Using Hypersafety Verification for Proving Correctness of Programming Assignments

Jude K Anil
TCS Research
Pune, India
jude.anil@tcs.com

Sumanth Prabhu S
TCS Research
Pune, India
sumanth.prabhu@tcs.com

Kumar Madhukar
TCS Research
Pune, India
kumar.madhukar@tcs.com

R Venkatesh
TCS Research
Pune, India
r.venky@tcs.com

ABSTRACT

We look at the problem of deciding correctness of a programming assignment submitted by a student, with respect to a reference implementation provided by the teacher, the correctness property being output equivalence of the two programs. Typically, programming assignments are evaluated against a set of test-inputs. This checks for output equivalence, but is limited to the cases that have been tested for. One may compose the programs sequentially and assert at the end that the outputs must match. But verifying such programs is not easy; the proofs often require that the functionality of every component program be captured fully, making invariant inference a challenge. One may discover mismatches (i.e., bugs) using a bounded model checker, but their absence brings us back to the question of verification. In this paper, we show how a hypersafety verification technique can effectively be used for verifying correctness of programming assignments. This connects two seemingly unrelated problems, and opens up the possibility of employing tools and techniques being developed for the former to efficiently address the latter. We demonstrate the practicability of this approach by using a hypersafety verification tool named *weaver* and several sample assignment problems.

CCS CONCEPTS

• **Software and its engineering → Formal methods**; **Correctness**; • **Theory of computation → Program reasoning**.

KEYWORDS

Programming Assignments, Verification, Hypersafety

ACM Reference Format:
Jude K Anil, Sumanth Prabhu S, Kumar Madhukar, and R Venkatesh. 2020. Using Hypersafety Verification for Proving Correctness of Programming Assignments. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381747

1 INTRODUCTION

In a celebrated paper about half a century back [14], C. A. R. Hoare remarked that computer programming is an exact science – in that all the properties of a program, and the consequences of executing it in any given environment can, in principle, be found from the text of the program itself by means of purely deductive reasoning. One of the most important properties of a program is whether or not it carries out its intended function. If it indeed does, a way of establishing this could be that the programmer provides the required reasoning herself, as annotations or lemmas that hold true at different program points. But this is difficult for a number of reasons – how to generate these annotations/lemmas, how to know that the lemmas themselves are correct, and how to know that all lemmas together suffice for the proof. It is at this point that automated program verification comes to the rescue.

This paper addresses the problem of verifying programs submitted by students as assignments in an introductory programming course. These programs are to be evaluated for correctness with respect to the corresponding problem descriptions. Since such courses usually attract a large number of students, it is not just desirable but essential do this evaluation automatically. In order to employ automatic program verification techniques, a formal specification is necessary to verify against. However, it is a difficult task to derive such a specification automatically from the problem statement. Therefore, we assume that a formal specification, in the form of a reference implementation for the problem, has also been provided by the teacher. In particular, we assume that there is a designated output variable where both the teacher and the students are supposed to store the output computed by their program, and a student's program is said to be correct if it *always* produces the same output as the teacher's program.

A naive solution to this problem is to take the two programs (the correct one by the teacher, and a proposed solution by a student), sequentially compose them, and place an assertion at the end that the outputs do match. This composed program may now be given to any off-the-shelf verification tool. But the task is still far from done; the verifier needs to capture full functionality of both the programs, leading to complex invariants which are difficult to obtain, even for very simple programs and properties. A cheaper alternative to verification that is usually taken in practice, be it assignments in a programming course or real-world software, is testing. A program can be tested for a number of test-cases to gather empirical evidence of its correctness. With a large number of carefully chosen test-cases, it is indeed possible to capture most of the bugs in any program or be convinced that there are none. However, this process is inherently limited in the sense that it cannot give correctness guarantees. Discovering shallow bugs (i.e., cases of output mismatch) may also be done very efficiently on the sequentially composed program by passing it to a bounded

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381747

model checker (e.g. Cbmc [4]). But it suffers from the same problem – it may only guarantee bounded safety, not safety in general.

