Training Difficulties in Deductive Methods of Verification and Synthesis of Program

Magdalina Todorova
Faculty of Mathematics and Informatics
Sofia University "St. Kl. Ohridski"
Sofia, Bulgaria

Daniela Orozova
Faculty of Computer Science and Engineering
Burgas Free University
Burgas, Bulgaria

***Abstract*—The article analyzes the difficulties which Bachelor Degree in Informatics and Computer Sciences students encounter in the process of being trained in applying deductive methods of verification and synthesis of procedural programs. Education in this field is an important step towards moving from classical software engineering to formal software engineering. The training in deductive methods is done in the introductory courses in programming in some Bulgarian universities. It includes: Floyd's method for proving partial and total correctness of flowchart programs; Hoare's method of verification of programs; and Djikstra's method of transforming predicates for verification and synthesis of Algol−like programs. The difficulties which occurred during the defining of the specification of the program, which is subjected to verification or synthesis; choosing a loop invariant and loop termination function; finding the weakest precondition; proving the formulated verifying conditions, are discussed in the paper. Means of overcoming these difficulties is proposed. Conclusions are drawn in order to improve the training in the field. Special attention is dedicated to motivating the use of specific tools for software analysis, such as interactive theorem proving system HOL, the software analyzers Frama−C and its WP plug−in, as well as the formal language ACSL, which allows formal specification of properties of C/C++ programs.**

Keywords—Program verification; deductive verification methods; automated theorem provers*; *proof assistants*; *education

I. Introduction

Applying formal methods of program verification and synthesis is an important part of the training in the introductory courses of programming in some Bulgarian universities. The experience which is shared is gained as a result of delivering such training through courses in the disciplines Introduction to Programming, Object−Oriented Programming and Data Structures for Bachelor's Degree students of specialties Informatics and Computer Sciences of Sofia University and Burgas Free University. Education is narrowed down to the following methods: Floyd's method of inductive statements for verification of flowchart programs, Hoare's method for verification of *while* programs, and Djikstra's method for transforming predicates for verification and synthesis of Algol−like programs. The limitations in the choice of methods of program verification and synthesis are imposed by the consideration that the training under investigation is delivered during the first three semesters of the Bachelor's Degree education, therefore the students lack sufficient knowledge in discrete mathematics and mathematical logic.

Some elements of the education are presented in the part, which is dedicated to the difficulties when applying deductive methods. Mere details on the training realization are provided in [1]-[4].

In [1], the following techniques are described, used in the education process in the field: axiomatic semantics, design by contract and generalized nets (GNs). Two main training approaches are considered. The first one combines the axiomatic semantics for proving total correctness of a procedural program with execution of the program [4]. The second one integrates axiomatic semantics in the GN models of the object−oriented programs under verification. The main stages of education and the process of education are considered. The results of the education are analyzed.

In [2], Floyd's method and the method of transforming predicates for program verification are presented. The main stages and methodology of training in these methods of deductive verification are discussed.

In [3], Hoare's method, the method of predicate transformer for synthesizing programs, runtime verification of programs and verification of object−oriented programs via developing their GN models are presented. The methodology of training in these methods of program verification and synthesis is considered. Examples of their application for the courses Introduction to Programming, Object−Oriented Programming and Data Structures are presented.

It is noted in the above cited articles that training in the field poses certain problems before the students; however, these problems has not been analyzed so far. Current article is dedicated to the analysis of the training difficulties; moreover, it presents means for overcoming these difficulties.

Bearing in mind the complexity of the field, the education is realized by: using simple examples; introducing the main terminology one at a time (e.g. precondition, postcondition, loop invariant, program specification); practicing the terminology not only during the lectures, but also during the seminars and lab sessions; applying deductive methods of verification is illustrated through automatized systems (such as HOL Interactive Theorem Prover [5] and the software analyzers Frama−C [6]); applying knowledge to realizing small projects.

