STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **L**, Number 1, 2005

CORRECTNESS: A VERY IMPORTANT QUALITY FACTOR IN PROGRAMMING

MILITON FRENŢIU

ABSTRACT. Correctness is one of the most important property of a program. Nevertherless, the students are usually not taught to prove the correctness of their programs, and when it is done, they dislike this activity. The importance of program correctness, and the necessity to teach it from the first course in programming, are underlined. Also, some consequences on the education activity are considered.

Keywords: Program correctness, Education, Software Engineering, Stepwise refinement

1. Introduction

Todays software systems have become essential parts of our everyday life. Computers are used in all fields of human activities. More and more programs are needed, and their complexity increases continuously. Software critical systems [But94] require error-free programming. To obtain programs without errors we need a new way of writing programs, new people, much better educated, able to do this.

There is a contradiction between the desire to obtain a system as quickly as possible, and to build a correct system. The experience shows that more than 75% of finished software products have errors during maintenance, and deadlines are missed and cost overruns is a rule not an exception [Gib94]. It was estimated [You90] that more than 50% of the development effort was spent on testing and debugging. Nevertheless, some errors are not detected by testing, and some of them are never detected. More, there are projects that have never been finished [Eff94]. And it is not an exception; it is estimated that from each six large projects two of them are never finished [Gib94, Rob98, or 31.1% according to Sta95]. Also, some other examples and problems may be found in [Gib94].

Due to unreliable software, many people died, and serious economical damages have been produced. There are many well known examples. Some of them are mentioned below, and more (107) can be found in [Der05].

Received by the editors: April 15, 2005.

During the Gulf war, on February 25-th 1991, a Patriot missile failed to intercept an incoming Iraqi Scud missile, due to accumulating errors with real numbers computations [Arn96, Der05]. Due to this error 28 American soldiers died.

On June 4, 1996, the first flight of the Ariane 5 rocket ended in failure. Approximately 40 seconds after initiation of the flight sequence, Ariane 5 exploded [Jez97]. The European Space Agency and French National Center for Space Studies, established an inquiry board to determine the cause of the accident. This accident happened due to a software reuse error of an integer conversion procedure.

On September 23, 1999 the Mars Climate Orbiter was lost when it entered the Martian atmosphere in a lower than expected trajectory [Lev01]. The cause of the accident was in giving data in English units instead of metric units.

According to Andrews [And02], "in the past 20 years, there have been approximately 1100 computer-related accidental deaths.

Let us consider that a program is composed of n procedures, and the probability of correctness of each procedure is at most p. Then the probability of correctness of the entire program is at most p^n [Dah72]. For $p < 1$ and n very large this value is very closed to 0. It means that the only chance to obtain a reliable program is $p = 1$, i.e. each procedure must be perfect. This means that we must prove the correctness of all used procedures, or, as will be shown later, we must obtain a correct procedure when we conceive it.

Therefore, the most important property of a program is whether it accomplishes the intentions of its user, i.e. if it is correct.

2. A short overview on program correctness

The concept of program correctness was introduced by Floyd [Flo67]. Also, a method for proving program correctness was given in the same paper. Before, the correctness of individual algorithms was proved by Hoare [Hoa67, Hoa68, Hoa71], Foley and Hoare [Fol71], London [Lon70a, Lon70b], Naur [Nau66].

In a program P we distinguish three types of variabiles, grouped as three vectors X, Y, and Z. The input vector $X = (x_1, x_2, \ldots, x_m)$ consists of the input variables. They denote the known data of the problem PP solved by the program P. We may suppose they do not change during computation. The output vector $Z = (z_1, z_2, \ldots, z_m)$ consists of those variables which denote the results of the problem PP. The program vector Y consists of the auxiliary variables, which denotes various partially results of the computation.

Two predicates are associated to the program P: an input predicate and an output predicate. The input predicate $\varphi(X)$ is TRUE for those values a of X for each the problem may be solved. The output predicate $\psi(X, Z)$ shows the relation between the results Z and the input values X. It is TRUE for those values a and b of the vectors X and Z for which the results of the problem are b when the initial/input data is a. The specification of the program P is the pair formed from the input predicate $\varphi(X)$ and the output predicate $\psi(X, Y)$.

The program P terminates in respect to the input predicate $\varphi(X)$ if for each value $a = (a_1, a_2, \ldots, a_n)$ of the vector X for which the predicate φ is TRUE, the execution of P terminates. In this case, the computation done by P is the sequence of states passed during the execution, and the value b of the vector Z in the final state is the result of the execution. We may write $b = P(a)$, i.e. P implements a function.

