, in combination with appropriate tool support.

Concretely, this paper presents a *Design-by-Contract* approach for Snap!. Snap! is a visual programming language targeting high school students. The design of Snap! is inspired by Scratch, another widely-used visual programming language. Compared to Scratch, Snap! has some more advanced programming features. In particular, Snap! provides the possibility to create parametrised reusable blocks, basically modelling user-defined functions. Also the look and feel of Snap! aims at high school age, whereas Scratch aims at an even younger age group. Snap! has been successfully integrated in high school curricula, by its integration in the *Beauty and Joy of Computing* course [12]. This course combines programming skills with a training in abstract computational thinking. We feel that the skills taught in the Beauty and Joy of Computing also provide the right background to introduce *Design-by-Contract*.

In an earlier paper [16], we already presented this position and approach to teaching *Design-by-Contract* using Snap!. This paper further motivates and explains our proposal. We also expand on the analysis of our Snap! extension by presenting further alternatives to our block designs. Moreover, we propose a series of lessons and exercises based on our Snap! extension and targeting school curricula.

The first step to support *Design-by-Contract* for Snap! is to define a suitable specification language. The visual specification language that we propose in this paper is built as a seamless extension of Snap!, i.e. we propose a number of new specification blocks and natural modifications of existing ones. These variations capture the main ingredients for the *Design-by-Contract* approach, such as pre- and postconditions. Moreover, we also provide blocks to add assertions in a program, as well as the possibility to specify loop invariants (which are necessary to support static verification of programs with loops). The choice of specification constructs is inspired by existing specification languages for *Design-by-Contract*, such as JML [17], choosing the most frequently used constructs with a clear and intuitive meaning. Moreover, all verification blocks are carefully designed to reflect the intended semantics of the specifications in a visual way. Below, in Sect. 3, we discuss pros and cons of different options for the visualisation of these specification blocks, and motivate our choice. In addition, we have also extended the standard expression pallets of Snap! with some common expressions to ease specifications such as quantifications over lists and implications, and with the specification-only expressions to refer to a return value of a function, and to the value of a variable in the function’s pre-state.

A main concern for a programmer, after writing the specification of the intended behaviour of their programs, should be to validate that these programs behave according to their specification. Therefore, we provide two kinds of tool support for *Design-by-Contract* in Snap!: (i) runtime assertion checking [8], which

checks whether specifications are not violated during a particular program execution, and (ii) static verification (or deductive verification) [19], which verifies that all possible program executions respect its specifications. The runtime assertion checker is built as an extension of the standard Snap! execution mechanism. As mentioned, it only checks for single executions, but has the advantage that it can be used quickly and provides intuitive feedback. The deductive verification support is built by providing a translation from a Snap! program into Boogie [2]. This requires more expertise to use it, and moreover, in the current set-up, the verification is done inside Boogie, and the error messages are not translated back. Improving this process will be a future challenge.

Another important aspect to take into account for a good learning experience are the error messages that indicate that a specification is violated. This is an important challenge, also because clear error messages are still a big challenge for existing *Design-by-Contract* checking tools (both runtime and static). We have integrated these messages in Snap!’s standard error reporting system, again sticking to the look and feel of standard Snap!. Moreover, we have put in effort to make the error messages as clear as possible, so that also a relative novice programmer can understand why the implementation deviates from the specification. Of course, improvements in error reporting in other verification tools can also lead to further improvements in our tool.

Finally, we sketch how the specification language and corresponding tool support could be introduced to high school students. Unfortunately, because of the Covid-19 situation, we have not been able yet to try our plan in a high school setting.

To summarise, this paper contributes the following:

- We propose a visual specification language, seamlessly integrated into Snap!, which can be used to specify the behaviour of Snap! programs, including the behaviour of user-defined blocks (functions).
- We provide tool support to validate whether a Snap! program respects its specification. Our tool support enables both dynamic and static verification, such that high school students immediately get a feeling for the different techniques that exist to validate a specification (and the efforts that are required).
- Verification errors are reported as part of the standard Snap! error messaging system.
- We outline a set of exercises and challenges that can be used to make high school students with basic Snap! knowledge familiar with the *Design-by-Contract* approach and the different existing validation techniques.



The remainder of this paper is organised as follows. Section 2 provides a bit more background information on Snap! and *Design-by-Contract*. Section 3 then discusses the visual specification language, and Sect. 4 the verification result reporting mechanisms. Section 5 discusses the tool support we provide, whereas our training plan for *Design-by-Contract* with Snap! is described in Sect. 6. Finally, Sect. 7 concludes, including related and future work.

## 2 Background

### 2.1 Snap!

**Snap!** is a visual programming language. It has been designed to introduce children (but also adults) to programming in an intuitive way. At the same time, it is also a platform for serious study of computer science<sup>[14]</sup><sup>1</sup>. **Snap!** actually re-implements and extends **Scratch** [27]. Programming in **Snap!** is done by dragging and dropping blocks into the coding area. Blocks represent common program constructs such as variable declarations, control flow statements (branching and loops), function calls and assignments. Snapping blocks together, the user builds a script and visualises its behaviour by means of turtle graphics visualisation, called *sprites*. Sprites can change shape, move, show bubbled text, play music, etc. For all these effects, dedicated blocks are available.

The **Snap!** interface divides the working area into three parts: the pallet area, the scripting area, and the stage area, see Fig. 1<sup>2</sup>. On the left, it shows the various programming blocks. Blocks are organised into pallets that describe their natural use. For instance, the *Motion* pallet contains blocks that allow you to define moves and rotations of your sprites, the *Variables* pallet contains blocks for declaring and manipulating variables, and so on. In **Snap!**, variables are dynamically typed; the main types supported in **Snap!** are Booleans, integers, strings and lists.

