Mechanical Verification of Total Correctness Through Diversion Verification Conditions

by

Peter Vincent Homeier

Department of Computer and Information Science
University of Pennsylvania
homeier@saul.cis.upenn.edu

and

David F. Martin

Computer Science Department
University of California at Los Angeles
dmartin@cs.ucla.edu

"It is clear that working out the details of this would be a lot of work."

— Michael Gordon, 1988

Problem Statement and Thesis

- ***Software today is notoriously subject to error***

  Current practices employ repeated testing to achieve quality.

  "Program testing can be used to show the presence of bugs, but never to show their absence!" — Dijkstra, 1972

  If a program is partially correct but not proven to terminate, then its correct answers may not ever be provided.

- ***Program proofs can greatly increase reliability, but are difficult***

  Program proofs verify a program's correctness once, that it satisfies its specification, instead of relying on repeated and incomplete testing.

  But proofs are complex and difficult to construct, esp. for termination.

- ***Verification Condition Generators partially automate the proof's creation***

  This significantly reduces the complexity and detail required from the user, making the proofs more *effective*.

- ***But VCGs have generally not themselves been proven sound***

  So the credibility of the proof rests on the credibility of the tool, which has not been verified.

- ***THESIS:*** ***Reliable partial automation of proofs can be achieved by verifying the Verification Condition Generator.***

Contributions

- ***A Mechanically Verified Verification Condition Generator***

  — for a small imperative programming language, including procedures and expressions with side effects

  — for total correctness (partial correctness + termination)

  — the proof of the VCG verification is conducted within and checked by the HOL mechanical theorem proof checker

  — includes a prototype VCG implementation within HOL

- ***A new method for proving the termination of mutually recursive procedures***

  — our main theoretical contribution

  — verified by analysis of *procedure call graph* structure, unlike all previous VCG work, which was directed by *syntactic* structure
    - Proof of progress achieved across a single call
    - Proof of recursive progress across multiple calls
    - Proof of termination

- ***New program logics have been invented***

  Five program logics are presented, with 14 correctness specifications

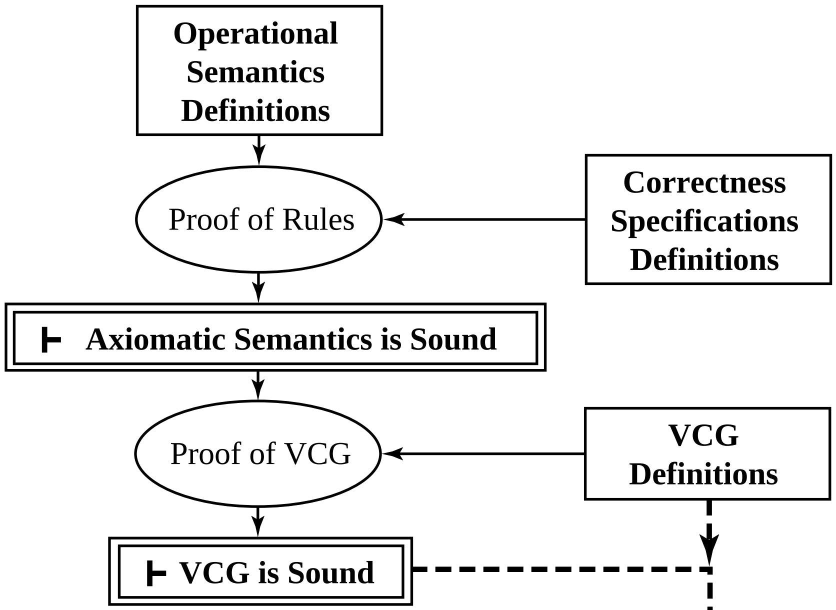
  Three logics are new, addressing
    - total correctness of expressions
    - progress up to procedure entrance
    - termination, conditional and unconditional

  All of these logics were *mechanically proven sound* within HOL.