Note that HOL and Frama−C systems are not introduced, due to the limited classes, only the results of their application are presented. The main goal is to demonstrate that knowledge about formal methods is useful and will support future software developers in designing, realizing and testing applications. Otherwise, part of the students may lose motivation for studying and applying formal methods in their practice.

The work is structured as follows. In Section II of the paper the difficulties, which students encounter when: defining the specification of the program under verification and synthesis; finding the weakest precondition; choosing an invariant and a termination function of the loop operator; proving the formulated verifying conditions, are analyzed. Dealing with difficulties during the training is presented in Section III. Some suggestions are given that assist: defining the specification; choosing an invariant and a termination function of the loop operator and proving the formulated verifying conditions. Recommendations for dealing with difficulties are described in Section IV.

II. Analysis of the Difficulties in Applying Deductive Methods of Program Verification and Synthesis

A. Defining the Specification

Program specification describes what has to be done as a result of program execution. At the beginning of the training students are introduced to Hoare's triple $\{Q\}\ S\ \{R\}$, where Q and R are predicates, and S is a program. This specification defines the total correctness of the program S with respect to Q and R. This can be interpreted as follows:

If executing S starts in a state which satisfies the predicate Q, what follows is that executing S terminates, after a limited time period, in a state which satisfies R.

The predicate Q is called *precondition*, and the predicate R – *postcondition.* Defining pre− and postconditions of some programs is a great challenge for some students. In most cases, these are programs solving tasks on each and existence. Proving the correctness of the specification $\{Q\}\ S\ \{R\}$ through applying Manna−Pnueli's rules [7], as well as proving partial correctness of S with respect to Q and R through applying Hoare's rules [8], is quite labor−intensive and demotivates even the best students to use the specification and the respective rules. That's why, for educational purposes, the more convenient specification, known as transforming predicate [9] is applied.

The transforming predicate $Wp(S, R)$ describes the set of all states, so that the start of the program execution from each of these states terminates, and the value of the output predicate R is *true.* $Wp(S, R)$ satisfies $\{Wp(S, R)\}\ S\ \{R\}$. Hoare's triple $\{Q\}\ S\ \{R\}$ is equivalent to $Q => Wp(S, R)$.

Finding $Wp(S, R)$, again, is far from a simple task, in most cases. For example, the definition of the transforming predicate of the loop operator is nearly unusable, but it helps to theoretically justify a methodology for verification and synthesis of code fragments containing loop operators [10].

That is why identification and check if the chosen specification holds is a complex activity. It requires of the students to have good knowledge of mathematical logic, as well as skills for defining and applying mathematical abstractions. This poses the main difficulty for training in the field: some of the students of specialty Informatics and Computer Sciences lack enough mathematical knowledge and skills. The complexity of the matter requires high motivation for applying these methods. However, first year students have little practical experience, they usually do not recognize the crucial importance of the formal methods for ensuring software quality. This demotivates them to apply any formal methods of software verification and synthesis. The lack of motivation is a great obstacle for the training in the field.

B. Choosing an Invariant and Termination Function for the Loop Operator

In order to verify a program formally, the following two tasks are to be solved:

- proving the partial correctness of the program with respect to given input/output specification;
- proving that the program terminates.

In order to solve these, the application of one of the above mentioned methods of deductive program verification and synthesis is discussed in the paper. The two tasks cannot be solved by any of these methods without finding and applying suitable invariants and termination functions of the loop operators of the program.

An invariant of a loop operator is a logical statement, which holds before the execution and after each execution of the loop operator.

The loop termination function is used to prove that the respective loop terminates. It gives the upper limit of the iterations to be completed by the end of the loop execution. The latter can be used to estimate the time left until the program ends.

In most cases, both loop invariant and loop termination function are not obvious. The task for correctly identifying them is of great importance for automatizing program verification. Solving this task is difficult, having in mind the volume and complexity of contemporary software, and the software realized for educational purposes, respectively. During the training process on finding the invariant of a loop operator, Gries's methodology [10] is applied. The loop invariant is seen as a weaker postcondition. The ways for finding a condition weaker than the postcondition, which are most commonly applied for educational purposes in finding the invariant of a loop are:

- deleting a conjunctive member;
- replacing a constant by a variable;
- enlarging the range of a variable.