Blocks are dragged and dropped from the pallets into the scripting area, located at the centre of the working area. The script in the scripting area defines the behaviour of a *sprite*, i.e. here the **Snap!** program is constructed. Blocks can be arranged by snapping them together, or by inserting them as arguments of other blocks. Blocks can only be used as arguments if their shapes match with the shape of the argument slots in the target block. These shapes actually provide a hint on the expected evaluation type of a block. For example, an operation block corresponding to a summation  shows rounded slots for its summands (for integer values), while an operation block corresponding to a conjunction  shows diamond slots indicating that it expects boolean operands.

The behaviour of the script is shown in the stage area located in the rightmost part of the screen.

In addition, at the bottom of the pallet area, there is a “Make a block” button. This allows the user to define his or her *Build Your Own Block* (BYOB) blocks. When pressed, a new floating “Block Editor” window pops out with a new coding area, in which the behaviour of the personalised block can be defined (similar to how a script is made in the scripting area). Figure 2 shows the definition of a BYOB block that will make the dog sprite jump and woof for each cookie it is fed. Once defined, the BYOB block becomes available to be used just as any other predefined block.

<sup>1</sup> <http://ddi-mod.uni-goettingen.de/ComputerScienceWithSnap.pdf>.

<sup>2</sup> Thanks Flor for the drawings!.

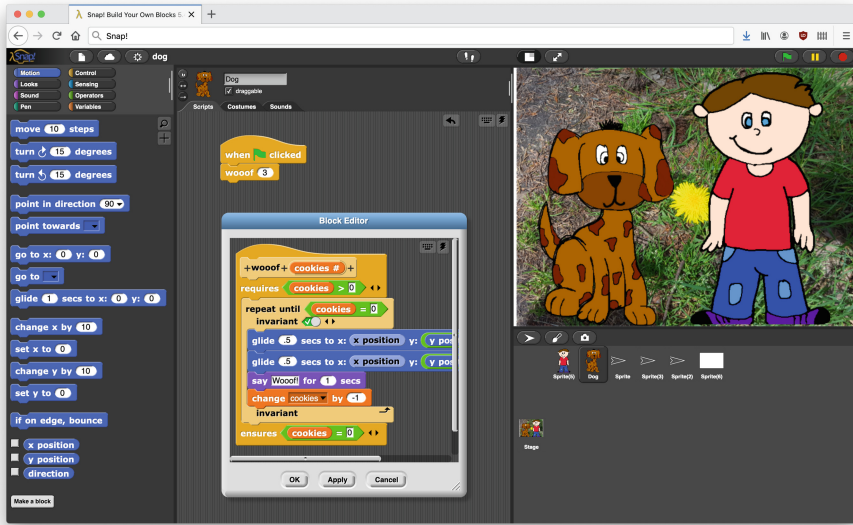


Fig. 1. The Snap! working area.

## 2.2 Program Verification

The basis of the *Design-by-Contract* approach [21] is that the behaviour of all program components is defined as a formally defined contract. For example, at the level of function calls, a function contract specifies the conditions under which a function may be called (the function's *precondition*), and it specifies the guarantees that the function provides to its caller (the function's *postcondition*). There exist several specification languages that have their roots in this *Design-by-Contract* approach. For example the Eiffel programming language has built-in support for pre- and postconditions [21], and for Java, the behavioural interface language JML [18] is widely used. As is common for such languages, we use the keyword *requires* to indicate a precondition, and the keyword *ensures* to indicate a postcondition.

If a program behaviour is specified using contracts, various techniques can be used to validate whether an implementation respects the contract. Here we distinguish in particular between *runtime assertion checking* and *deductive program verification*, which we will refer to as *static verification*.

Runtime assertion checking validates an implementation w.r.t. a specification at runtime. This means that, whenever during program execution a specification is reached, it will be checked for this particular execution that the property specified indeed holds. In particular, this means that whenever a function will be called, its precondition will be checked, and whenever the function returns, its postcondition will be checked. An advantage of this approach is that it is easy and fast to use it: one just runs a program and checks if the execution does

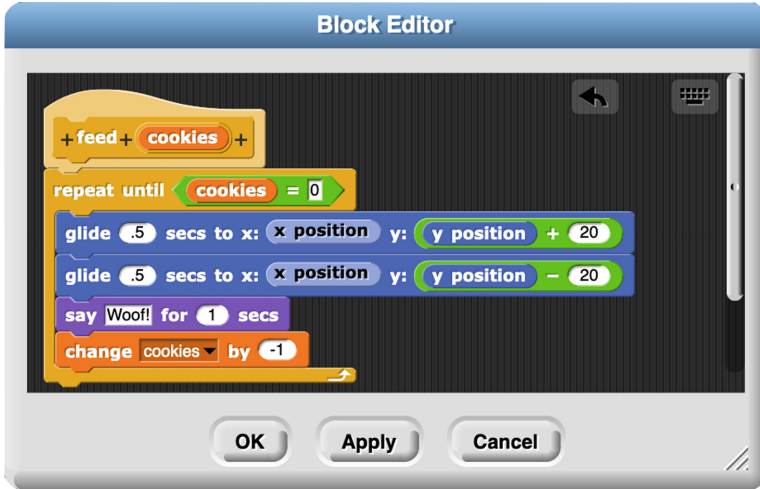


Fig. 2. A BYOB block editor.

not violate the specifications. A disadvantage is that it only provides guarantees about a concrete execution.

In contrast, static verification aims at verifying that all possible behaviours of a function respect its contract. This is done by applying Hoare logic proof rules [15] or using Dijkstra’s predicate transformer semantics [10]. Applying these rules results in a set of first-order proof obligations; if these proof obligations can be proven it means that the code satisfies its specification. Advantage of this approach is that it guarantees correctness of all possible behaviours. Disadvantage is that it is often labour-intensive, and often many additional annotations, such as for example loop invariants, are needed to guide the prover.