The program P is partially correct with respect to the specification if for the value a for which $\varphi(a)$ is TRUE and the execution terminates with the results $b = P(a)$ then $\psi(a, b)$ is TRUE. The program P is totally correct with respect to $\varphi(X)$ and $\psi(X, Y)$ if the program P terminates with respect to $\varphi(X)$ and it is partially correct with respect to $\varphi(X)$ and $\psi(X, Y)$.

A method for proving partial correctness of a flowchart program is due to Floyd [Flo67] and it uses a set of cut-points. This is a set of points on the arcs of the flowchart such that every loop includes at least one such cut-point. Also, there is a cut-point on the arc leading from the START box, and there is a cut-point on the arc leading to the HALT box.

To each cut-point i of the flowchart, a predicate $\mu_i(X, Y)$ is associated. This predicate, called an invariant predicate, is invariantly true for the current values of X and Y in this cut-point, i.e. it characterizes the relation that must exist between variables at this point. At the START cut-point the corresponding invariant predicate is $\varphi(X)$, and at the HALT cut-point it is $\psi(X, Z)$.

The set of cut-points defines the paths that must be verified. Let α be a path leading from the cut-point i to the cut-point j, with no intermediate cut-points (there can be more such paths). To this path we associate a predicate $R_\alpha(X, Y)$ which gives the condition for the path α to be traversed, and a function $r_\alpha(X, Y)$ such that if Y are the intermediate values in the cut-point i then, when the path is traversed, $Y' = r_\alpha(X, Y)$ are the values of Y in the cut-point j. A verification condition is associated to the path α. This condition is:

$$\forall X \forall Y (\mu_i(X, Y) \wedge R_\alpha(X, Y) \rightarrow \mu_j(X, r_\alpha(X, Y)))$$

Floyd [Flo67] proved that if all the verification conditions are true then the program is partially correct with respect to $\varphi(X)$ and $\psi(X, Z)$.

Floyd also suggested a method for proving termination using well-founded sets. A well-founded set M is a partially ordered set, without infinite decreasing sequences. For each path α from i to j, a termination condition is formed:

$$\varphi(X) \wedge R_?(X, Y) \rightarrow (u_i(X, Y) > u_j(X, r_\alpha(X, Y))$$

Here

$$u_i : D_X \times D_Y \rightarrow M$$

is a function associated to the cut-point i.

If all termination conditions are proved then the program P terminates over $\varphi(X)$.

The ideas of Floyd were developed by Hoare [Hoa69] who introduced an axiomatic method for proving the partial correctness of a program.

Then Dijkstra [Dij75] introduces the important concept of weakest precondition. His idea, of formally deriving correct programs from specifications, was continued by Gries [Gri81] who states that is more important to develop correct programs than to prove latter their correctness: A program and its proof should be developed hand-in-hand with the former usually leading the way. This idea was developed further by Dromey [Dro89], and Morgan [Mor90].

3. Consequences of Program correctness theory on teaching Programming

One way to change the software engineering situation shortly described in the Introduction, also known as "software crisis, is a better education of the new generations of programmers. It is the time to teach programming in conformity with the theory of program correctness. As Naur has underlined in [Nau66], "it is a deplorable consequence of the lack of influence of mathematical thinking on the way in which computer programming is being pursued. We think that teaching programming well is an important part of our tasks as teachers in the universities.

It is not only possible, but necessary, to explicitly teach the methods and principles for good programming. Some early papers on program correctness [Nau66, Lon70a, Lon70b] have proved the correctness of some concrete algorithms. Just Hoare has made a significant move from a posteriori proof of an existing program [Hoa67, Hoa68, Fol71] to a program design method [Hoa71].

We need to teach program correctness for many reasons. First one is the impact this theory has on the future programmers in general. Second, we need very skilled people for program verification activities. The old testing is needed, but not sufficient and not efficient. Program inspection [Gil93] is required for CMM level3 [Pau93], and the inspection team must contain people aware of program correctness theory. More, we need skilled people to use Formal Methods for building all future safety-critical systems [But94].

Another reason for teaching program correctness comes from the consequences of program correctness on programming methodology. Why do we need specifications? What is the importance of program clarity and simplicity in software engineering? Why we must write all kind of documents?

Some of the most important rules considered important for programming well, which are consequences of the program correctness theory, are given in [Fre93, Fre94, Fre97]. We select some of the most important and simple ones below (specifying the original bibliographical source):

- Define the problem completely (i.e., write the precondition $\varphi(X)$ and the postcondition $\psi(X, Z)$) [Gri81, Led75, Sch90].
- Think first, program later [Led75].

