Universal Journal of Computational Mathematics 1(3): 67-72, 2013 http://www.hrpub.org
DOI: 10.13189/ujcmj.2013.010301

Axiomatic Basis for Computer Programming

Lauretta O. Osho[1], Francisca Ogwueleka[2], Oluwafemi Osho[3,*]

[1]Department of Computer Science, Federal University of Technology, Minna, Nigeria
[2]Department of Computer Science, Federal University, Wukari, Nigeria
[3]Department of Cyber Security Science, Federal University of Technology, Minna, Nigeria
*Corresponding Author: femi.osho@futminna.edu.ng

Copyright © 2013 Horizon Research Publishing All rights reserved.

Abstract This paper considers a formal method, known as axiomatic semantics, used to prove the correctness of a computer program. This formal method extracts, using some proof rules, the mathematical verification conditions from a computer program. The axioms of program flow, including, sequential flow, iteration, and alternation flows are presented. Using the axiomatic basis the completeness of two variants of integer multiplication program is proved. Results show that computer programs can actually be verified sufficiently for correctness without necessarily testing them, or more practically put, to complement their testing.

Keywords Computer, Computer Programming, Axiomatic, Completeness, Correctness

1. Introduction

Computer programming is consisted in the designing, writing, testing, debugging, and maintaining of the source code of computer programs. It employs one or more programming languages. Programming occurs at the implementation phase of the software development life cycle (SDLC) [1].

Usually, the choice of a programming language is subject to, amongst other factors, the task for which the program is meant, and the efficiency with which the language helps programs written in it to execute. Different programming languages support different style of programming

Basically, there are fundamental properties that must be satisfied by every program. Some of these include reliability, which is a measure of the correctness of the program result; usability, depicting the ease of use of the program for its intended purpose; portability, the ability of a program to run on different platforms; maintainability, the ability of a program to be modified, upgraded, or improved upon [1]; and robustness, the ability of a program to withstand unusual conditions [2]; to mention but few. Reliability is closely tied to the correctness of the algorithm, and the amount of errors present in the program codes. Its validity relies on the variable values before the initiation of the program [3].

Hence, considering the preconditions and post conditions, one can prove the correctness of a program (or part of it). This system is known as axiomatic semantics [4]. This paper focuses on self-evident basis for proofs of some program properties.

1.1. Background of Study

In the software development lifecycle (SDLC), one of the most important stages is the test phase, as shown in Figure 1. At this phase, the software is verified to ascertain its capability to do what it is expected to do under all conditions. It is at this stage that design and implementation flaws are supposed to be detected [5]. Testing thus incorporates debugging, which is the process of finding and removing errors (bugs) in a program [6].

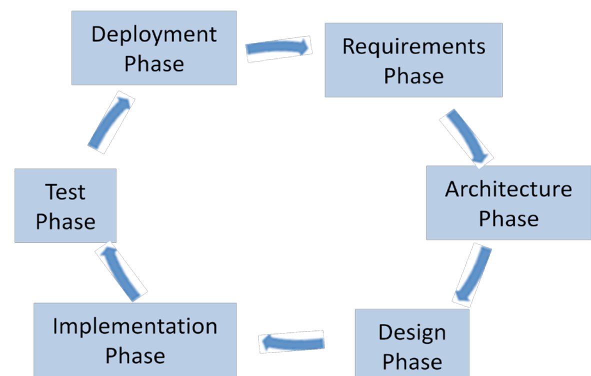


Figure 1. A general software development lifecycle [5]

The higher the algorithmic complexity of a program is, which invariably determines how complex a program is, the more likely there would be errors in the program. Program testing is often rigorous, especially for very large and complex programs. Except in small programs, testing cannot prove the correctness of a program. Testing only exposes errors in the codes. And in most cases, errors in a program never cease to be exhausted.

Consequently, the need for other complementary methods of achieving correctness of program cannot be overemphasi zed. These formal methods and the intuitive judgment of programmers can mutually support each other [7]

One of the formal methods used for proving program correctness is the axiomatic method [7]. The word axiomatic connotes self-evidence, obviousness. An axiomatic method has the potency of providing basis for measuring the quality of a programming language [7].

1.2. Statement of Problem

The conventional way of ascertaining the correctness of programs has always been through the testing for error detection and removal. The less the amount of error detected and successfully removed the more correct the program is deemed. However, for large and complex programs, this method has never proved adequate. In addition to the rigor usually associated with the exercise, it hardly exposes all the bugs (errors) in the program codes. Therefore, it is expedient to seek other methods of establishing program correctness.