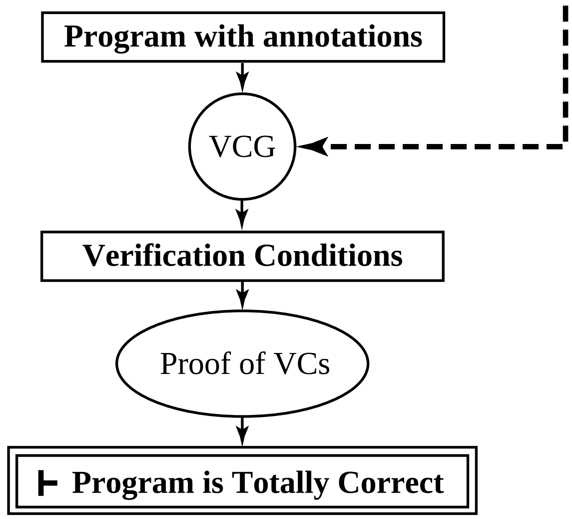
■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Overview of Approach

Verification of VCG:



Use of VCG:



Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Syntax of Sunrise Programming Language

The syntax is represented in HOL logic by new concrete recursive types.

`exp:`	e	::=	$n \mid x \mid {++}x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$
`exp list:`	es	::=	$\langle\rangle \mid CONS\ e\ es$
`bexp:`	b	::=	$e_1 = e_2 \mid e_1 < e_2 \mid es_1 << es_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid {\sim} b$
`cmd:`	c	::=	**skip** $\mid$ **abort** $\mid$ $x := e \mid$ $c_1 ; c_2 \mid$ **if** b **then** c_1 **else** c_2 **fi** $\mid$ **assert** a **with** a_{pr} **while** b **do** c **od** $\mid$ $p\ (x_1, \ldots, x_n ; e_1, \ldots, e_m)$
`decl:`	d	::=	**procedure** p (**var** $x_1, \ldots, x_n$; **val** $y_1, \ldots, y_m$); **global** $z_1, \ldots, z_k$; **pre** a_{pre}; **post** a_{post}; **enters** p_1 **with** a_1; … **enters** p_j **with** a_j; **recurses with** a_{rec}; c **end procedure** $\mid$ $d_1 ; d_2 \mid$ **empty**
`prog:`	π	::=	**program** $d ; c$ **end program**

Table 1. Sunrise Programming Language Syntax

Syntax of Sunrise Assertion Language

The syntax is represented in HOL logic by new concrete recursive types.

`vexp:`	v	::=	$n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$
`vexp list:`	vs	::=	$\langle\rangle \mid CONS\ v\ vs$
`aexp:`	a	::=	**true** | **false** | $v_1 = v_2 \mid v_1 < v_2 \mid vs_1 << vs_2 \mid$ $a_1 \wedge a_2 \mid a_1 \vee a_2 \mid \sim a \mid$ $a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid (a_1 => a_2 \mid a_3) \mid$ **close** $a \mid \forall x.\ a \mid \exists x.\ a$

Table 2. Assertion Language Syntax

- $vs_1 << vs_2$ is lexicographical less than.
- $(a_1 => a_2 \mid a_3)$ is a conditional expression.
- The **close** operator forms the universal closure of an expression, with the effect of universally quantifying all free variables.

These two languages, though related, are distinct, with translation functions *VE* : `exp->vexp` and *AB* : `bexp->aexp`.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

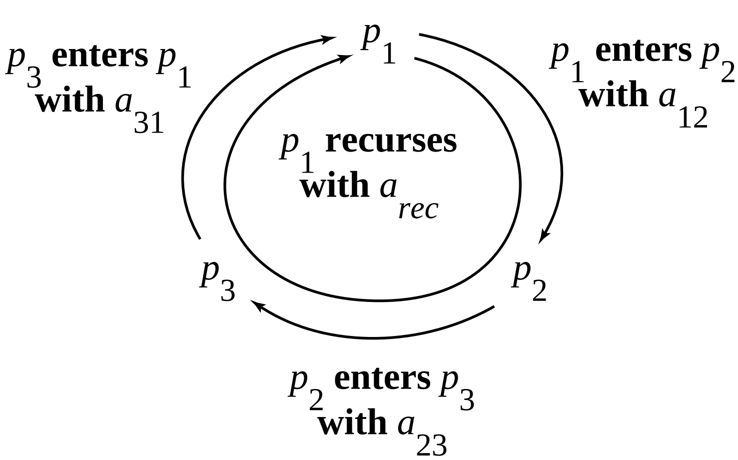
Required Annotations

- Annotations are a pragmatic necessity for simple VCGs
- **while** loops require two annotations:

  assert a **with** a_{pr} **while** b **do** c **od** — invariant, progress expression