### 3 Visual Program Specifications

This section discusses how to add visual specification constructs to **Snap!**. Our goal was to do this in such a way that (1) the intended semantics of the specification construct is clear from the way it is visualised, and (2) that it smoothly integrates with the existing programming constructs in **Snap!**

Often, Design-by-Contract specifications are added as special comments in the code. For example, in JML a function contract is written in a special comment, tagged with an @-symbol, immediately preceding the function declaration. The tag ensures that the comment can be recognised as part of the specification. There also exist languages where for example pre- and postconditions are part of the language (e.g., Eiffel [22], Spec# [3]). We felt that for our goal, specifications should be integrated in a natural way in the language, rather than using comments. Moreover, **Snap!** does not have a comment-block feature. Therefore,

we introduce variations of the existing block structures, in which we added suitable slots for the specifications. This section discusses how we added pre- and postconditions, and in-code specifications such as asserts and loop invariants to Snap!. In addition, to have a sufficiently expressive property specification language, we also propose an extension of the expression constructs. In particular, we provide support for specification-only expressions to represent the result and the old value of an expression, as well as quantified expressions. For all our proposals, we discuss different alternatives, and motivate our choice.

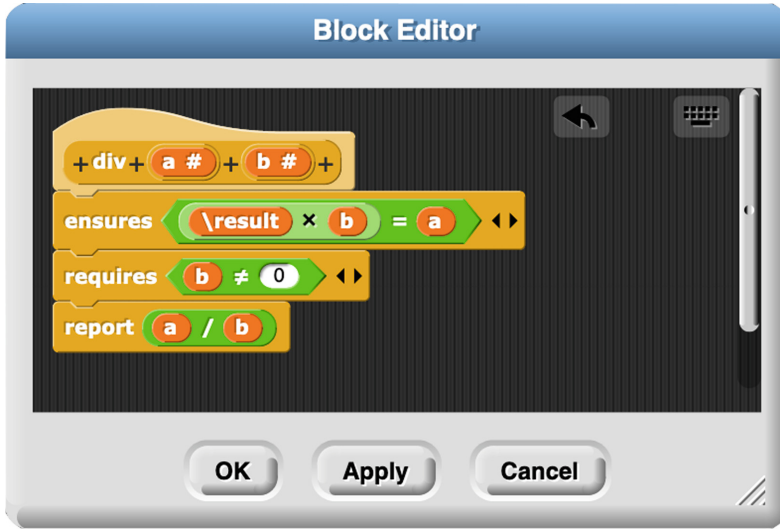
### 3.1 Visual Pre- and Postconditions

To specify pre- and postconditions for a BYOB script, we identified the following alternatives:

1. Individual pre- and postcondition blocks, which can be inserted in the script, just as any other control block (see Fig. 3 for an example). The advantage of this approach is that the user has flexibility in where to use the specifications, and the user explicitly has to learn where to position the pre- and the postconditions. However, the latter could also be considered as a disadvantage, because if the block is put in a strange position, the semantics becomes unclear from a visual point of view. Moreover, it even allows the user to first specify a postcondition, followed by a precondition.
2. Disconnected, dedicated pre- and postconditions blocks, defined on the side (see Fig. 4). This resembles the special comment style as is used by many deductive verification tools. Drawback is that it clutters up the code, and the connection between the BYOB block and the specification block is lost. Moreover, there is no clear visual indication of when the specified properties should hold.
3. A variation of the initial block, with a slot for a precondition at the start of the block, and a slot for a postcondition at the end of the block (Fig. 5). This shape is inspired by the c-shaped style of other Snap! blocks, such as blocks for loops. The main advantage is that it visualises at which points in the execution, the pre- and the postconditions are expected to hold. In addition, it also graphically identifies which code is actually verified. Moreover, the shapes are already familiar to the Snap! programmer. If the slots are not filled, default pre- and postconditions `true` can be used.

Taking into account all advantages and disadvantages of the different alternatives, we decided to implement this last option as part of our DbC-support for Snap!.

Also for the shape of the pre- and postcondition slots, we considered two possible alternatives. A first alternative is to use a single diamond-shaped boolean slot, using the shape of all boolean blocks in Snap!. An arbitrary boolean expression can be built and dragged into this slot by the user. A second alternative is




**Fig. 3.** Separate pre- and postcondition blocks. (Snap! seems to implement mathematical real numbers.)

to add a multiple boolean-argument slot, where we define the semantics of the property to be the conjunction of the evaluation of each of these slots. This is similar to how Snap! extends a list or adds arguments to the header of a BYOB. We opted for the last alternative, because it makes it more visible that we have multiple pre- or postconditions for a block, and it also makes it easier to maintain the specifications.

### 3.2 Visual Assertions and Loop Invariants

For static verification, pre- and postconditions are often not sufficient, and we need additional in-code specifications to guide the prover. These can come in the form of assertions, which specify properties that should hold at a particular point in the program, and loop invariants. While adding assertions for static verification to guide the prover might be a challenge for high school students, they can also be convenient for runtime assertion checking to make it explicit that a property holds at a particular point in the program. As such, intermediate asserts have an intuitive meaning, and can help to “debug” specifications – therefore, we have decided to support them in our prototype.

*Visual Assertions.* To specify assertions, both the property specified and the location within the code are relevant. To allow the specification of assertions at arbitrary places in a script, we define a special assertion block  similar to all other control blocks. The body of the assertion block consists of a multiple boolean-argument slot, similar to how we did this for pre- and postconditions.