A formal method of proving correctness of a computer program involves the use of axioms of discrete mathematics as basis for logical reasoning to extract from and prove the mathematical verification conditions of the program.

2. Review of Related Works

There have been considerable advances in the quantity and quality of methods deduced to prove the correctness of programs. While each had some strengths there were always limitations in their uses. For instance, though recursion induction, structural induction, and interpretation of flowcharts methods were applicable for proving correctness of programs that contained repetitions, the first two dealt with programs that achieved repetition by recursion, while the last one with programs that achieved by jumps and assignment [8]. Floyd's method has been used in proving correctness of programs in flow diagram form [9], Floyd-Hoare logic in proving imperative programs [10]. Burstall [9] extended Floyd's method to handle lists processing commands, while [11] used the inductive assertion method for multiprocess programs.

Some methods focused on the type of programs. Nakamura [12] proved the correctness of functional programs using Mizar, a popular proof checker. Basically, proof checkers contain library of mathematical models, lemmas and theorems. The checker works by relating the semantics of programs to the models. On the other hand, [10] considered modular functional programs.

Another method of classifying these methods is in whether they prove partial or total correctness of programs. Total correctness simply is consisted in partial correctness plus termination of loops [10,13,14]. Floyd-Naur's and structural induction methods, and Hoare logic are used for partial correctness, while Floyd's method, the extended Hoare logic is used for total correctness [14].

3. Mathematical Foundations

Axiomatic basis for information flow in computer programming was considered. The axioms (proof rules) for sequential program flow, considering the assignment statement; iteration program flow, considering the while…do and do…while statements; and the decision (alternation) flow, considering the **if…else** statement are presented.

3.1. Axiomatic Definitions for Integers

The integers are a system consisting of a set $\mathbb{Z}$ together with two symbols, '+' and '$\cdot$ ', denoted by ($\mathbb{Z}$, +, $\cdot$), which maps $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. They satisfy the following [15,16]:

1. $(x + y) + z = x + (y + z)$ (associativity of addition)
2. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (associativity of multiplication)
3. $x + y = y + x$ (commutativity of addition)
4. $x \cdot y = y \cdot x$ (commutativity of multiplication)
5. $\exists 0 \in \mathbb{Z}$ such that $0 + x = x = x + 0 \ \forall x \in \mathbb{Z}$ (additive identity element)
6. $\exists 1 \in \mathbb{Z}$ such that $1 \cdot x = x = x \cdot 1 \ \forall x \in \mathbb{Z}$ (multiplicative identity element)
7. $\forall x \exists - x \in \mathbb{Z}$ such that $x + (-x) = 0 = (-x) + x$ (additive inverse)
8. $x \cdot (y + z) = x \cdot y + x \cdot z$ (left distributive law)
9. $(x + y) \cdot z = x \cdot z + y \cdot z$ (right distributive law)

Further, there is an order relation $\leq$ on $\mathbb{Z}$, such that:

1. $x \leq x \ \forall x \in \mathbb{Z}$
2. If x $\leq y$ and y $\leq x$, then $x = y$
3. If $x \leq y$ and $y \leq z$, then $x \leq z$
4. $\forall x, y \in \mathbb{Z}$, either $x \leq y$ or $y \leq x$

3.2. Definition of Notations

i .$A\{B\}C$ is used to express relationship between a precondition (A), a program {B}, and a postcondition (C). It is interpreted "If the assertion A is true before initiation of a program B, then the assertion C will be true on its completion" [3].

ii .$A \vdash B$ means A yields B [1].

iii.$\frac{A_1,\ldots A_n}{B}$ implies that if logical statements A1,…An are true, then so is B [17].

3.3. Axiom of Assignment

Consider an assignment statement:

$$x = f$$

Where x is a variable identifier, and f is an expression of a programming language.

The precondition entails an assertion A(f), which must have been true of the expression f , before the assignment is made. With this in place, *A(x)* is said to be true.

