

Teaching Design by Contract using Snap!

1st Marieke Huisman
Formal Methods and Tools
 University of Twente
 Enschede, The Netherlands
 m.huisman@utwente.nl

2nd Raúl E. Monti
Formal Methods and Tools
 University of Twente
 Enschede, The Netherlands
 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 prototypal 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 make this as intuitive as possible.

Index Terms—verification, software, education

I. INTRODUCTION

Research in deductive program verification has made substantial progress over the last years: tools and technique 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 [1], [2], [3]. 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 believe that it is important to make deductive program verification techniques accessible also to novice programmers. Therefore, we have to teach the *Design-by-Contract* [4] (DbC) approach, which requires the programmer to explicitly specify the assumptions and responsibilities of code in a modular way, in parallel with actually teaching programming, i.e. DbC should be taught as an integral part of the process leading from design to implementation. In this paper, we make the *Design-by-Contract* idea accessible to high school students, in combination with appropriate tool support, which is currently unavailable.

Concretely, this paper presents a *Design-by-Contract* approach for Snap! [5]. 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! targets high school students, 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 [6]. This course combines programming skills with a training in abstract computational thinking.

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 and loop invariants in a program and we extend the standard expression pallets of Snap! with some common expressions to ease specifications. The choice of specification constructs is inspired by existing specification languages for *Design-by-Contract*, such as JML [7], 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.

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: (i) runtime assertion checking [8], which checks whether specifications are not violated during a particular program execution, and (ii) static checking (or deductive verification) [9], 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. The deductive verification support is built by providing a translation from a Snap! program into Boogie [10].

Another important aspect to take into account for a good learning experience are the error messages that indicate that a specification is violated. 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 relatively novice programmer can understand why the implementation deviates from the specification.

II. BACKGROUND

A. 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[11]. Snap! actually re-implements and extends Scratch [12]. 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 visualisations, called *sprites*. Sprites can change shape, move, show bubbled text, play music, etc. For all these effects, dedicated blocks are available.

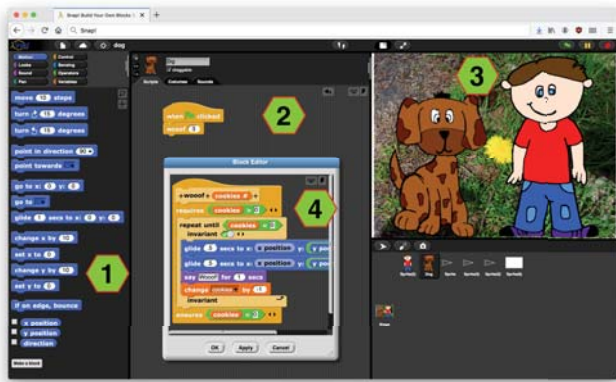

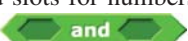


Figure 1: The Snap! working area.

The Snap! interface divides the working area into three parts: the pallet area, the scripting area, and the stage area, indicated by labels 1, 2 and 3, respectively, in Fig. 1. On the left, the various programming blocks are organised into pallets that describe their natural use. For instance, the *Variables* pallet contains blocks for declaring and manipulating variables. In Snap!, variables are dynamically typed. Blocks are dragged and dropped from the pallets into the scripting area, located at the centre of the working area where 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 instance, rounded slots for numbers  and diamond slots for booleans .

The behaviour of the script is shown with turtle graphics drawings 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). Label 4 in Fig. 1 shows a BYOB block being edited. Once defined, the BYOB block becomes available to be used just as any other predefined block.

B. Program Verification

The basis of the *Design-by-Contract* approach [13] is that the behaviour of all program components is defined as a contract. For example, 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 [14], and for Java, the behavioural interface language JML [15] 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.

Dynamic verification 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 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 [16] or using Dijkstra’s predicate transformer semantics [17]. 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.

III. 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 [18], Spec# [19]). We felt that for our goal, specifications should be integrated in a natural way in the language, rather than using comments. 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.

A. Visual Pre- and Postconditions

To specify pre- and postconditions for a BYOB script, we provide a variation of the initial hat 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. 2).

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 postcondition `true` can be used. Notice that the pre- and postcondition slots consist of multiple boolean-argument slots, and we define 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.