- Write and use modules as much as possible [Led75, Sch90].
- Prove the correctness of algorithms during their design [Gri81].
- Decide which are the needed program variables, and what are their meanings. Write invariants for these variables and insert them as comments in the program [Gri81, Led75].
- Choose suitable and meaningful names for variables [Led75].
- For each variable of a program, make sure that it is declared, initialised and properly used [Nau66].
- Verify the value of a variable immediately it was obtained [Nau66].
- Use comments to document the program [Led75].
- Verify each part of a program as soon as possible [Gri81, Sch90, Gil93].
- Use symbolic names for all entities (constants, types, variables, procedures and functions) [Fre94].
- Avoid to use global variables [Sch90].
- Hand-check the program before running it [Led75].
- Write the documentation of the program simultaneously with its building [Sch90].
- Give attention to the clarity and simplicity of your program [Fre01a]!

Some people [Ste91] are against proving correctness. Others consider it is expensive, since they think the effort to build a program in such a way is considerable increased.

Certainly, proving correctness of a real large program is too complicated and, maybe, inappropriate. But the correctness of the important and difficult procedures used in the system may be proved. And the specifications of these procedures are not changed from time to time by the client, as some “researchers argue against proving correctness.

Also, proving correctness has an important impact on program verification. It is well known that proving correctness and testing complete each other. There are some aspects (performance, for example) that can be verified only by testing. But testing is a time consuming activity that must be reduced through other forms of verifications (inspections [Fag76, Mye78, Gil93], symbolic executions [Kin76]). And a correct algorithm is not accompanied by debugging. When testing discover errors we must start the most difficult and unpleasant life-cycle activity, which is debugging. The minor errors may easily be corrected, meanwhile debugging logical errors consumes much time and effort, and often are unsuccessful.

Almost all students have learned some programming at school. They think they know what programming is all about. They express resentment when they a forced to document their activity, to design the program, or to think to the correctness of their programs. They run directly to computer and introduce their programs, and run them. If the first execution seems OK they are satisfied. They are not used to test seriously their programs. We need to fight with these people, to change their habit, to educate them in a different way. That is why we have decided to stress from the very beginning that "programming is a high intellectual activity [Hoa91], that the correctness is the most important property of good programs.

According to ISO-9126 the factors of the program quality are functionality, reliability, usability, efficiency, maintainability, and portability. Is program correctness present in defining the quality of a program? We put this question since it is not directly mentioned in the quality factors.

Nevertheless, correctness is clearly present in the above mentioned factors. Functionality supposes the program is correct, and reliability expects as few errors as possible. And maintainability is increased in many ways if the program is correct. First, crorrective maintenance is not needed for a correct program. Second, perfective and adaptive maintenance is easier for a well design program. And correctness is strongly connected to good design.

The software engineering course "Algorithms and Pascal programming was given to the first year computer science students. The course concerned was presented during their first semester in the University. It is the first course on programming, and the main parts of it are: a Pseudocode language for describing algorithms, developing correct algorithms from specifications, simple Pascal programming, the life-cycle of a program, the methods of designing, coding and verifying simple programs.

There are some years since the notion of correctness and the Floyds method for proving correctness are taught in this course. Also, it was underlined that it is more important to construct a correct program than to prove later its incorrectness. But is for the first time when the accent was given to develop correct algorithms from specifications, not on proving correctness.

We used for many years a Pseudocode language to describe algorithms. This language has those three computation structures needed to write structured algorithms: the sequential structure, the alternative and the iterative computation structure. Each year stepwise refinement [Wir71] was considered a very important programming method, but it was presented in an informal way. In 2004, for the first time, the development of correct algorithms from specifications was used in a more rigorous way [Mor90]. The refinement rules for assignement, for the sequential composition, alternation computation structure (if-then-else), and for iteration were adapted to the used Pseudocode language [Fre04].

We gave many examples of refinement, and the students exercised these rules. At the final examination we observed an improvement in the correctness of student algorithms. The results at students exams in 2004 were compared to those of the students on 2003 [Fre04], and the statistical hypothesis on mean values equality was verified. It was rejected, since the results were significantly better in 2004. Therefore, we may conclude that the correctness of the students algorithms at the examination has improved in the last year. Certainly, their may be more reasons for this, but the main difference consisted only in the introduction of the development of algorithms from specifications, using clear defined rules.

4. Conclusions

We need confidence in the quality of our software products. We need to educate the future software developers in the spirit of producing correct, reliable systems. For this we must teach students to develop correct programs. We are aware that usually programmers do not prove the correctness of their pograms. There always must be a balance between cost and the importance of reliability of the programs. But just when the well educated people do not prove the correctness, their products are more reliable than the products of those "programmers who never studied program correctness. Therefore, we consider that the students must hear, and must pay attention to the correctness of their products.