This paper connects the problem of verifying programming assignments to that of *hypersafety verification*. Hypersafety verification (or, verification of hypersafety properties) is a problem that has gained a lot of interest in the last few years [9, 27]. A hypersafety property describes the set of valid interrelations between multiple finite runs of a program. In particular, a k-safety property is one whose violation is witnessed by at least k finite runs of a program [5]. Azadeh et al. have very recently proposed an effective approach [9] for proving k-safety properties by composing k (memory-disjoint) copies of the program in parallel, and looking for a *reduction* of the composed program which: *i)* is easier to prove, and *ii)* when proved correct, would imply that the composed program was correct too. A reduction is typically obtained by removing redundant traces owing to statements that are *independent*. The focus of their technique is to look for a good reduction, i.e. one that admits a simple proof. With this in mind, we propose a novel approach of verifying programming assignments – by sequentially composing the reference and student's programs, and looking for a reduction that is easy to prove. We also explored the feasibility of this idea through an initial set of experiments, by using a hypersafety verification tool named *weaver*[1] to verify several sample assignment problems.

The core contributions of this paper are:

- *an idea* – that verification of an assignment program *wrt* a correct implementation can be done by finding a reduction, of their sequential composition, that is both easy and sufficient to prove (in a way similar to [9]), and
- *its demonstration* – that this is not only practically feasible, but also an efficient way to address this problem.

Outline of the paper We now start with an illustration of the hypersafety verification technique of [9], and a few examples that demonstrate our idea (Sect. 2). After that, we present an informal sketch of how the approach works, and point to the artifacts from our initial experiments on this (Sect. 3). This is followed by a discussion of the related work in Sect. 4. Finally, Sect. 5 concludes the paper and suggests interesting directions of pursuing this further.

2 ILLUSTRATIVE EXAMPLE

We briefly illustrate the hypersafety verification technique of [9], by taking an example from that paper itself. Consider the program shown in Listing 1 *(i)*. The program takes as input two non-negative numbers and multiplies them. Let us say that we wish to prove that multiplication distributes over addition. For that, consider three copies of the multiplication program, shown in Listing 1 *(ii)*, *(iii)*, and *(iv)*, with inputs as $\langle(a+b), c\rangle$, $\langle a, c\rangle$, $\langle b, c\rangle$. Now, if these three copies are composed sequentially, then the analysis requires non-linear invariants to be deduced. In particular, at the end of the first copy, we would need to know that $x_1 = (a+b) * c$. On the other hand, if we can discover a reduced program like the one shown in Listing 2, it suffices to have a linear loop invariant $x_1 = x_2 + x_3$. This loop invariant is inductive for both loops in the reduced program, and is also sufficient to prove the property. Note that the crux of the technique is to find a reduction which captures all program behaviors, and is also easy to prove at the same time. Their tool, weaver, does this efficiently by solving these problems together. I.e., it searches for the reduction and its proof at the same time.

```
int a, b;
int i = 0, x = 0;

while (i < a)
    x = x + b;
    i = i + 1;
```

(i)

```
int a, b, c;
int i1 = 0, x1 = 0;

while (i1 < a + b)
    x1 = x1 + c;
    i1 = i1 + 1;
```

(ii)

```
int a, c;
int i2 = 0, x2 = 0;

while (i2 < a)
    x2 = x2 + c;
    i2 = i2 + 1;
```

(iii)

```
int b, c;
int i3 = 0, x3 = 0;

while (i3 < b)
    x3 = x3 + c;
    i3 = i3 + 1;
```

(iv)

Listing 1: Illustrative example for hypersafety verification

```
int a, b, c;
int i1 = 0, i2 = 0, i3 = 0;
int x1 = 0, x2 = 0, x3 = 0;

while (i2 < a)
    x1 = x1 + c; i1 = i1 + 1;
    x2 = x2 + c; i2 = i2 + 1;

while (i3 < b)
    x1 = x1 + c; i1 = i1 + 1;
    x3 = x3 + c; i3 = i3 + 1;
```