This is expressed as:

$$\vdash A_0\{x = f\}A(x)$$

where,

$$A_0 = A(f)$$

3.4. Axiom for Iteration (while…do)

Consider the statement:
while P do Q
To deduce an axiom, suppose L is an assertion invariant over execution of P. P becomes false when the iteration terminates. Hence, the axiom for iteration is given as:

$$\frac{L \wedge P\{Q\}L}{L\{\textbf{while } P \textbf{ do } Q\} \neg P \wedge L} \quad (1)$$

L is called the loop invariant. It is an assertion that is usually true before the loop is initiated; it remains true after every execution of the loop; and also at the end of the loop.

3.5. Axiom for Iteration (do…while)

Consider the statement:
do Q while P

This is similar to the case in the **while**…**do** statement except that in this case the statement in the loop would have been executed first before the condition is tested. This means that if the condition is found to be false at the end of the first iteration, the embedded statement would have been executed once. This is given as:

$$\frac{L\{Q\}L}{L\{\textbf{do } Q \textbf{ while } P\} \neg P \wedge L} \quad (2)$$

3.6. Axiom for Alternation

Consider the statement:
if P then Q_1 else Q_2
Axiomatically, this is represented as:

$$\frac{L \wedge P\{Q_1\}B, L \wedge \neg P\{Q_2\}B}{L\{\textbf{if } P \textbf{ then } Q_1 \textbf{ else } Q_2\}B} \quad (3)$$

4. Prove of Correctness of Program for Integer Multiplication

Two different programs for integer multiplication are proved using some of the axioms for integers. The first one uses the method of repeated addition.

Although simple programs are used here, using axioms to prove correctness of programs, on a large scale, basically complements the intuitive judgment of programmers in program testing.

4.1. Program for Integer Multiplication by Repeated Addition

$$\{B \geq 0\}$$

$$x = A; y = B; p = 0;$$

while $y > 0$ do

$$p = p + x;$$

$$y = y - 1;$$

end while

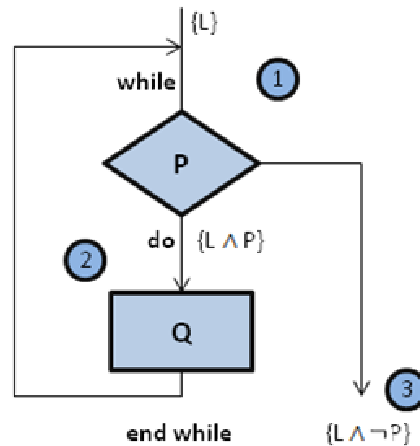
$$\{p = A \times B\}$$

4.1.1. Axiomatic Proof

Based on the axiomatic definition for iteration (also known as loop):

$$\frac{L \wedge P\{Q\}L}{L\{\textbf{while } P \textbf{ do } Q\} \neg P \wedge L}$$

We look for a loop invariant that is true before, within and outside the loop. This, when combined with the exit condition, produces the assertion that follow the loop (that is, the postcondition). The structure is illustrated in Figure 2 (redrawn from [18]):



1 **Initialization:** Show that loop invariant is initially valid

2 **Preservation:** Show that loop invariant remains valid after each execution of loop

3 **Completion:** Prove that loop invariant and exit condition imply the post condition

Figure 2. Structure of the while Rule

The loop invariant involves a relationship between unvarying variables within the loop. In a while condition, it also incorporates the loop condition, however, in a modified form. The modification is often necessary to accommodate the exit case of the loop [18].

For this problem, we can discover the loop invariant be examining how $A \times B$ is derived. If we let $A = 3$ and $B = 4$, we have:

p	x	y
0	3	4
3	3	3
6	3	2
9	3	1
12	3	0

We can deduce that $p + (x \times y) = A \times B$. Also, we modify our loop condition to accommodate the exit case, $= 0$. Hence, we have $y \geq 0$. We can therefore say that our loop invariant is:

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

To show that the loop invariant is initially true, we derive it from the initialization commands and the precondition.