We think that we educate our students for their future profession, that last for several decades. Also, they must be prepared for changes in software engineering. They need to acquire now the knowledge needed to build more reliable systems. Our simple experiment confirmed the improvement in the correctness of students algorithms compared to the previous year.

Certainly, proving correctness of algorithms is not enough to obtain more reliable systems, but it is necessary. For years we ask the students to understand and to respect some important rules of programming [Fre93, Fre00, Fre03, Led75]. Many of them are simple consequences of the theory of program correctness [Fre97].

Thus, from first year, students are taught about program specification and design. The entire life-cycle is presented, the importance of documentation for all steps is underlined. Top-down and the other programming methods are taught, and the students hear that the design is more important than coding. Each part of the design must be specified, the code must be explained by comments, the comments should be neither more nor less than needed. Since we also observed that students do not like to write comments [Fre02] we tried to explain their necessity, and to force them to document the code [Fre03].

The fact that program verification is a very important activity which extends from the first statement of the problem, until the end of the project is repeated many times. And a special attention was given to the inspection of all documents. The students must hear from the beginning that testing is not enough, that inspection of all phases may be more useful. Thats why we ask the students to have a notebook at their laboratories, and the entire life-cycle of their programs must be reflected in this notebook. And they must respect the order of the activities; the specification and the design must be before coding.

We must fight against the idea that proving correctness is expensive, that proving needs time the developers think they do not have. We must tell them they have very much time for debugging!

References

(And02) Andrews P., Safety-critical projects: Can formal methods help?, Builder-com, 2002, sept. 12.

(Arn96) Arnold D.N., Two disasters caused by computer arithmetic errors, http://www.ima.umn.edu/~arnold/455.f96/disasters.html

(But94) Butler, R.W., S.C.Johnson, Formal Methods for Life-Critical Software, NASA Langley Research Center, http://smesh.larc.nasa.gov/

(Dah72) Dahl O.J., E.W.Dijkstra, and C.A.R.Hoare, Structured programming, Academic Press, New York, 1972.

(Der05) Dershowitz N., Software horror stories, School of Computer Science, TelAviv University, http://www.cs.tau.ac.il/~nachumd/verified/horror.html

(Dij75) Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm.A.C.M., 18(1975), 8, pp.453-457.

(Dro89) Dromey G., Program Derivation. The Development of Programs from Specifications, Addison Wesley, 1989.

(Eff94) Effy Oz, When Professional Standards are LAX. The CONFIRM Failure and its lessons, Comm. A.C.M., 37(1994), 10, 29-36.

(Fag76) M. Fagan, Design and Code Inspections to Reduce Errors in Program Development, IBM Systems Journal, 15 (3), 1976.

(Flo67) Floyd R.W., Assigning meanings to programs, Proc. Symposium in Applied Mathematics, 19, AMS, 1967, pp.19-32.

(Fol71) Foley M., and C.A.R.Hoare, Proof of a recursive program: Quicksort, The Computer Journal, vol.14,n.4, p.391-395.

(Fre84) Frentiu M., On the program correctness, Seminar on Computer Science, preprint 1984, 75-84.

(Fre93) M.Frentiu, B.Prv, Programming Proverbs Revisited, Studia Universitatis Babes-Bolyai, Mathematica, XXXVIII (1993), 3, 49-58.

(Fre94) M.Frentiu, B.Prv, Elaborarea programelor. Metode si tehnici moderne, Ed.Promedia, Cluj-Napoca 1994, 221 pages, ISBN 973-96114-9-4.

(Fre95) M.Frentiu, Reguli de programare pentru incepatori, in Lucrarile Conferintei "Informatizarea invatamantului, Balti, 4-7 octombrie 1995.

(Fre97) M.Frentiu, On program correctness and teaching programming, Computer Science Journal of Moldova, vol.5(1997), no.3, 250-260.

(Fre00) M.Frentiu, On programming style program correctness relation, Studia Univ. Babes-Bolyai, Seria Informatica, XLV(2000), no.2, 60-66.

(Fre01a) M.Frentiu, Verificarea Corectitudinii Programelor, Univ. "Petru-Maior", Tg. Mures, 2001, 116 pagini, ISBN 973-8084-32-6.

(Fre01b) M. Frentiu, and H.F.Pop, A Study of License Examination Results Using Fuzzy Clustering Techniques, Research Seminar on Computer Science, 2001, pp. 99-106.

(Fre02a) M. Frentiu, The Impact of Style on Program Comprehensibility, Proceedings of the Symposium Zilele Academice Clujene, 2002, pp. 7-12.