Listing 2: Reduction of Listing 1

Consider the code shown in Listing 3. This solves the task of multiplying two integers given as input. Let us say the teacher solves it by going in a loop from 1 to the first integer input, and adding the second input to a sum (initially 0) each time. The student, on the other hand, does it in the other order – she goes from 1 to the second integer input, and adds the first input each time. For verifying a program that sequentially composes these components, a non-linear invariant $a * b = b * a$ is needed. The code was given to VeriAbs[2], but the tool timed out after 900 seconds and failed to verify the program. An equivalent code (in the weaver's input format) was given to weaver for verification and it was able to verify that quickly.

Correct/Reference version

```
int i = 0, m = 0;

while(i < b)
    m = m + a;
     i = i + 1;

return m;
```

Student version

```
int i = 0, m = 0;

while(i < a)
    m = m + b;
     i = i + 1;

return m;
```

Listing 3: Multiplication of positive integers

[1]an implementation from Azadeh et al. [9], available at https://github.com/weaver-verifier/weaver

[2]VeriAbs [8] has been the winner in the ReachSafety category at SV-COMP for last two years [2, 21]

For another example, consider the problem in listing 4. The task here is to find the the sum of all the multiples of 3 or 5 below 1000, and was adapted from a problem from an online programming problems repository. Here the student version is incorrect. When given to VeriAbs, it timed out after 900 seconds and failed to check correctness, while weaver was able to find a proof for incorrectness.

Correct/Reference version

```
int s = 0, i = 1;
int m3 = 0, m5 = 0;

while(i < 1000)
    m3 = i; m5 = i;

    while(0 < m3)
        m3 = m3 - 3;
    while(0 < m5)
        m5 = m5 - 5;

    if((m3 == 0) ||
       (m5 == 0))
        s = s + i;

    i = i + 1;
return s;
```

Student version

```
int s = 0, i = 1;
int m3 = 0, m5 = 0;

while(i < 1000)
    m3 = i; m5 = i;

    while(0 < m3)
        m3 = m3 - 3;
    while(0 < m5)
        m5 = m5 - 5;

    if((m3 == 0) &&
       (m5 == 0))
        s = s + i;

    i = i + 1;
return s;
```

Listing 4: Finding the sum of all multiples of 3 and 5

The problem in listing 5 is that of primality testing. Here, we composed two copies of the same program, assuming that the student's implementation is same as that of teacher's. Once again, we were able to verify this with weaver, while VeriAbs timed out.

```
int i = 2; int j;
bool b_break = false;
bool is_prime = true;

while((i < n) && (!b_break))
    j = i;
    while(j < n)
        j = j + i;
    if(j == n)
        is_prime = false; b_break = true;
    i = i + 1;
return is_prime;
```

Listing 5: Primality testing for natural numbers

3 OUR APPROACH

We have seen, with the examples illustrated in the previous section, how the problem of verifying that two programs give the same output may be addressed using a technique for automated hypersafety verification. Here, we give an intuitive idea of why this works. We have seen that an off-the-shelf verification tool struggles to do this, for the reason that full-functionalities of all component programs need to be captured and that may lead to complex invariants. Weaver works around this problem by finding a reduction of the composed program, and proving that correct. This is sound, because the reduction (equivalently, a set of traces of the composed program) is chosen such that it is representative of all possible behaviors. The way to obtain a reduction is to exploit the independence relation between statements of the program, and remove redundant traces. For example, if s and s' are two independent statements (say, if they are from separate components), then a reduction need not capture the trace $\langle P; s'; s; Q\rangle$, if it already captures $\langle P; s; s'; Q\rangle$ (where P and Q are sequences of program statements). This is inspired by partial-order reduction techniques for verifying concurrent programs [11]. However, a reduction obtained like this may also be difficult to prove. To deal with this, weaver does not construct a reduction separately and looks for its proof. Instead, it finds a set of reductions *simultaneously* with a proof that can prove at least one of the reductions in the set. The set of reductions, by construction, have the property that a proof of any member of the set establishes the correctness of the original, composed program.