Initialization:

$$\{B \geq 0\} \subset$$
$$\{A = A \ \wedge B = B \geq 0 \ \wedge 0 = 0\}$$
$$x = A; y = B; p = 0$$
$$\{x = A \ \wedge y = B \geq 0 \ \wedge p = 0\} \subset$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Preservation:

To prove that the loop invariant is preserved, we combine the loop invariant and the entry condition at the top of the loop, and from the bottom of the loop, we 'push back' invariant through the body of the loop. We have thus:

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge y > 0\} \supset$$
$$\{p + (x \times y) = A \times B \ \wedge y > 0\}$$
$$p = p + x; y = y - 1;$$
$$\{p + x + [x \times (y - 1)] = A \times B \ \wedge y - 1 > 0\} \subset$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Completion:

Finally, we must prove the assertion after the while loop (that is, the postcondition, $\{p = A \times B\}$) can be derived from $(\neg P \wedge L)$, where L is the loop invariant, and P, the while loop test condition:

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge \ \neg (y > 0)\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge \ y \leq 0\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \ \wedge \ y = 0\} \Rightarrow$$
$$\{p = A \times B \ \wedge \ y = 0\} \Rightarrow$$
$$\{p = A \times B\}$$

4.2. More Efficient Program for Integer Multiplication

$$\{m = A \wedge n = B \geq 0\}$$
$$x = m; y = n; p = 0;$$

While

$$y > 0$$

do

if $2 \times (y/2) \neq y$ then

$$p = p + x;$$

end if

$$x = 2 \times x;$$
$$y = y \ / \ 2;$$

end while

$$\{p = A \times B\}$$

Hint: consider the two cases where y is even $(y = 2k)$ and y is odd $(y = 2k + 1)$. Remember that / denotes integer division.

4.2.1. Axiomatic Proof

For this problem, to get the loop invariant, we also see how $A \times B$ is calculated, using small numbers. We consider both possible cases consequent upon the ***if*** statement. If we let $A = 3$ and $B = 4$, we have:

p	x	y
0	3	4
0	6	2
0	12	1
12	24	0

On the other hand, If we let $A = 4$ and $B = 3$, we have:

p	x	y
0	4	3
4	8	1
12	16	0

We can also deduce that $p + (x \times y) = A \times B$. Modifying our loop condition, we have $y \geq 0$. We discover in this case too that our loop invariant is:

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Initialization:

$$\{m = A \ \wedge \ n = B \geq 0\} \subset$$
$$\{m = A \ \wedge \ n = B \geq 0 \ \wedge 0 = 0\}$$
$$x = m; y = n; p = 0;$$
$$\{x = A \ \wedge y = B \geq 0 \ \wedge p = 0\} \subset$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Preservation:

Case 1: y is even. That is, $y = 2i > 0$ for some integer $i \geq 0$.

Consequently, the ***if*** condition is false. Hence,

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge y > 0\} \supset$$
$$\{p + (x \times y) = A \times B \wedge y > 0\}$$
$$x = 2 \times x; y = y/2;$$
$$\{p + \left(2 \times x \times \frac{y}{2}\right) = A \times B \ \wedge (\frac{y}{2}) > 0\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \wedge y > 0\} \subset$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Case 2: y is odd. That is, $y = 2i + 1 > 0$ for some integer $i \geq 0$.

Consequently, the ***if*** condition is true, that is, $(2 \times y/2) \neq$

y. Hence,

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge y > 0\} \supset$$
$$\{p + (x \times y) = A \times B \wedge y > 0\}$$
$$p = p + x; x = 2 \times x; y = y/2\,;$$
$$\{p + x + \left(2 \times x \times \frac{y}{2}\right) = A \times B \ \wedge (\frac{y}{2}) > 0\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \wedge y > 0\} \subset$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0\}$$

Completion:
We prove that $p = A \times B$ is derived from $(\neg P \wedge L)$

$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge \ \neg (y > 0)\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \ \wedge y \geq 0 \ \wedge \ y \leq 0\} \Rightarrow$$
$$\{p + (x \times y) = A \times B \ \wedge \ y = 0\} \Rightarrow$$
$$\{p = A \times B \ \wedge \ y = 0\} \Rightarrow$$
$$\{p = A \times B\}$$

5. Summary and Conclusion

5.1. Summary

In this paper, some axioms of program properties, including, sequential flow, iteration (for while…do and do…while control statements), and alternation flows were stated. And using the axioms as basis the completeness of two variants of integer multiplication program was proved.

5.2. Conclusion

The use of axioms and rules of inference to prove the correctness of a program (or part of it), are meant to complement the intuitive judgment of programmers. As a matter of fact, both are best utilized when they mutually support each other.

Today, there are other methods for establishing program correctness. These include new branches of applied discrete mathematics to formalize programming concepts, and different forms of logic to simplify reasoning. For corecursive programs, they can be proved using fixpoint induction, coinduction, approximation lemma, or fusion methods [19]. As a matter of fact, there are currently computer applications used for proving correctness of programs. These include ESC/Java [20], SPARK, eCv (MISRA-C), Spec# [21], and Dafny [22].