Using only one of these three methods for generating a weaker postcondition sometimes does not lead to identifying a suitable invariant. In this case, a combination of these methods

is performed, as well as a combination of the precondition and postcondition.

Finding an invariant these ways is a labor−intense work. In addition, some questions arise, such as: Which conjunctive member of the postcondition to be eliminated, so a suitable invariant to be found? In case of more constants, which one to be exchanged for a variable? What limits the variable area to be increased to, and whose variable area to be increased? Which methods to be combined? Which parts of the pre− and postconditions to be combined?

Finding the respective invariant is related to checking several logical conditions, which poses additional difficulties.

For example, in order for P to be an invariant of the *while* loop of the program for finding the factorial of the natural number x (the example traditionally used as a loop containing program fragment)

```
y = 1; z = 0;
while (z != x)
{ z++;
  y = y*z;
}
```

P must satisfy the following conditions:

$\{ P \wedge z \neq x \}$ z++; $y = y*z$ $\{ P \}$

$y = 1 \wedge z = 0 \Rightarrow P$

$P \wedge \neg (z \neq x) \Rightarrow y = x!$

where $y = x!$ is the postcondition.

In most cases, checking for meeting these conditions proves to be a complex task.

In order to find the loop termination function, an assumption can be applied that it gives the upper bound on how many iterations remain to be executed before loop termination occurs. The invariant of the loop operator suggests the definition of its termination function in almost all applications developed by the students. Therefore, the correct identification of an invariant is closely related to the choice of a loop termination function. Thus, the difficulties in finding the invariant also reflect on finding the loop termination function.

C. Proving the Formulated Verifying Conditions

Although proving the verifying conditions is narrowed down to proving the truthfulness of a system of implications, proving these implications manually, more often than not, is not an easy task. Even educational examples require the students to have basic knowledge in discrete mathematics and mathematical logic. They should also be able to work with propositions, in order to perform equivalence transformations (to know the laws of equivalence; the rules of substitution and transitivity). They have to have also at least basic knowledge on deductive proofs (inference rules; proofs and subproofs), and on predicates (extending the range of a state; quantification; free and bound identifiers; some theorems about textual substitution and states). In addition, they have to be aware of notations and conventions for arrays.

Supposedly, a great deal of this knowledge is to be acquired as a result of studying the following disciplines: Discrete Mathematics; Languages, Automata and Calculability; Introduction to Software Engineering; Algebra; Geometry and Mathematical Analysis, which are being taught in parallel with the courses in programming. However, a large number of the students have a different predisposition and are not motivated enough to study mathematics disciplines, as stated above.

Apart from the difficulties identified earlier, the following are of importance as well:

- Lack of environments for teaching in the field. There are not enough tools for automatizing the process of applying deductive methods, which to be applicable to training beginner programmers. Environments are needed to facilitate students in applying deductive methods of verification.
- There are no adequate didactic materials to support such education.
- The textbooks on the matter are not enough.

III. Dealing with Difficulties During the Training

Our experience in teaching formal methods at academic level, in addition to our observations on how graduates apply the knowledge in the field in their practice, made us believe that education in formal methods of verification and synthesis is useful and needed. It is only through training that these methods are applied in software industry. In order for this to be successful, measures for overcoming the difficulties described in Section II have to be taken.

Some suggestions for coping with these difficulties are proposed below.

A. Defining the Specification

In order to tackle the problems with defining the specification, the students need to get acquainted and are taught to apply some specification language. The experience with ANSI/ISO C Specification Language (ACSL) [11]-[13] provides ample evidence in favor of recommending it for applying formal methods of verification of C and some C++ programs. This language allows for relatively easily specifying properties of C and some C++ programs, after which these properties to be formally verified. ACSL is a Behavioral Interface Specification Language implemented in the Frama−C framework WP plug−in. Frama−C is an open code platform, analyzing source code written in the programming language C. It combines the following analysis techniques in a common framework: Frama−C's WP plug−in, Frama−C's value analysis plug−in, Frama−C's RTE plug−in and Frama−C's E−ACSL plug−in. Frama−C's WP plug−in is suitable for education purposes.

