Recent Advances in Program Correctness Verification

Colin James III, CEC Services, LLC, pcv-demo@cec-services.com, see 4-VL.com.

Introduction

The purpose of this paper is to describe the theory and implementation of software program correctness verification (PCV). There are two sections: what is tested; and how it is tested.

Validation and Verification are named collectively as "V&V". In order to meet requirements, validation tests if the correct thing is built, and verification tests if the thing built is built correctly[1]. Correctness means the logical, mathematical proof that a software component is built correctly. The mechanism for proof is four-valued bit code (4vbc)[2]. This is defined as four atomic elements of dibits: {00} not bivalent; {01} true; {10} false; and {11} bivalent. The left right sides of the the dibits are additional variables as false and true sides {F|T}. True {01} really means {0|1}, where {0|} means the switch on the false side is off, not false, and {|1} means the switch on the true side is on, true. In other words, {01} means {not false | true}. Similarly, {00} means {not false, not true}, "not false AND not true", which is impossible and a contradiction. {11} means {false, true}, "false OR true", which is a tautology and the basis for proving axioms and theorems in formal logics[3]. From these atomic dibits, pairs of dibits as 4-bits are derived to describe as true, false, or meaningless for:

The picture of the reality of software:		The picture of the non reality of software:	
{10 01}	{00 01}	{10 00}	{00 00}
conditionally true	necessarily	not permissible	contradiction
{01 10}	{11 10}	{01 11}	{11 00}
conditionally false	not necessarily	permissible	not optional
{01 01}	{11 01}	{01 00}	{00 11}
logically true	possibly	ought to be case	optional
{10 10}	{00 10}	{10 11}	{11 11}
logically false	not possibly	not ought to be case	tautology

[1] Carnegie Mellon Software Engineering Institute (SEI) classifies Verification in the Capability Maturity Model Integration under product development as CMMI-DEV at Level 3 of 3.

[2] See 4vbc.com and 4-VL.com

[3] See references [1] and [2] below.

The word labels become the descriptions of the relative and cumulative correctness of the software components to be verified.

What is tested

TrueBASIC[4] is a portable educational language chosen here to explain PCV. In TrueBASIC, software programming may be decomposed into three types of structures as loop, branch, and reuse. The loop forms are DO-LOOP, DO-LOOP-UNTIL, DO-LOOP-WHLE, DO-UNTIL-LOOP, DO-WHILE-LOOP, and FOR-NEXT. The branch forms are IF-THEN-END-IF, IF-THEN-ELSEIF-END-IF, and SELECT-CASE-END-SELECT. (The SELECT form is not evaluated here because it can be implemented more clearly in the IF-THEN-END-IF form.) The reuse form to encapsulate a subroutine is SUB-END-SUB. While this is a form of flow control invoked by CALL, it is arguably not the branch form of IF-THEN that is based on a test.

The loop form has an iterator that is checked against a sentinel limiter at the beginning (top) or the end (bottom) of the loop. The FOR-NEXT is checked at the top and automatically iterates. Therefore the developer is relieved of manually incrementing the test counter in the automatic "i = 1 TO 10". However, this may be mixed blessing because the manual control of the iteration forces the developer to pay closer attention to exactly how the loop advances. The advantage of a DO-WHILE-LOOP comes when manually incrementing the iterator directly above the bottom line of the loop. This is because to end the loop prematurely, or short circuit it, the iterator can then clearly be set equal to the loop limiter from within an IF-THEN test block. This avoids the vagaries of the arbitrary EXIT-DO or EXIT-FOR short circuit statements so as to impose a clear ending should an early exit strategy from the loop be required.

The preferred DO-WHILE-LOOP has additional lines of code before the block to prepare the value of the limiter and the initial iterator. The formula after the WHILE clause is a subtraction test to zero in the syntax of "sentinel – iterator = 0" because of an advantage. Most hardware processors have arithmetic controllers that decrement slightly faster than they increment because of fewer assembly language instructions and machine code cycles. Hence the operation of subtraction is preferred. The further preferred syntax is "NOT(sentinel – iterator = 0)" because of another advantage. The NOT operator is in the same class of fast, primary operators such as subtraction. The NOT operator also takes precedence over the slower, secondary operators of ">" greater than and "<" less

---

[4] This note is about programming style that is best by test to improve source code readability for others. In TrueBASIC, each line begins with a standard keyword, such as the assignment statement of LET. As a convention, library commands have a leading upper case letter. Variable names are explicitly not in Hungarian notation as "HungarianNotationVariable" but instead use name blocks shortened into three letter blocks and separated by the underscore character "_" such as "hun_nte_var". The use of parenthesis and mathematical operators are accentuated with a leading space where "SQR(b^2-2*a*c)" is written as "SQR((b ^ 2) – (2 * a * c)) with no trailing spaces.

than. Hence there are fewer machine cycles for "IF NOT(sentinel – iterator = 0) THEN" than for "IF (sentinel – iterator >= 0) THEN". A secondary advantage supports the readability of source code. It is easy for the eye consistently to find and read the WHILE test in the preferred format, and to distinguish by the absence of NOT from other tests in that syntax[5]. The disadvantage to the DO-WHILE-LOOP is that it takes longer to implement.

The structure of branching is meant to simplify rather than confuse. To do that requires that flow control make no assumptions, implements requirements by explicit test of every logical test case. Here the IF-(true state)-THEN is incomplete. However, it is complete if accompanied by the IF-NOT(true state)-THEN form. The advantage of this approach is to provide complete logical coverage that affords clearer visual control.

Complex branching structures are recomposed from unnested IF's into nested IF's as follows.