Weaver does the above in a counterexample-guided fashion, using an interpolating prover as an oracle. We do not give the details of their algorithm due to lack of space. Besides, that is not the focus of this paper; our purpose here is to relate the two problems and demonstrate that this can indeed be done in practice. Apart from the examples illustrated in Sect. 2, we have tried the proposed approach on a few more sample assignment problems. Those problem statements, along with the corresponding programs (both in C, and in weaver's input format), and the results have been made available at *http://bit.ly/nier2020* as artifacts. It must be noted that the input format of weaver, and converting to it, is currently a limitation of our approach. Our immediate goal is to work on an implementation that overcomes this.

4 RELATED WORK

The novelty of our work lies in connecting a hypersafety verification technique, with correctness of programming assignments – in particular, to show that a reduction based technique for the former can effectively be used for the latter. Since evaluation of programming assignments has been studied in many different contexts, we only give a brief overview of the relevant literature.

There are several steps involved in programming assignment assessment [12, 18, 22], e.g. checking for plagiarism, testing correctness of the program, giving valuable feedback, etc. Our focus is on evaluating the correctness of programs, for which testing is the most commonly applied approach [12, 18]. However, it is incomplete in the sense that it may uncover bugs but cannot provide correctness guarantees. It can be argued that plagiarism checking, in a manner, tests for similarity of student program against given references, but this testing is usually done in a superficial manner. These methods cannot be adapted as such to create a robust correctness checker. There are also methods of plagiarism check [6, 7, 15] that can possibly be adapted for this task, but it is not clear if they would lead to complete approaches or not. There has been work aimed at more complete approaches e.g. [17], where semantic notions of execution paths are used to check for correctness. But a proof may still be difficult to find. Our approach, though semantic, differs in that it looks for a reduction that admits a simple, easy-to-find proof.

The problem of checking correctness of programming assignment against a reference implementation is ultimately an instance of program equivalence checking. Equivalence checking for sequential programs has been studied well in the context of semantic alignment [3], translation validation [16, 28], design and verification of compiler optimizations [19, 29], and program synthesis and superoptimization [1, 24]. A common approach to proving equivalence is constructing a simulation relation between a pair of programs. This can be done either by static analysis or a data driven analysis [10, 20, 23, 25, 26]. A drawback of this approach is the need for execution runs to have adequate coverage of the program. Gupta et al. present a static analysis based simulation relation construction for proving equivalence [13]. They do an incremental construction of a simulation relation, where at each incremental step, the invariants at the currently correlated program locations are inferred and future correlations are guided through the invariants inferred thus far. The pattern of using counterexamples to guide the search is found in weaver as well. Since the problem is undecidable in general, such patterns for searching can often be used to achieve better performance.

5 CONCLUSION AND FUTURE WORK

This paper connects two important problems in program verification – one of verifying programming assignments with respect to a correct program that is given, and the other of verifying hypersafety properties. This opens up the possibility of using techniques and tools being developed for hypersafety verification, to prove assignment programs correct. It also encourages us to speak about provable correctness of programming assignments, in contrast to the popular practice of only testing them.

For an idea paper like this one, it is imperative that there be a number of interesting directions of pursuing it further. An immediate goal, of course, is to work on the scalability of the proposed approach. This will enable a large class of applications for this technique, from verifying arbitrary assignment programs, to even doing an incremental analysis of software across its multiple versions. Another direction that looks worthwhile to explore is whether we can take advantage of the fact that this is only a 2-safety problem, and not the general k-safety one. It would also be interesting to see if the search for reductions may be combined with the search for a suitable abstraction; this would be useful given that the technique works in a counterexample-guided fashion anyway. The intent here being that abstractions would lead to even simpler proofs, that are a lot more easier to find.