This paper primarily focuses on the axiomatic method of proving correctness. Further works may include using the same method to prove more complex programs, or using other methods to verify the correctness of the two integer multiplication programs presented.

Appendix – List of Symbols

Symbol	Meaning/Definition
$\mathbb{Z}$	Integer
$\exists$	There exist
$\in$	Member of
$\forall$	For all
$\vdash$	Proves (implies, yields)
$\neg$	Not (negation)
$<$	Less than
$\leq$	Less than or equal to
$\geq$	Greater than or equal to
$\wedge$	Logical and
$\supset$	Superset of (includes in set)
$\subset$	Subset of
$\Rightarrow$	Implies

REFERENCES

[1] http://en.wikipedia.org/wiki/Computer_Programming

[2] The Linux Information Project (2005). Retrieved from: http://www.linfo.org/robust.html

[3] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM, Vol. 12, No. 10. pp. 576 – 583.

[4] Lusth, J. C. (2011). Programming Languages: Axiomatic Semantics. Retrieved from http://rra.cs.ua.edu/proglan/axiom2.html

[5] Owens, D. Integrating Software Security into the Software Development Life Cycle. System Securities, San Diego, CA 62123, USA.

[6] Program Testing. Retrieved from: http://www.cse.unsw.edu.au/~billw/testing.html

[7] Hoare, C. A. R. (2009). Retrospective: An Axiomatic Basis for Computer Programming. Communications of the ACM, Vol. 52, No. 10. pp. 30 – 32.

[8] Burstall, R. M. (1968). Proving Properties of Programs by Structural Induction.

[9] Burstall, R. M. Some Techniques for Proving Correctness of Programs which Alter Data Structures. University of Edinburgh.

[10] Owens, C. A. (1999). Proving Correctness of Modular Functional Programs. University of Edinburgh.

[11] Lamport, L. (1977). Proving the Correctness of Multiprocess Programs, IEEE Transactions on Software Engineering, Vol. SE-3, No. 2.

[12] Nakamura, Y. (2007). Proving the Correctness of Functional

Programs Using Mizar. Studies in Logic, Grammar, and Rhetoric, 10 (23).

[13] Capretta, V. G52DOA – Total Correctness. Available on: http://www.cs.nott.ac.uk/~vxc/g52doa /total_correctness.pdf

[14] Cousot, P. Methods and Logics for Proving Program. In ``Handbook of Theoretical Computer Science'', J. van Leeuwen (Ed.), vol. B `` Formal Models and Semantics'', Ch. 15, pp. 843--993, Elsevier, 1990.

[15] Greenleaf, F.P. (2000-2008). Algebra I (Section 2: The System of Integers). Retrieved from www.cims.nyu.edu/~naor/homepage%20files/integers.pdf

[16] Thunder, J. (2009). Axioms for Integers. Retrieved from http://www.math.niu.edu/~jthunder/Courses /2009Fall/420/sec1/aug24/intaxioms.pdf

[17] Andrews, G. R., and Reitman, R. P. (1980). An Axiomatic Approach to Information Flow in Programs. ACM Transactions on Programming Languages and Systems, Vol. 2, No. 1. Pages 56-76

[18] Slonneger, K. and Kurtz, B. L. (1995). Formal Syntax and Semantics of Programming Languages: a Laboratory Based Approach (Chapter 11: Axomatic Semantics). Addison Wesley Publishing Company, Inc.

[19] Gibbons, J. and Hutton, G. (2005). Proof Methods for Corecursive Programs, Fundamenta Informaticae XX (2005) 1–14, IOS Press.

[20] Ouimet, M. Formal Software Verification: Model Checking and Theorem Proving. Retrieved from: http://webdocs.cs.ualberta.ca/~piotr/Courses/662/Reading/ESL-TIK-00214.pdf

[21] Ireland, A. Rigorous Methods for Software Engineering Program Specification. Retrieved from: http://www.macs.hw.ac.uk/~air/rmse/lectures/lec-6-prog-spec.pdf

[22] Leino, M. Dafny: An Automatic Program Verifier for Functional Correctness. Retrieved from: http://research.microsoft.com/en-us/um/people/leino/papers/krml203.pdf