Unnested IF's (3):

```
IF tru_001 THEN
END IF

IF tru_002 THEN
END IF

IF tru_002 THEN
      END IF
```

Nested IF's (6):

```
IF      ( tru_001) THEN
END IF
IF NOT( tru_001) THEN
        IF      (tru_002) THEN
        END IF
                IF NOT( tru_002) THEN
                IF      ( tru_003) THEN
                        END IF
                        IF NOT( tru_003) THEN
                        END IF
                END IF
        END IF
```

The three unnested IF's may appear separately in any order and with the same result. The unnested IF's obtain comprehensive test coverage only when the NOT of their respective tests is also evaluated. The nested format accommodates the evaluation of all possible test cases. The nested format also provides a mechanism to specify the precedence of one test over another based on the practical frequency of the test. For example, if the test NOT(tru_003) is logically visited least often, then it is appropriate to place that test most deeply in the nest. The method of placing the test based on how often its code is reached is named stacking. The method of nesting the tests to ensure complete case coverage is named packing. The entire technique is named "stack and pack". The

[5] As a programming side note, implementation of interlocking loops in nested DO-WHILE loops clearly separates the iteraters and makes obvious how the values of the iteraters relate.

disadvantage of the stack and pack IF blocks is that it requires two times more IF blocks to implement than do the unnested IF blocks.

The SUB-END-SUB structure is its own straightforward form. The short circuit statement of EXIT-SUB is avoided by using IF-THEN blocks. To recap, the fundamental programming structures considered here for verification of correctness are DO-WHILE-LOOP, IF-THEN-END-IF, and SUB-END-SUB.

How it is tested

The software blocks above share common features at run time. They may exist or not exist in the test program. They may have or not have entry accessibility to their code. They may contain code that is executable or may not contain code as a null stub. They may raise or not raise exceptions such as errors. These test conditions are respectively named Exist, Enter, Execute, and Exception (or Error) and are collectively named "The Four E's".

The mandatory structure of the input test code is encapsulated as a subroutine in the form SUB-END-SUB. It is then invoked from a CALL located at the program level of mainline processing.

The test codes is parsed for the blocks DO-WHILE-LOOP, IF-THEN-END-IF, and SUB-END-SUB. Test directives are embedded into the test code before and after the lines of DO-WHILE and IF-THEN, and after the line of SUB. The test directives have the arbitrary syntax of "CALL Test_ …" and "SUB Test_ …". The test code is rewritten to include these test directives and reparsed. Keywords that are deemed illegal by the parser are "EXIT" and "STOP" which cause the PCV program to terminate. The PVC program then determines what block forms exist within the test code. The test code is executed from within the PVC program which acts as a real time program monitor. When the test code is executed, its test directives write flags for the entry accessibility of each block visited in real time. The PVC program evaluates each block for the presence of executable code. If a block does not have entry accessibility, then the content of the block is evaluated anyway for the presence of code. This is because the inaccessibility of code within a block cannot necessarily exclude that code from being evaluated for correctness. If the content of the block contains executable code, then the PCV program executes that code segment in real time and notes exceptions raised. If the content of the block contains no executable code, then the block is flagged as not executable, and the error result is noted as unknown.

In the case of the SUB-END-SUB block, if there is executable code present then the block is entered and the code is executed. The error result is that of either "no error present" or "no error not present". However, if the code within the SUB block contains a CALL to another subroutine, other than to embedded test directives, then the presence of that object CALL evaluates the contents of the SUB block as unknown as "no error present or not present". This is because the correctness result for the target subroutine is

not necessarily visible since that result is tabulated independently from the code block containing the object call. If the block is not entered then the contents of the block cannot be executed. In this case the error result is that of "no error present or not present". The output results produce an 8-bit number that is blocked in dibits in the format of abcd_efgh.

Exist (ab)	= XXcd_efgh:	not present	10cd_efgh	128;	present	10cd_efgh	64
Entry(cd)	= abXX_efgh:	not present	ab10_efgh	32;	present	ab01_efgh	16
Execute(ef)	= abcd_XXgh:	not present	abcd_10gh	8;	present	abcd_01gh	4
No Error(gh)	= abcd_efXX:	not present	abcd_ef10	2;	present	abcd_ef01	1
		unknown	abcd_ef11	3			

Each block tested is assigned an 8-bit correctness code. If the code is an even number, then a run time exception was raised, making the block ultimately incorrect. If the code is an odd number, then the block has no run time exceptions, but may have a degree of incorrectness due to no entry accessibility of that block meaning the block is potentially dead code. The correctness code for each block and its preceding blocks may be compiled into a running accumulation of correctness. The intermediate block values are compiled using the logical AND operator modulo 256 (modulo 255 + 1). Because odd number multiplied by odd numbers produce odd numbers, and even modulo greater than the largest odd number, in this case 255, assures that a modulo result of zero may not become an explosive annihilator as the multiplicand.

Conclusion

Program Correctness Verification (PCV) is advanced by its implementation in four-valued bit code (4vbc). The PCV as described and implemented in TrueBASIC in this paper is also rapidly extendable to Ada2005, C++, Cobol, FORTH, Fortran, Java, and Python. The advantage of PCV is that there is now a fully automated and mechanical method to prove mathematically the correctness of software. Hence PCV may save the resources of large consumers of requirement built software, such as the Department of Defense, during the final verification phase.

References

[1] Goodwin, Garry; James III, Colin. ""Logical Foundations of Four Valued Bit Code (4VBC)", 8th International Workshop on Boolean Problems, Freiberg (Sachsen), 2008, 239-250.

[2] James III, Colin. "How to Map Software Loops and Flows into dibits of Four-Valued Bit Code", 18th International Workshop on Post-Binary ULSI Systems, Naha, Okinawa, Japan, 2009, 42-49.