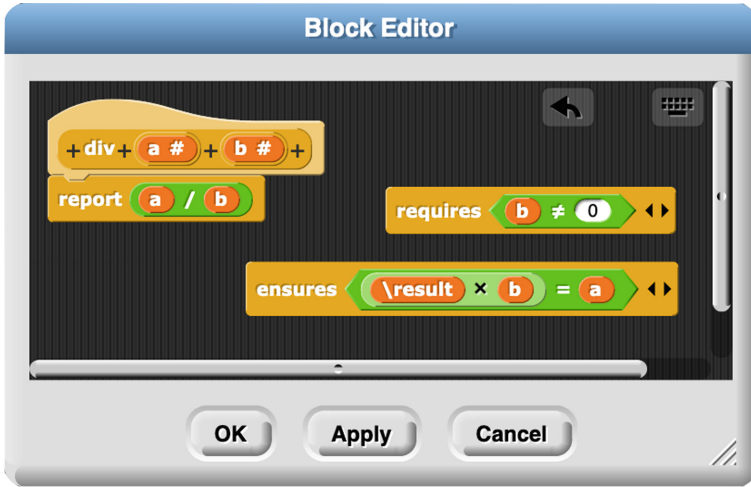


Fig. 4. Detached contracts

*Visual Loop Invariants.* Loop invariants are necessary for static verification of programs with loops [29]. A loop invariant should hold at the beginning and end of every loop iteration. Typically, in textual Design-by-Contract languages, loop invariants are specified just above the loop declaration.

We considered several options for specifying loop invariants in Snap!. One option is to require that the loop invariant is the first instruction of the loop body. However, this does not visually indicate that the invariant also has to hold at the end of every iteration (including the last one). Therefore, we opted for another variant, where we have a (multi-boolean argument) slot to specify the loop invariant in the c-shaped loop block.

This slot is located just after the header where the loop conditions are defined. In addition, the c-shaped loop block repeats the word invariant at the bottom of the block (see Fig. 6) to visually indicate that the invariant is checked after each iteration. We also considered to use an arrow for this purpose, emerging from the invariant declaration and moving along the c-shape up to the bottom. A final option we considered was to implement a ghost placeholder at the end of the c-shaped block, which would be automatically filled with the loop invariant declared at the top. However, we did not further explore this option because we feared it could create confusion about where to enter the loop invariant and it could also use a lot of space with redundant code if not carefully implemented.

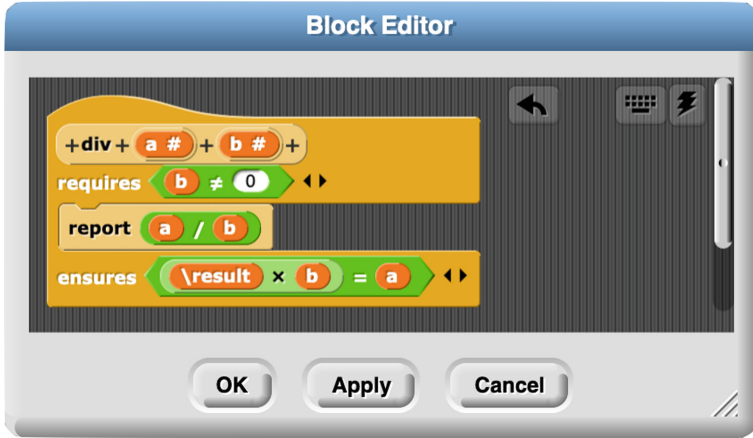


Fig. 5. Extended block with contracts







Fig. 6. Visual loop invariants.

### 3.3 Visual Expressions

As mentioned above, properties are expressed using Snap!’s visual expression language, extended with several specification-specific constructs. Therefore we have introduced some specification-only keywords, as commonly found in Design-by-Contract languages.

- An *old* expression is used in postconditions to indicate that a variable/expression should be evaluated in the pre-state of the function, for example to specify a relation between the input and output state of a function. To support this, we introduced an operator block **\old** with a slot for a variable name.
- A *result* expression refers to the return value of a function inside its postcondition, to specify a property about the result value of a reporter BYOB. This is supported by the introduction of a constant **\result** operator.

In addition, we also introduced syntax to ease the definition of complex Boolean expressions, adding the operator blocks , ,  and , as well as syntax to write more advanced Boolean expressions, introducing support for quantified expressions.

In Design-by-Contract languages, universally quantified expressions are typically written in a format similar to:  $\forall x \in \text{Domain} : \text{filter}(x) : \text{assertion}(x)$ . For example, to specify that an array *ar* is sorted, one would write something like  $\forall i \in \text{Int} : 0 < i \wedge i < |\text{ar}| : \text{ar}[i - 1] \leq \text{ar}[i]$ . We find this notation very general and not always suitable for runtime assertion checking: for instance, in many language, it is allowed to leave the *filter* expression empty, i.e. quantify over an unbounded range. However, experience shows that this is hardly ever needed, and requiring an explicit range avoids many mistakes. Therefore, our quantified expressions in Snap! require the user to explicitly specify the range (See Fig. 7).

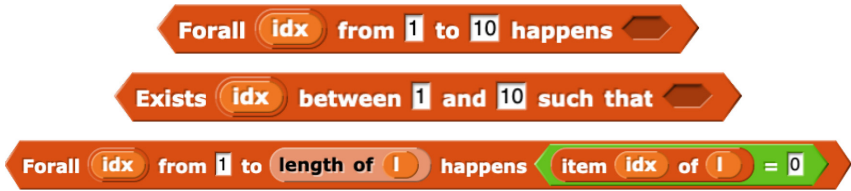


Fig. 7. Global and Existential quantifiers templates, and an example application.

### 3.4 Discussion

Based on feedback from one of the reviewers of this paper, we realised that we might have to deviate more from the standard terminology as is used in the *Design-by-Contract* community, as this might be confusing for high school students, without any former experiences in this area.