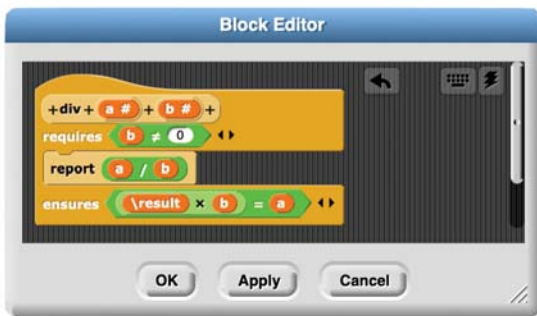


Figure 2: Hat block extended with contracts

B. 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, such as assertions, which specify properties that should hold at a particular point in the program, and loop invariants. Moreover, assertions can also be convenient for runtime assertion checking to make it explicit that a property holds at a particular point in the program.

a) *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 `assert` similar to all other control blocks.

b) *Visual Loop Invariants:* Loop invariants are necessary for static verification [20]. A loop invariant should hold at the beginning and end of every loop iteration. To account for this, we provide a (multi-argument boolean) slot to specify the loop invariant in the traditional Snap! 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 Figure 3) to visually indicate that the invariant is checked after each iteration.

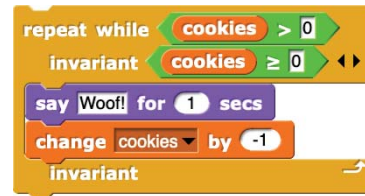


Figure 3: Visual loop invariants.

C. Visual Expressions

In addition, 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. 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. We support this by introducing a constant `\result` operator, that allows to specify a property about the result value of a reporter BYOB.

We also introduce syntax to ease the definition of complex Boolean expressions, by means of the operator blocks `implies`, `≥`, `≤` and `≠`, as well as syntax to write more advanced Boolean expressions, introducing support for quantified expressions (See Fig.4).



Figure 4: A global quantification expression block

IV. GRAPHICAL APPROACH TO VERIFICATION RESULT REPORTING

Another important point to consider is how to report on the outcome of the verification: (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 inexperienced users.

In order to signal a contract violation, or any assertion invalidated during dynamic verification, we use Snap!'s pop-up notification windows. 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. This allows to be very precise about the error, even when the BYOB body is not currently visible.

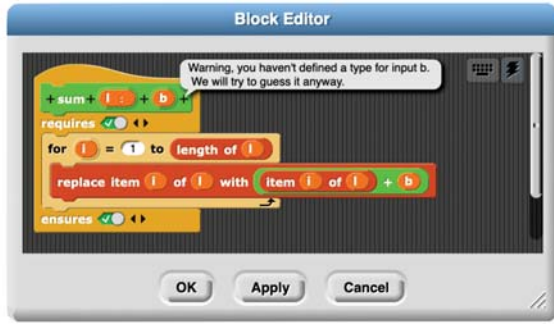


Figure 5: Static verification compilation notification.

In order to signal errors while compiling to Boogie, such as making use of dynamic typing or nested lists in your Snap! BYOB code, we use Snap!’s speech bubbles that can emerge at specific points in the script while describing 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. 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.5). Notice that currently we do not report the results of static verification within Snap!, since our extension only returns a compiled Boogie code which has to be verified with Boogie separately.

V. TOOL SUPPORT

We have developed our ideas into a prototypical extension to Snap! which can be found at <https://git.snt.utwente.nl/montire/verifiedsnap/>. This repository also contains a set of running examples to showcase the new support for verification. These are available in the *lessons* folder under the root directory along with an exercise sheet named *exercises.pdf*. The extension uses the same technology as the original Snap! and can be run by just opening the *snap.html* file in most common web-browsers that support java-script.

Our extension supports both dynamic and static verification of BYOB blocks. Dynamic verification 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 and verified with Boogie. Boogie can be run locally or on the cloud at <https://rise4fun.com/Boogie/>. *Dynamic verification* has been fully integrated into the normal execution flow of a Snap! program, and thus there is no real restrictions on the characteristics of the BYOB that can be dynamically verified. For *Static verification*, we have restricted data types to be Integers, Booleans and List of Integers. Moreover, we do not support dynamic typing of variables. Finally, we only focus on compiling an interesting subset of Snap! blocks for the sake of teaching *Design-by-Contract*.

VI. CONCLUSIONS

This paper presented a prototypical program verification extension to Snap!. The extension is intended to support the teaching of *Design-by-Contract* in the later years of high schools. 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.

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 would like to carry out an empirical study on our proposed approach. This will require the development of a concrete study plan and its evaluation in a Dutch classroom.

Computer science curricula that uses blocks programming is widely and freely available [21], [22], [23], [24], [25]. Nevertheless, it is hardly spotted that they include topics around design and verification of code. The words ‘test’ or ‘testing’ are also rare around the curricula 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 [26], [27]. We have not found any work on teaching these concepts in schools, nor implementations on block programming that support teaching them.