The Frama−C/WP plug−in enables deductive verification of C programs that have been annotated with ACSL. This plug−in uses Hoare−style weakest precondition computations to formally prove ACSL properties of a C code. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants [13].

By using the formal language ACSL, the software analyzers Frama−C and Frama−C's WP plug−in, defining of the program specification by the student will be narrowed down to: defining program precondition and postcondition, of the invariants and termination functions of the loop operators, used in the program. Thus, the learner will not have to find the transforming predicate, as well as to prove the formulated statements.

Fig. 1 shows the definition of the function *sqrt2*, which finds the biggest integer, whose square is not bigger that n (n is a given non−negative integer). The definition is annotated according to the specification, given in ACSL language.

The precondition (*INT_MAX*/2 > n >= 0) is given via the requires clause, and the postcondition (*result* >= 0 && *result* * *result* <= n < (*result* + 1) * (*result* + 1)) − through the ensures clause. The invariant (a >= 0 && a * a <= n) and the loop termination function (n – a * a) are given by the clauses loop invariant and loop variant, respectively (see Fig. 1).

The result of realizing this specification through the WP plug−in of Frama−C (see Fig. 2) shows that 11 goals are achieved (5 goals are simplifications, done via the simplifier Qed that is integrated into Frama−C/WP, and 6 goals are proofs, done via the SMT solver Alt−Ergo).

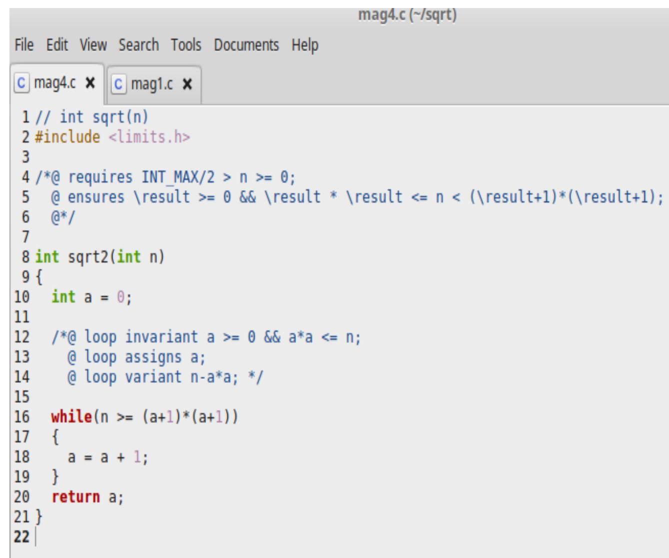

Fig. 1. Function sqrt2, specified through ACSL.

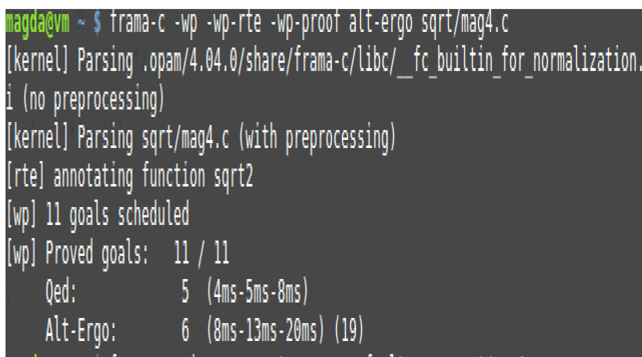

Fig. 2. Results of analysis of the function *sqrt2* via Frama−C.