Some remarks that were made are the following:

- Consider renaming **ensures** to something more intuitive, such as **promises** or **checks**.
- The loop block with invariant could be misinterpreted: students might think that the loop should stop repeating if the invariant does not hold anymore.
- It might be unclear to what state **old** is referring, and it might be worth to investigate if it is possible to make this more intuitive, for example by using a naming convention to denote earlier program states, as in Kaisar [5].
- Rename quantifier blocks such as **Forall .. from .. to .. happens ..** to something like **All .. from .. to .. satisfy ..**

As future work, we plan to do some experiments with high school students to understand what is the best and most intuitive terminology for them.

## 4 Graphical Approach to Verification Result Reporting

Another important point to consider is how to report on the outcome of the verification. We have to consider two aspects: (1) presenting the verdict of a passed verification, and (2) in case of failure, giving a concrete and understandable explanation for the failure. The latter is especially important in our case, as we are using the technique with unexperienced users. **Snap!** provides several possibilities to present script output to the user, and we discuss how these can be used to present the verification outcomes.

1. Print the value of a variable in the stage using special output blocks. This has the advantage that the stage is the natural place to look at to see the outcome of the script. A drawback is that you loose the connection with the script, to indicate where the verification failed, and that it deviates from the intended use of the stage. Therefore, we decided not to choose this option.
2. Use sounds, using special sound blocks. This is a quick way to indicate that there is verdict, but does not provide any indication of the reason of verification failure. However, it might be an interesting option to explore for vision-impaired users.
3. Use block glowing: when a script is run, the script glows with a green border, when the script fails to execute due to some error the block glows with a red border. This glowing can be reduced to a single block, that caused the failure.
4. Have speech bubbles emerge at specific points in the script that describe the cause of failure. This has the advantage that the failing block can easily be singled out by the location of the bubble, while the cause of failure is described by the text inside the bubble.
5. Use pop-up notification windows. These windows are used by **Snap!** to show help information about blocks for instance. They are also used as confirmation windows for removing BYOB blocks. These windows have the advantage that a failing block can be printed inside them even when the failing script is not currently visible to the user.

We opted for a combination of alternatives:

1. In order to alert about a contract violation, or any assertion invalidated during runtime assertion checking, we opted for option 5. This allows to be very precise about the error, even when the BYOB body is not currently visible to the user (See Fig. 8). A possible extension of this solution would be that clicking **ok** would immediately lead the user to the corresponding code block.
2. In order to alert of errors while compiling to **Boogie**, such as making use of dynamic typing or nested lists in your **Snap!** BYOB code, we opted for option 4. We find this option less invasive than a pop-up window but still as precise, and we can be sure that the blocks involved will be visible since static verification is triggered from the BYOB editor window (See Fig. 9). Notice that the identification of the results of static verification is not something we do from within **Snap!**, since our extension only returns a compiled **Boogie** code which has to be verified with **Boogie** separately.

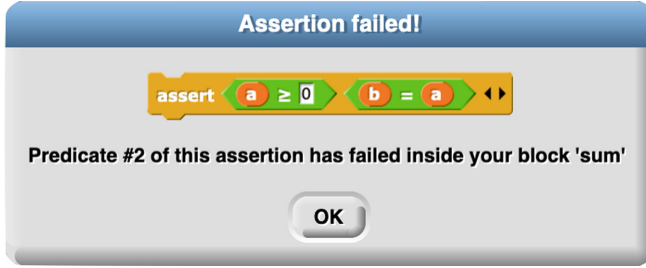


Fig. 8. Failure notification for runtime assertion checking.

Although not currently implemented, we think that verification options 2 and 3 would be interesting alternatives for reporting successful verification.

## 5 Tool Support

We have developed our ideas into a prototypal extension to Snap! which can be found at <https://gitlab.utwente.nl/m7666839/verifiedsnap/>. A couple of running examples for verification, including the solutions to the exercise sheet from Sect. 6, have been added to the *lessons* folder under the root directory. The extension uses the same technology as the original Snap!. After downloading the project to a computer, one can just run the “snap.html” file found at the root directory by using most common web-browsers that support JavaScript.

Our extension supports both runtime assertion checking and static verification of BYOB blocks. Runtime assertion checking is automatically triggered when executing BYOB blocks in the usual way. For static verification, a dedicated button located at the top right corner of the BYOB editor window allows to trigger the compilation of the BYOB code into an intended equivalent Boogie code. The compiled code can be then downloaded to verify it with Boogie. Boogie can either be downloaded<sup>3</sup> and run locally or can be run on the cloud at <https://rise4fun.com/Boogie/>.

The newly introduced verification blocks were naturally distributed along the existing block pallets. These blocks can be used for both types of verification and consist of an **implication** block, **less-or-equal**, **greater-or-equal**, and **different than** blocks, an **assertion** block, a **result** and an **old** block, two types of **looping blocks with invariants**, and **global** and **existential quantification** blocks.

*Runtime assertion checking* has been fully integrated into the normal execution flow of a Snap! program, and thus there are no real restrictions on the BYOB that can be dynamically verified. When any of the newly defined blocks is reached, the block is loaded into the stack of the executing process and is evaluated just as

<sup>3</sup> <https://github.com/boogie-org/boogie>.

