Program Correctness Verification (PCV)

See 4-VL.com, CEC-Services.com; info@4-VL.com

PCV is a fully automated and mechanical method to prove mathematically the correctness of software source code by tabular lookup.

Lookup tables (LUTs) are implemented with four-valued bit code (4vbc).

Program Correctness Verification

- Quality = Meeting requirements
- V & V = Validation and Verification
- Validation = Correct requirement is fulfilled
- Verification = Requirement fulfilled correctly
- Correctness = Proof of fulfillment without error
- Exception = Condition raised or caused by errors

Proof via multivalued logic (MVL)

- Logical dibits = Bivalent, four valued-bit code (4vbc);
  { 00}, { 01}, { 10}, { 11}
- Dibit side = ‹ Left | Right›, ‹Sinistro | Dextro›
- Dibit meaning = ‹ False | True›, ‹ 0 | 1›; True
  ‹ True | False›, ‹ 1 | 0›; False
  ‹ False | False›, ‹ 0 | 0›; Contradiction
  ‹ True | True›, ‹ 1 | 1›; Tautology

Semantic and syntactic meanings

- Semantic values
  = { 10}; Not({ 10})
- Semantic meanings
  = { 10} False; { 01} True
- Syntactic values
  = { 10 + 01}; Not({ 10 + 01})
- Syntactic meanings
  = { 11} Tautology False OR True;
  { 00} Contradiction Not False AND Not True

The picture of software is based on

- **Reality**

Conditionally true	{10 01}
Conditionally false	{01 10}
Logically true	{01 01}
Logically false	{10 10}
Necessarily	{00 01}
Not necessarily	{11 10}
Possibly	{11 01}
Not possibly	{00 10}

- **Non reality**

Permissible	{01 00}
Not permissible	{10 11}
Ought to be case	{01 00}
Not ought tb case	{10 11}
Optional	{11 00}
Not optional	{00 11}
Tautology	{11 11}
Contradiction	{00 00}

To test blocks of statement code (BoC)

Allowed

Looping

- DO-LOOP
- FOR-NEXT

Branching

- IF-THEN-END-IF

Calling

- SUB-END-SUB

Not Allowed

Looping

- EXIT-DO
- EXIT-FOR

Branching

- EXIT
- THEN-ELSEIF
- SELECT-CASE

Calling

- EXIT-SUB

DO-WHILE-LOOP vs FOR-NEXT

```
LET sentinel    = x
LET iterator    = y
LET scalar      = z

DO WHILE sentinel – iterator >= 0

    . . . .

    LET iterator = iterator + scalar

LOOP
```

More test control on the variables means more code to program and slower implementation.

```
LET sentinel    = x
LET fiducial    = y
LET scalar      = z

FOR iterator = fiducial TO
                    sentinel STEP scalar

    . . . .

NEXT iterator
```

Less test control on the variables means less code to program and faster implementation.

IF-THEN-END-IF: Unnested vs Nested

Unnested

```
IF tru_001 THEN
END IF

IF tru_002 THEN
END IF
```

The unnested format evaluates in any order for the same result. Short circuit is not controlled and implicit to the fall through of the code.

Nested

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

The nested format evaluates in a set order to obtain test coverage of all possible cases. Short circuit is controlled and explicit to the fall through of the code.

Blocks of code are tested by the 4-E's.

- **Exist**

  The BoC exists and is present even as a null stub.

- **Entry**

  The BoC is accessible with entry and exit points.

- **Execute**

  The BoC executes in real time via a monitor.

- **Error**

  The BoC executes without or with an exception.

Dibits map 4-E's in 8-bits of *abcd_efgh*.

- **Exist** (ab = 128, 64)

  10cd_efgh: not present; 01cd_efgh: present

- **Entry** (cd = 32, 16)

  ab10_efgh: not present; ab01_efgh: present

- **Execute** (ef = 8, 4)

  abcd_10gh: not present; abcd_01gh: present

- **No Error** (gh = 2, 1, 3)

  abcd_ef10: not present; abcd_ef01: present

  abcd_ef11: unknown

How the 4-E codes build a test result.

- Each BoC test results in an 8-bit correctness code that accumulates by applying the logical connective AND (multiply) to the four constituent codes:

  [(XXcd_efgh) AND (abXX_efgh) AND (abcd_XXgh) AND (abcd_efXX)] modulo 256 = correctness code

- A correctness code of an odd number means the BoC raised no exceptions at execution time, but the BoC may have a degree of incorrectness from no entry point, ie, it may be dead code not accessed in the program flow.

Is correctness code a theorem or axiom?

- The correctness code is checked for being an axiom or theorem by its 8-bit look up table (LUT).
- A theorem is of the form *wxyz_wxyz* where the groups of 4-bits (tetra-bits) are the same.

  LUT 119 is “0111 0111” and the connective “OR”
- An axiom is of the form *jklm_pqrs* where the groups of 4-bits are not the same. (Modal logic □ means necessarily.)

  LUT 6 is “0000 0110” and the connective “□OR”
- This part of the PCV process is a tableau method.

Abstracting PCV into Design Correctness Verification (DCV)

- Design Specification Languages (DSL) describe abstractions, such as to map *supposed equivalents*:
  - Every student has rented a car from Avis.
  - There is at least one car Avis has rented to every student.
- The equivalent above in polyadic predicate logic is:
  - $\forall x(Sx \rightarrow \exists y(Cy \,\&\, Rayx)); \exists x(Cx \,\&\, \forall y(Sy \rightarrow Raxy))$
- PETE (Polyadic Expander, Tokenizer, Evaluator), based on PCV, finds 8-bit results for the above to be:
  - { 01 00 11 **1**0} and { 01 00 11 **0**0}, ie, *not the same*

Potential military applications for DCV

- As based on PCV, the abstraction of DCV serves as the seminal SEIR product
- SEIR means Strategic Electronic Inquiry Ratifier to:

  Translate a question in a foreign language (exists at NASA)

  Speak and display the question (exists at NASA and NEC)

  Express the question in symbolic polyadic predicate logic

  Expand, tokenize, and evaluate the expression in PETE

  Perform the above steps on a foreign answer

  Combine the Q and A expressions in an 8-bit truth value