(Fre02b) M. Frentiu, H.F.Pop, A Study of Dependence of Software Attributes using Data Analysis Techniques, Studia Universitatis Babes-Bolyai, Informatica, 47(2),2002, 53-60.

(Fre03) M. Frentiu, On programming style, Babes-Bolyai University, Department of Computer Science, http://www.cs.ubbcluj.ro/∼mfrentiu/articole /style.html

(Fre04a) M. Frentiu, Program Correctness in Software Engineering Education, Proceedings of the International Conference on Computers and Communications ICCCM4, pp.154-157, Oradea, 2004, may 27-29.

(Fre04b) M. Frentiu, Formal Methods in Software Engineering Education, Proceedings ICELM1, Tg. Mures, 2004.

(Fre04c) Frentiu M., An Overview on Program Inspection, Proceedings of the Symposium “Zilele Academice Clujene”, 2004, pp. 9-14.

(Gib94) Gibs W.W., Softwares Chronic Crisis, Scientific American, september, 1994.

(Gil93) Tom Gilb and Dorothy Graham, Software Inspection, Addison-Wesley, 1993.

(Gri81) Gries D., The Science of Programming, Springer-Verlag, Berlin, 1981.

(Hoa69) Hoare C.A.R., An axiomatic approach to computer programming, Comm. A.C.M., 12 (1969), pp. 576-580.

(Hoa71) C.A.R.Hoare, Proof of a program: FIND, Comm.ACM, 14(1971), pp.39-45.

(Hoa91) Hoare C.A.R., The Mathematics of Programming, LNCS, 206, 1991, pp.1-18.

(Iga75) Igarashi S., London R.L., and Luckham D.C., Automatic Program Verification I: a logical basis and its implementation, Acta Informatica, 4(1975), 145-182.

(Jez97) Jezequel J.M., B.Meyer, Put it in the contract. The lessons of Ariane, Computer (IEEE), vol.30 (1997), no.2, p.129-130.

(Kat76) Katz, and Manna, Logical analysis of programs, Comm.ACM, 19(1976), 4, p. 188-206.

(Kin76) King J.C., Symbolic Execution and Program Testing, Comm. ACM, 19 (1976), 7, p.385-394.

(Led75) Ledgard H.F., Programming Proverbs for Fortran Programers, Hayden Book Company, Inc., New Jersey, 1975.

(Lev01) Leveson N.G., Systemic Factors in Software-Related Spacecraft Accidents, AIAA 2001, 47-63.

(Lon70a) London R.L., Proving Programs Correctness. Some Techniques and Examples, BIT, 10 (1970), pg.168-182.

(Lon70b) London R.L., Proof of Algorithms. A new kind of cerification, Comm. ACM, 13 (1970), pg.371-373.

(Mor90) Morgan, C., Programming from specifications, Springer, 1990.

(Mye78) Myers, A., A Controlled Experiment in Program Testing and Code Walk-throughs Inspection, Comm.A.C.M., 21(1978), no.9, pp.760-768.

(Nau66) Naur, Proof of Algorithms by general snapshots, BIT, 6(1966), pg.310-316.

(Nic04) V.Niculescu, M. Frentiu, Designing Correct Parallel Programs from Specifications, Proceedings of the 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004), Orlando, USA, July 18-21, 2004, pp.173-178.

(Pau93) Paulk M.C., B.Curtis, M.B.Chrissis, C.V.Weber, The Capability Maturity Model for Software, Tech.Report, CMU/SEI-93-TR-25, and IEEE Software, 10(1993,4, 18-27.

(Rob98) Robinson H., et all, Postmodern Software Development, The Computer Journal, 41(1998), 6, p.363-375.

(Sch90) Schach S.R., Software Engineering, IRWIN, Boston, 1990.

(Sta95) The Standish Group Report: Chaos, http://www.scs.carleton.ca/∼bean /PM/Standish-Report.html

(Ste91) Stevenson D.E., 1001 Reasons for not Proving Program Correct: A Survey, Philosophy and Computers, 1, 1991.

(You90) Yourdon, E., Modern SoftwareAnalysis, Yourdon Press, Prentice Hall Buiding, New Jersey 07632, 1989

(Weg74) Wegbreit, The synthesis of loop predicates, CACM, 17(1974), 102-112.

(Wir71) Wirth N., Program development by stepwise refinement, Comm. ACM 14(1971), 4, 221-227.

Babes-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science, Department of Computer Science, M. Kogalniceanu Str. 1, Cluj-Napoca, ROMANIA

E-mail address: mfrentiu@cs.ubbcluj.ro