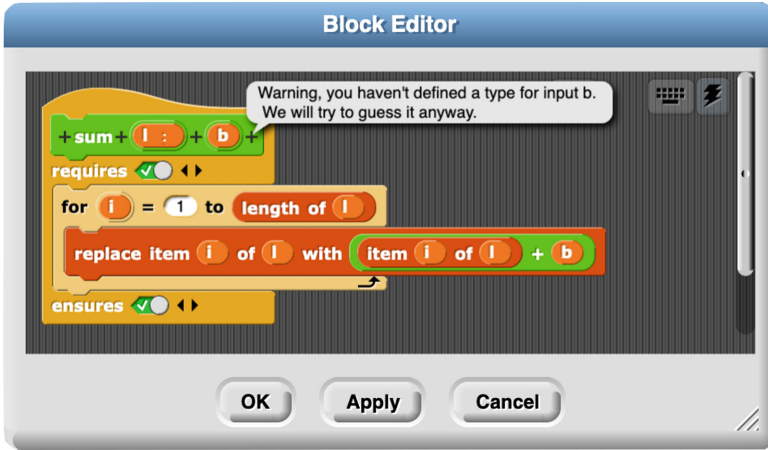


Fig. 9. Static verification compilation notification.

any other **Snap!** block. To make this evaluation possible, a considerable amount of code was introduced to define each block's behaviour. In some cases, such as the implication or the assertion block, the implementation simply consists on a few lines addition to the original scripts of the **Snap!** project. For other blocks, such as the verifiable loops, the implementation effort was considerably bigger, since it involved implementing the execution logic and evaluation flow of the body of these blocks. Nevertheless, these cases only involved naturally introducing new code in the existing code base. The implementation of the **result** and **old** block, and the implementation of the pre- and postcondition evaluations, turned to be more involved and required modifying the evaluation process of the **BYOB** blocks. For instance the evaluation of **report** blocks, i.e., blocks that return a value to the programmer of the calling block, was modified in order to, on one hand catch the reported value of the block for the purpose of evaluating **result** blocks in the postconditions, and on the other hand to make sure that the postcondition is evaluated at every possible exit point of a **BYOB** block.

*Static Verification.* As we decided to develop only prototypal support, we restrict the kind of **BYOB** blocks that can be verified with **Boogie**. We have restricted data types to be integers, booleans and list of integers. In **Snap!** there is no such concept as integers. The restriction is introduced on the verification level by interpreting all **Snap!** real numbers as integers. Furthermore, we do not support dynamic typing of variables in the sense that we statically check that the inferred type of a variable does not change during the execution of a program. Finally, we do not target to compile every available **Snap!** block into **Boogie** and focus only on an interesting subset for the sake of teaching *Design-by-Contract*. The encoding of the blocks into **Boogie** is fully implemented into the new *verification.js* module at the *src* directory of the **Snap!** extension. A considerable part of the encoding is straightforward since a direct counterpart for the construct

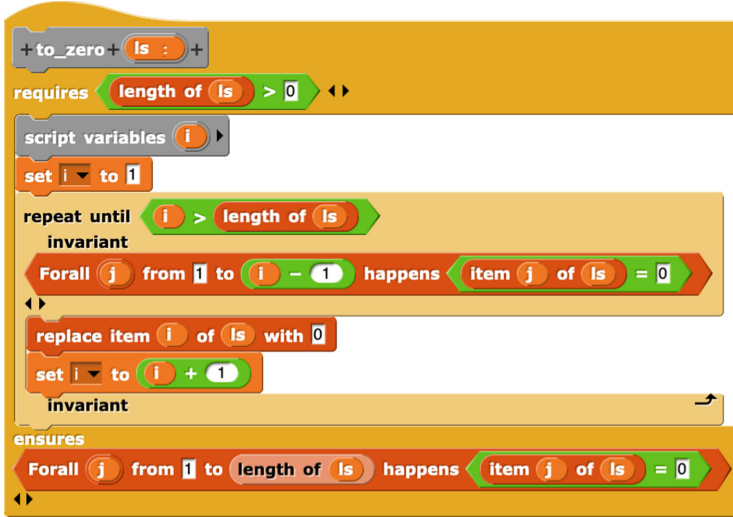


Fig. 10. A BYOB block that sets all elements of a list to zero.

can be found in the Boogie language. Nevertheless, some special attention was needed in the following situations:

1. There is no built-in support for lists, nor sequences in Boogie. Thus, we encode lists of integers as maps from integers to integers, and their length as a separate integer variable. Some operations on lists have a complex encoding. For instance, inserting an element in a list is encoded as shifting the tail of the list towards the end using a loop in order to make place for the new element. Lists initialisation is encoded as successive assignments to the map.
2. For ranges, such as those used for global and existential quantification, we also use maps. In this case, if the range bounds are statically known then the initialisation is encoded as successive assignments to the map. If the bounds are not known, then we use a loop and appropriate loop invariants for the initialisation.
3. Parameters in Boogie's procedures are immutable. This is not the case in Snap! where for instance the content of a list can be modified within a BYOB block. This required to encode the inputs parameters of BYOBs as global variables, which Boogie allows to modify inside a procedure. To avoid checking whether variables may be modified, we applied this transformation to global variables to all parameters.
4. Local variables in Boogie are declared at the starting point of the function and are statically typed. As a consequence, the compilation from the dynamically typed Snap! language is not direct and even involves a type inference mechanism.

To illustrate, List.1.1 shows our Boogie compiled code for the BYOB block of Fig. 10.

```
// This code has been compiled from a verifiable Snap!
  project.

var ls_length : int;
var ls: [int]int;

procedure to_zero ()
  modifies ls;
  modifies ls_length;
  requires (ls_length >= 0);
  ensures (forall j : int ::
    (1 <= j && j <= ls_length) ==> (ls[j-1] == 0));
{
  var i: int;

  i := 1;
  while (!(i > ls_length))
    invariant (forall j : int ::
      (1 <= j && j <= (i-1)) ==> (ls[j-1] == 0));
  {
    ls[i-1] := 0;
    i := (i+1);
  }
}
```

Listing 1.1. Boogie compiled code from Snap! block 10

## 6 Sketch of Teaching Plan

The current situation due to the COVID-19 pandemic jeopardised our plans to test our approach on teaching program verification in high-schools. We have developed an initial lesson plan, containing a sequence of exercises and learning goals to teach program verification to students while assisted by our Snap! with verification extension. The intention of this section is to serve as initial guidance to build more complete plans for teaching software verification that can be integrated into the computer science curricula at high schools. To further develop this lesson plan, we plan to collaborate with didactical experts, that have experience with high school teaching of computer science. The exercises sheet containing the exercises that we mention for each lesson, and the kick-off Snap! projects for each exercise, can be found inside the *lessons* directory at [30]. Along with each lesson we specify which topics the teachers should introduce, such that the students should be able to carry out the exercises.