Marieke Huisman is a professor in Software Reliability at the University of Twente. She obtained her PhD in 2001 from the Radboud University Nijmegen. Afterwards she worked at INRIA Sophia Antipolis, and since 2008 at University of Twente. Her research interests are in the verification of concurrent software, as implemented in the VerCors program verifier. She is in particular interested in making verification usable in a practical setting, and she works for example on annotation generation, and support for different programming languages.



Raúl E. Monti received his PhD in 2018 from Universidad Nacional de Córdoba. He is currently a PostDoc at the Universiteit Twente. His research interests involve the development and practical application of formal foundations and tools for analysis and verification of software and hardware systems by means of model checking and deductive verification. His work involves interacting with industry to apply his research in the verification of industrial (embedded) systems and software.

REFERENCES

- [1] S. De Gouw, J. Rot, F. De Boer, R. Bubel, and R. Hähnle, "OpenJDK's java.util.collection.sort() is broken: The good, the bad and the worst case," in *Proc. 27th Intl. Conf. on Computer Aided Verification (CAV)*, San Francisco, ser. LNCS, D. Kroening and C. Pasareanu, Eds., vol. 9206. Springer, Jul. 2015, pp. 273–289.
- [2] W. Oortwijn, M. Huisman, S. J. Joosten, and J. van de Pol, "Automated verification of parallel nested dfs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 247–265.
- [3] M. Safari, W. Oortwijn, S. Joosten, and M. Huisman, "Formal verification of parallel prefix sum," in *NASA Formal Methods*, R. Lee, S. Jha, and A. Mavridou, Eds. Cham: Springer International Publishing, 2020, pp. 170–186. [Online]. Available: https://doi.org/10.1007/978-3-030-55754-6_10
- [4] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992. [Online]. Available: <https://doi.org/10.1109/2.161279>
- [5] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley, "Snap!(build your own blocks)," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 759–759.
- [6] D. Garcia, B. Harvey, and T. Barnes, "The beauty and joy of computing," *Inroads*, vol. 6, no. 4, pp. 71–79, 2015. [Online]. Available: <https://doi.org/10.1145/2835184>
- [7] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston, MA: Springer US, 1999, pp. 175–188.
- [8] Y. Cheon, "A runtime assertion checker for the Java Modeling Language," Ph.D. dissertation, Department of Computer Science, Iowa State University, Ames, 2003, technical Report 03-09.
- [9] K. R. M. Leino, "Towards reliable modular programs," 1995.
- [10] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 364–387. [Online]. Available: https://doi.org/10.1007/11804192_17
- [11] B. Harvey and J. Mönig, "Snap! reference manual," URL <http://snap.berkeley.edu/SnapManual.pdf>, 2017.
- [12] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. S. Silver, B. Silverman, and Y. B. Kafai, "Scratch: programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, 2009. [Online]. Available: <https://doi.org/10.1145/1592761.1592779>
- [13] B. Meyer, J.-M. Nerson, and M. Matsuo, "Eiffel: object-oriented design for software engineering," in *European Software Engineering Conference*. Springer, 1987, pp. 221–229.
- [14] B. Meyer, "Eiffel: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.
- [15] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of jml accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, 2005.
- [16] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [17] E. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
- [18] B. Meyer, *Eiffel: The Language*. Prentice-Hall, 1991. [Online]. Available: <http://www.eiffel.com/doc/#etl>
- [19] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, ser. LNCS, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362. Springer, 2005, pp. 151–171.
- [20] T. Türk, "Local reasoning about while-loops," in *VSTTE 2010. Workshop Proceedings*, R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, Eds. ETH Zürich, 2010, pp. 29–39.
- [21] "The Beauty and Joy of Computing. An AP CS Principles Course," <https://bjc.edc.org/>. Accessed October 2020.
- [22] "CS First," <https://csfirst.withgoogle.com/s/en/home>. Accessed October 2020.
- [23] "The Creative Computing Curriculum," <http://creativecomputing.gse.harvard.edu/guide/>. Accessed October 2020.
- [24] "An Introduction to Programming. A Pencil Code Teacher's Manual," <https://manual.pencilcode.net/>. Accessed October 2020.
- [25] P. Factorovich and F. Sawady, *Actividades para aprender a Program. AR: Segundo ciclo de la educación primaria y primero de la secundaria*. Miller Ed Buenos Aires, 2015.
- [26] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in scratch," in *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011*, G. Röbling, T. L. Naps, and C. Spannagel, Eds. ACM, 2011, pp. 168–172. [Online]. Available: <https://doi.org/10.1145/1999747.1999796>
- [27] E. Aivaloglou and F. Hermans, "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*, 2016, pp. 53–61.