B. Choosing an Invariant and a Termination Function of the Loop Operator

As the choice of an invariant of the loop operator is related to proving conditions, a means of facilitating the solving of this task is using automated theorem provers. Some of the most successful systems for theorem proving are: HOL Light, Mizar, ProofPower, Isabelle and Coq. An experience regarding formal verification of procedural and object−oriented programs using the theorem prover system HOL is shared in [14]. Through this theorem prover, it can be checked if the predicate P, chosen to be an invariant of the operator for the loop *while* (B) S, satisfies the conditions for an invariant:

$$\{ P \wedge B \} \; S \; \{ P \}$$

$$Q => P$$

$$P \wedge \neg B \; => R$$

Where Q is the precondition and R is the postcondition of the loop operator. It also can be used in proving if the loop termination function t satisfies the following conditions:

$P \wedge B => t > 0$ and

$$P \wedge B => Wp(t1 = t;\, S,\, t < t1)$$

Where $t1$ is the value of t before executing the body S of the loop.

Another suitable module for finding a loop invariant is *Jessie*, complemented by *Apron* library. The module *Jessie* is included in Frama−C.

In order to find a loop termination function, the following strategy can be applied: wording down the functions of the loop termination function; formalizing the wording as a mathematical expression; checking if the mathematical expression satisfies the formal requirements for a loop termination function. It is recommended the checking to be performed both manually and via any of the automatic theorem provers.

C. Proving the Formulated Verifying Conditions

In order to prove the verifying conditions, the HOL Interactive Theorem Prover, as well as some of the theorem provers Qed, CVC4, Z3, Alt−Ergo, CVC3, E and Coq, which are supported by the Software Analyzers Frama−C can be applied.

IV. Recommendations for Dealing with Difficulties

The experience gained as a result of the training justifies the formulation of the following recommendations for dealing with difficulties in applying deductive methods of program verification and synthesis:

- Manual application with automated tools to be integrated. Thus, part of the problems will be avoided, and a better balance between simplicity, visualization and precision will be maintained.
- Programming environments, adequate to the educational goals, to be developed.

- Student motivation regarding studying the field to be increased. To this end, adequate samples to be designed, through which the advantages of deductive methods for ensuring the quality of industrial software and for decreasing the price of software project realization to be demonstrated.
- More efficient educational methods in the field to be introduced [15]. In addition to teaching through examples [16] and project−based learning approach [17], it is essential the training to be organized with respect to student knowledge level. For each level, appropriate methods to be chosen and applied.
- The teaching experience regarding the difficulties encountered during such education to be more widely discussed. The quality of education to be unified by employing cloud management [18].
- Cloud−based educational networks to be established in order to facilitate trans−institutional collaboration on creating and applying educational products and services in the field of programming, and formal methods of verification and synthesis, in particular [19].
- Appropriate formalization of programming language teaching to be made [20].
- Appropriate didactic materials on the matter to be designed to support the training.
- Studying the field to be given the status of a separate core discipline. Thus, each student will have to study and apply formal methods.

V. Conclusion

Education in the field of applying formal methods for developing correct software is the most efficient way of implementing these methods in software industry. The reason to state this is that a relatively large portion of the trained students continue using these methods in their further study at both Bachelor and Master degrees. This tendency is especially visible during courses such as Numerical Methods and Robotics [21]. Some of the graduate students who have completed this training also try to apply it in their practice as software specialists. Others continue their study at a PhD level in the field.

In order to overcome the challenges during the training in deductive methods of program verification and synthesis, an educational environment for verification of procedural and object−oriented programs is under development [22]. The environment is based on GNs and only provides tools for training in program verification so far. Future work includes expanding education framework with tools for supporting program synthesis, as well as integrating automatic systems for theorem proving in it.

References

[1] M. Todorova, “Applying program verification methods in software specialists education,” Proceedings of the 7th International Technology, Education and Development Conference, Valencia, Spain, pp. 6260−6270, 2013.

[2] M. Todorova and D. Orozova, “Applying deductive verification to bachelor degree courses in programming,” Proceedings of the the 10th Annual International Conference of Education, Research and Innovation, Seville, 16−18 November, 2017, pp. 5055−5065, 2017.