We assume that students that take on this plan will already have some experience with Snap! and know their way around the coding area. In fact, we target



students from the last years of high-school which are already following a lecture assisted by Snap!.

## Plan


We divide our teaching plan in lessons, starting from what we expect to be the simplest and moving towards more involved or unintuitive aspects. We describe each lesson by its goals and examples of Snap! verification exercises to assist the learning.

*Lesson 1.* The goal of this first lesson is to understand the concept of runtime assertion, as validating expectations that we have from the program at certain points of its execution.

We propose two sets of exercise. The first one already contains assertions at certain points of the code. The students are asked to modify the inputs to the code in a way that they obtain cases where the assertions hold and cases where they do not.

In the second set of exercises, the assertions are missing from the code, and the students are asked to define them themselves and to validate them with different inputs.

Exercise *I* of the exercises sheet may serve as inspiration for preparing exercises for this lesson.

We recommend that the teacher introduces the concept of *predicate*, presents the assertion block  to the students and compares its behaviour against other blocks which do not stop the execution when the predicate does not hold. It should be clear that not fulfilling the predicate will not be tolerated by the code.



*Lesson 2.* The goal of this lesson is to introduce the concept of contract. The student is expected to learn the difference between pre- and postconditions for a method/function and how do these play the role of specifying the expected behaviour of the block of code.

We also expect from this lesson that the student obtain some initial practice and intuition on how to correctly translate a natural language specification into corresponding pre- and postconditions formulae.

We propose to carry out this lesson in three steps: first introduce preconditions as contracts with the caller which we can rely upon when developing our code. Then we continue with postconditions showing how they can help to figure out if the code behaves as specified. Then we integrate pre- and postconditions as formal specifications of the code behaviour and we stress their importance by examples where not defining them may result in unexpected behaviour.

Exercises *II*, *III* and *IV* of the exercises sheet may serve as inspiration for each of the steps of this lesson.

We recommend that the teacher introduces the new initial look of a BYOB block and explains how we can initially check for several predicates in the

‘requires’ list of the block, while we may use the ‘ensures’ list to check for predicates when leaving the block. The teacher should also introduce the special blocks  and  assisted by examples.

*Lesson 3.* The goal of this lesson is that students realise that runtime assertion is testing, and that this is different from static verification, which offers full guarantees on the contract fulfilment.

We propose to make the students test BYOB examples where the error is not easy to spot by assertion checking. Maybe offering misleading tests that do not discover the mistake is another way of creating a false feeling of correctness. The next step is to use static verification by compiling the same **Snap!** code to **Boogie**. This should show them that their belief is mistaken and trigger them to spot the error. It will also show them the limits of testing and the importance of static verification.

You can use exercise *V* of the exercises sheet for inspiration on the type of exercises to undertake this lesson.

It is important to introduce the students with the differences between *assertion checking* and *static verification*. Most importantly, static verification should not be seen as magic but as logical reasoning with formal guarantees. It is also important to indicate the students where to find the new **Boogie** compilation button, located at the top-right corner of the scripting window and BYOB coding window, and explain that this will translate the **Snap!** code into an equivalent **Boogie** code that they can then verify. It is recommended to introduce **Boogie** and a little bit of its language to show them that their specifications are indeed present and verified in the compiled code. Of course the students will need guidance to interpret **Boogie**’s output and map it to their original **Snap!** code.

*Lesson 4.* In this lesson we will introduce *loop invariants*. The goal is that students learn where and when an invariant is validated during a loop verification, and that they are necessary for static verification to succeed. Another goal is to explain the students how to use *quantifiers* to specify properties about lists. In fact, the use of quantifiers to define loop invariants is specially tricky and will require some practice from the students.

Example exercise *VI* of the exercises sheet may inspire other exercises to accompany this lesson.

This lesson is about an advanced verification topic. Students may need help to understand the results of quantification on empty ranges, thus it is important to introduce this while presenting the quantification blocks. Furthermore we recommend presenting these blocks separately from the exercises, using single block examples. We also recommend to show examples of loop invariants that fail on entering the loop, along with examples that fail after looping some amount of times, and finally examples of edge cases due to, for instance, mistakes on the quantification bounds.

*Lesson 5.* For the last lesson we propose a bigger project involving a game, such as for example *Crisis*, which is a multiplayer strategy game. The goal is to

demonstrate the use of formal verification in the context of a project as well as to motivate the students with something more fun than disconnected exercises. Moreover, the difficulty of the verification tasks can be increased in comparison with the previous lessons, taking advantage on the motivation of eventually playing the game.

We propose to present the game with a careful description of its rules, trying to remove ambiguities that may result in unnecessary difficulties to translate into formal specifications. We also propose to hand the students an incomplete Snap! implementation of the game. The incomplete parts may consist of BYOB blocks which still need to be specified, developed, or fixed, in order to be able to play the game.

Exercise VII of the exercises sheet may serve as inspiration for game-kind projects.

## 7 Conclusions

In this paper we presented a prototypal program verification extension to the Snap! tool. The extension is intended to support the teaching of *Design-by-Contract* in the later years of high schools. For this reason, we paid considerable attention to the didactic aspects of our tool: the looks and feel of the extension should remain familiar to Snap! users, the syntax and structure of the new blocks should give a clear intuition about their semantics, and the error reporting should be precise and expressive. Whether we succeeded will have to be evaluated in practice.