REFERENCES

[1] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 394–403. https://doi.org/10.1145/1168919.1168906

[2] Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III.* 133–155.

[3] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

[4] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, Vol. 2988. Springer, 168–176.

[5] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210.

[6] G. Cosma and M. Joy. 2012. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Trans. Comput.* 61, 03 (mar 2012), 379–394. https://doi.org/10.1109/TC.2011.223

[7] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. 2010. Code Comparison System based on Abstract Syntax Tree. *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)* (2010), 668–673.

[8] Priyanka Darke, Sumanth Prabhu, Bharti Chimdyalwar, Avriti Chauhan, Shrawan Kumar, Animesh Basakchowdhury, R. Venkatesh, Advaita Datar, and Raveendra Kumar Medicherla. 2018. VeriAbs: Verification by Abstraction and Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 457–462.

[9] Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 200–218.

[10] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018.* 1–9.

[11] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag, Berlin, Heidelberg.

[12] Aleksejs Grocevs and Natālija Prokofjeva. 2016. The Capabilities of Automated Functional Testing of Programming Assignments. *Procedia - Social and Behavioral Sciences* 228 (2016), 457 – 461. https://doi.org/10.1016/j.sbspro.2016.07.070 2nd International Conference on Higher Education Advances,HEAd'16, 21-23 June 2016, València, Spain.

[13] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition. In *Theory and Applications of Satisfiability Testing – SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.). Springer International Publishing, Cham, 365–382.

[14] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[15] Hiroshi Kikuchi, Takaaki Goto, Mitsuo Wakatsuki, and Tetsuro Nishino. 2015. A Source Code Plagiarism Detecting Method Using Sequence Alignment with Abstract Syntax Tree Elements. *IJSI* 3 (2015), 41–56.

[16] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. *SIGPLAN Not.* 44, 6 (June 2009), 327–337. https://doi.org/10.1145/1543135.1542513

[17] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training* (Montreal, Quebec, Canada) *(ICSE-SEET '19)*. IEEE Press, Piscataway, NJ, USA, 126–137.

[18] Amit Kumar Mandal, Chittaranjan Mandal, and Chris Reade. 2007. A System for Automatic Evaluation of Programs for Correctness and Performance. In *Web Information Systems and Technologies*, Joaquim Filipe, José Cordeiro, and Vitor Pedrosa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–380.

[19] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. *SIGPLAN Not.* 35, 5 (May 2000), 83–94. https://doi.org/10.1145/358438.349314

[20] ThanhVu Nguyen, Timos Antopoulos, Andrew Ruef, and Michael Hicks. 2019. A Counterexample-guided Approach to Finding Numerical Invariants. arXiv:cs.SE/1903.12113

[21] Results of the 9th International Competition on Software Verification. 2020. https://sv-comp.sosy-lab.org/2020/results/results-verified/.

[22] Marko Pozenel, Luka Fürst, and Viljan Mahnic. 2015. Introduction of the automated assessment of homework assignments in a university-level programming course. *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2015), 761–766.

[23] Sumanth Prabhu, Kumar Madhukar, and R. Venkatesh. 2018. Efficiently Learning Safety Proofs from Appearance as well as Behaviours. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings.* 326–343.

[24] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 305–316.

[25] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 574–592. https://doi.org/10.1007/978-3-642-37036-6_31

[26] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. *SIGPLAN Not.* 48, 10 (Oct. 2013), 391–406. https://doi.org/10.1145/2544173.2509509

[27] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 161–179.

[28] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Savannah, GA, USA). ACM, New York, NY, USA, 264–276.

[29] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. *SIGPLAN Not.* 46, 6 (June 2011), 295–305. https://doi.org/10.1145/1993316.1993533