[3] M. Todorova and D. Orozova, “How to build up contemporary computer science specialists – formal methods of verification and synthsis of programs in introduction courses on programming,” Proceedings of the 9th annual International Conference of Education, Research and Innovation, Seville, 14th, 15−16 November, 2016, pp. 4249−4256, 2016.

[4] M. Todorova and P. Armyanov, “Runtime Verification of computer programs and its application in programming education,” Global Science and Technology Forum: International Journal of Mathematics, Statistics and Operations Research, vol. 1, No. 1, pp. 105−110, 2012.

[5] Realease Notes for HOL4, Kananaskis−11, https://hol−theorem−prover.org/kananaskis−11.release.html, 2012.

[6] Frama−C Software Analyzers, https://frama−c.com/index.html

[7] Z. Manna, Mathematical Theory of Informatics, Science and Art Publishing House, Sofia, 1983.

[8] C. A. R. Hoare, “An axiomatic basis for computer programming,” Communication of the ACM, vol. 12, No 10, 1969.

[9] E. Dijkstra, Discipline of Pprogramming, Prentice Hall, 1976.

[10] D. Gries, The Science of Programming, Springer−Verlag, New York, Heidelwerg, Berlin, 1981.

[11] V. Prevosto, ACSL Mini−Tutorial, CEA List, INRIA Centre de Recherche SACLAY – ILE de France.

[12] P. Baudin, J.Filliatre, C. Marche, B. Monate, Y. Moy, and V. Prevosto, ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4, http: //www.frama-c.cea.fr/ download/acsl_1.4.pdf, 2008.

[13] J. Burghardt, R. Clausecker, J. Gerlach, and H. Pohl, ACSL By Example. Towards a Verified C Standard Library, Version 14.1.1 for Frama−C (Silicon), 2017.

[14] M. Todorova, “Formal verification of procedural and object−oriented programs using the HOL theorem proof system,” Automatics and Informatics, John Atanasoff Union of Automatics and Informatics, ISSN 0861–7562, vol. XLII, No. 2, pp. 25−27, 2008.

[15] P. Armyanov, A. Semerdzhiev, K. Georgiev and T. Trifonov, “The effects of progressive evaluation and obligatory homeworks on student motivation and achievements,” Proceedings of the 12th International Technology, Education and Development Conference, Valencia, Spain, 2018, pp. 618−625, 2018.

[16] I. Donchev, “An approach to teaching object−oriented programming concepts by examples,” Thirty Seventh Spring Conference of the Union of Bulgarian Mathematicians, Borovets, April 2−6, 2008, pp. 335−341, 2008 (in Bulgarian).

[17] K. Kaloyanova, “An implementation of the project approach in teaching information systems courses,” 8th International Technology, Education and Development Conference, Valencia, Spain, pp. 7090−7096, 2014.

[18] S. Hadzhikoleva and E. Hadzhikolev, “QAHEaaS or quality assurance in higher education as a service,” Tem Journal, vol. 5, No.3, 2016, pp. 363−370, ISSN: 2217−8309.

[19] S. Hadzhikoleva, E. Hadzhikolev, S. Cheresharov, and L. Yovkov, “Towards building cloud education networks,” Tem Journal, vol.7, No.1, 2018, pp. 219−224, ISSN: 2217−8309.

[20] V. Dimitrov, Deriving semantics from WS−BPEL specifications of parallel business processes on an example, Computer Research and Modeling, vol. 7, No 3, pp. 445−454, 2015.

[21] I. Patias and V. Georgiev, Design of Robotic Systems, St Kliment Ohridski University Press, ISBN 978−954−07−4207−6, 2017.

[22] M. Todorova and K. Kanev, “Educational framework for verification of object−oriented programs,” The Joint International Conference on Human−Centered Computer Environments HCCE’2012, Hamamatsu, Japan, pp. 23−27, 2012.