Our extension allows to analyse BYOB blocks both by runtime assertion checking and static verification. Runtime assertion checking is fully integrated into Snap! and there is no limitation on the kind of blocks that can be analysed. Static verification compiles the Snap! code into a Boogie equivalent code and the verification needs to be run outside of Snap!. Moreover, we make some restrictions on the kind of BYOB blocks we can compile, in order to keep the complexity of the prototype low. As future work we would like to lift these restrictions as much as possible by integrating the remaining Snap! blocks into the compilation and by allowing other data types to be used. Also, we would like to integrate the verification into Snap!, translating Boogie messages back to the Snap! world, to help student to interpret them.

We also plan to extend Snap! with a Sequence or Array library. This will allow to teach students to verify codes that may commit an ‘index out of bound’ error. Currently, the implementation of lists in Snap! hides this kind of mistakes and works around the problem by returning a special value whose behaviour is not clearly specified. Nevertheless this behaviour gets around quite well, and the student will usually not notice any mistake.

In the last section of our work we sketch a plan of lessons including goals and exercises to teach *Design-by-Contract* to high-school students with some previous knowledge of Snap!. We were not able to test this plan given the current pandemic. As it is still preliminary, it should be improved with the help

of a didactic expert on high school teaching of computer science. Moreover, a considerable amount of extra teaching material and teacher guidance should be developed to accompany this plan. Finally, in a follow up work, we expect to be able to test the plan in classrooms and analyse if the learning goals are met.

*Related Work.* Computer science curricula that uses blocks programming are widely and freely available [4, 6, 11, 13, 25]. Nevertheless, they don't seem to include any topics around design and verification of code. Also, the words 'test' or 'testing' are rare and, where mentioned, they are not sufficiently motivated. The drawbacks of teaching coding with blocks without paying attention to design nor correctness has already been analysed [1, 20]. We have not found any work on teaching these concepts in schools, nor implementations on block programming that allow to support the teaching of design by contract.

## References

1. Aivaloglou, E., Hermans, F.: How kids code and how we know: an exploratory study on the Scratch repository. In: Proceedings of the 2016 ACM Conference on International Computing Education Research, pp. 53–61 (2016)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). [https://doi.org/10.1007/11804192\\_17](https://doi.org/10.1007/11804192_17)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-30569-9\\_3](https://doi.org/10.1007/978-3-540-30569-9_3)
4. The Beauty and Joy of Computing. An AP CS Principles Course. <https://bjc.edc.org/>, Accessed Feb 2022
5. Bohrer, B., Platzer, A.: Structured proofs for adversarial cyber-physical systems. ACM Trans. Embed. Comput. Syst. **20**(5s), 93:1–93:26 (2021)
6. The Creative Computing Curriculum. <http://creativecomputing.gse.harvard.edu/guide/>, Accessed Feb 2022
7. Cervesato, I., Cortina, T.J., Pfenning, F., Razak, S.: An approach to teaching to write safe and correct imperative programs – even in C (2019). <https://www.cs.cmu.edu/~fp/papers/pic19.pdf>
8. Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language. PhD thesis, Department of Computer Science, Iowa State University, Ames. Technical Report 03–09 (2003)
9. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_16](https://doi.org/10.1007/978-3-319-21690-4_16)
10. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Upper saddle River (1976)
11. Factorovich, P., Sawady, F.: Actividades para aprender a Program. AR: Segundo ciclo de la educación primaria y primero de la secundaria. Miller Ed Buenos Aires (2015)

12. Garcia, D., Harvey, B., Barnes, T.: The beauty and joy of computing. *Inroads* **6**(4), 71–79 (2015)
13. CS First. <https://csfirst.withgoogle.com/s/en/home>. Accessed Feb 2022
14. Harvey, B., Mönig, J.: Snap! reference manual (2017). <http://snap.berkeley.edu/SnapManual.pdf>
15. Hoare, C.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969)
16. Huisman, M., Monti, R.E.: Teaching design by contract using snap! In: 2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG), pp. 1–5. IEEE (2021)
17. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Springer, Boston (1999). [https://doi.org/10.1007/978-1-4615-5229-1\\_12](https://doi.org/10.1007/978-1-4615-5229-1_12)
18. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.* **55**(1–3), 185–208 (2005)
19. Leino, K.R.M.: Towards reliable modular programs. Technical report, California Institute of Technology (1995)
20. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.: Habits of programming in Scratch. In: Rößling, G., Naps, T.L., Spannagel, C. (eds.) *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, 27–29 June 2011*, pp. 168–172. ACM (2011)
21. Meyer, B.: Eiffel: a language and environment for software engineering. *J. Syst. Softw.* **8**(3), 199–246 (1988)
22. Meyer, B.: Eiffel: The Language. Prentice-Hall, Upper Saddle River (1991)
23. Meyer, B.: Applying design by contract. *Computer* **25**(10), 40–51 (1992)
24. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-45190-5\\_14](https://doi.org/10.1007/978-3-030-45190-5_14)
25. An Introduction to Programming. A Pencil Code Teacher’s Manual. <https://manual.pencilcode.net/>, Accessed Feb 2022
26. Pfenning, F., Cortina, T.J., Lovas, W.: Teaching imperative programming with contracts at the freshmen level (2011). <https://www.cs.cmu.edu/~fp/papers/pic11.pdf>
27. Resnick, M., et al.: Scratch: programming for all. *Commun. ACM* **52**(11), 60–67 (2009)
28. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) *NFM 2020. LNCS*, vol. 12229, pp. 170–186. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-55754-6\\_10](https://doi.org/10.1007/978-3-030-55754-6_10)
29. Türk, T.: Local reasoning about while-loops. In: Joshi, R., Margaria, T., Müller, P., Naumann, D., Yang, H. (eds.) *VSTTE 2010. Workshop Proceedings*, pp. 29–39. ETH Zürich (2010)
30. Snap! extension for runtime assertion checking and static verification. <https://gitlab.utwente.nl/m7666839/verifiedsnap/>. Accessed Feb 2022