- Procedure declarations require five kinds:

  procedure p (**var** $x_1, \ldots, x_n$; **val** $y_1, \ldots, y_m$);
      global $z_1, \ldots, z_k$; — global variables
      pre a_{pre}; — precondition
      post a_{post}; — postcondition
      enters p_1 **with** a_1; — entrance progress expression a_i
          … for every procedure p_i called in c
      enters p_j **with** a_j;
      recurses with a_{rec}; — recursion progress expression
      c
  end procedure



- The recursion progress expression $a_{rec} = (v < x)$ specifies the progress expected between recursive calls, that v strictly decreases.
- The sum of the progress a_{12}+a_{23}+a_{31} must imply the progress of a_{rec}.

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Bicycling Example

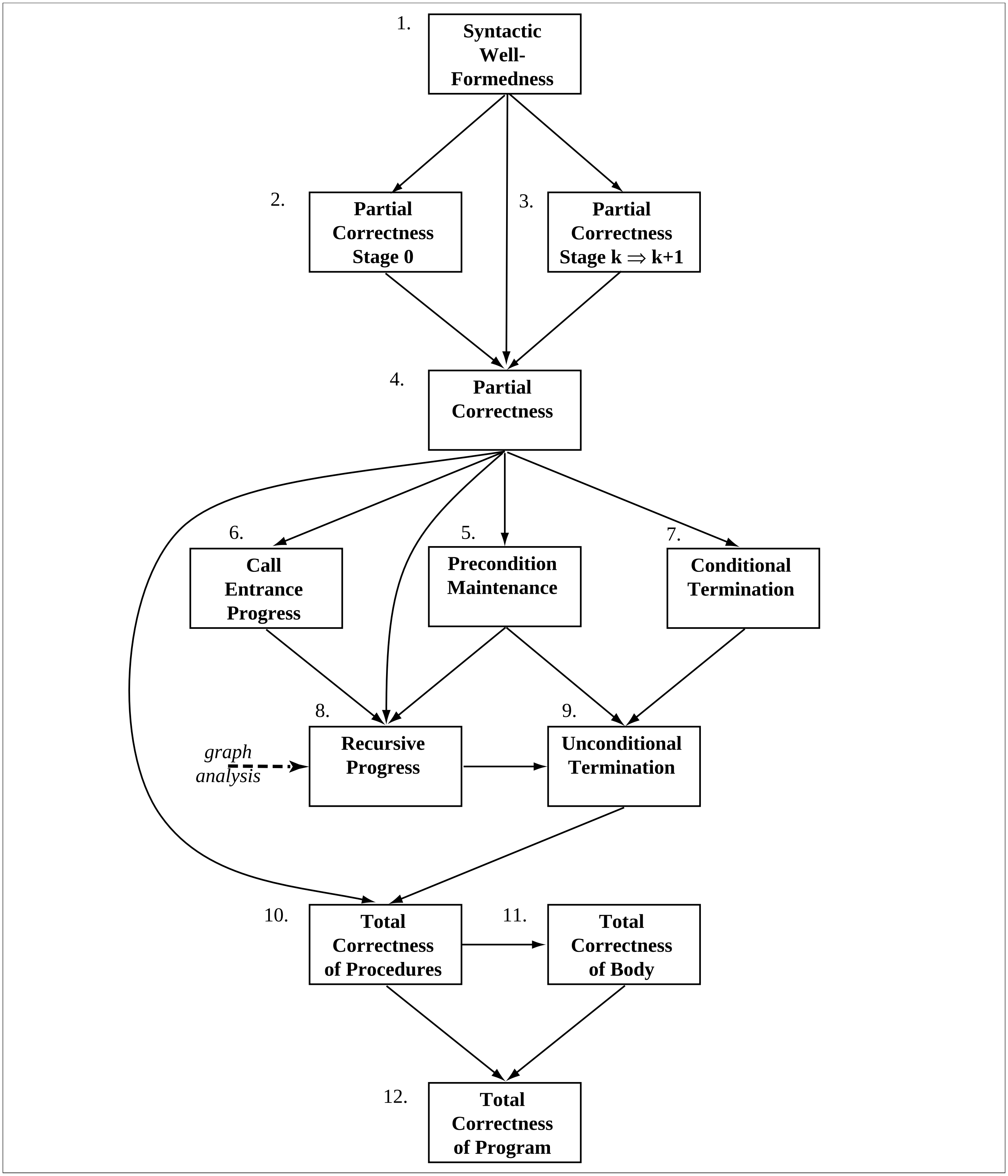
```
program
    procedure pedal(; val n, m);
        global  a, b, c;
        pre     n * m + c  =  a * b;
        post    c  =  a * b;
        enters pedal with  n < 'n  ∧  m = 'm;
        enters coast with  n < 'n  ∧  m < 'm;
        recurses with  n < 'n;

        if  n = 0  ∨  m = 0  then
            skip
        else
            c := c + m;
            if  n < m  then
                coast(; n – 1, m – 1)
            else
                pedal(; n – 1, m)
            fi
        fi
    end procedure;

    procedure coast(; val n, m);
        global  a, b, c;
        pre     n * (m + 1) + c  =  a * b;
        post    c  =  a * b;
        enters pedal with  n = 'n  ∧  m = 'm;
        enters coast with  n = 'n  ∧  m < 'm;
        recurses with  m < 'm;

        c := c + n;
        if  n < m  then
            coast(; n, m – 1)
        else
            pedal(; n, m)
        fi
    end procedure;

    a := 7;  b := 12;  c := 0;
    pedal(; a, b)
end program
[ c = 7 * 12 ]
```



Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Scale of VCG Logical Structure in HOL

- ***Definition and Verification of Verification Condition Generator***

  — 57,000+ lines of proof code

  — 8 new types

  — 217 definitions of new functions and constants

  — 906 major theorems proved

  — 22 new HOL theories constructed

  — largest logical structure among contributed libraries

- ***Secure Application of VCG to Examples***

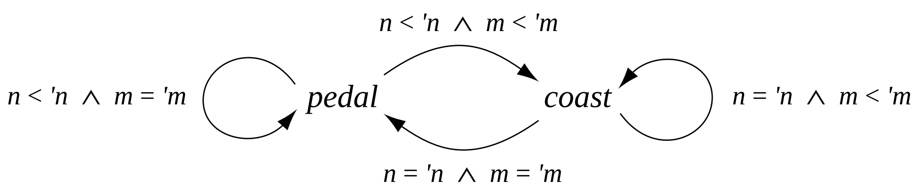
  — 10,000 lines of code for VCG tactics, parser, prettyprinter

  — 7 examples; largest is
    - 4 procedures
    - 68 lines of code
    - Generates 13 verification conditions.

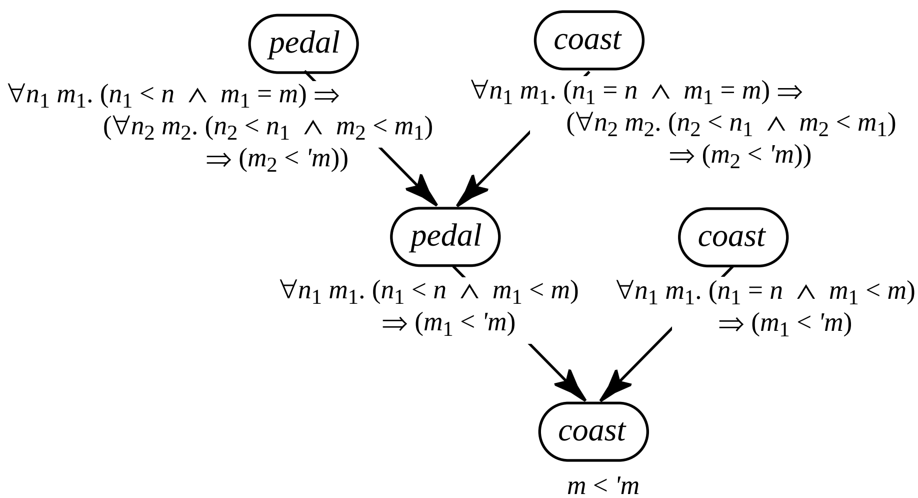
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

VCG Analysis of Call Graph Structure

The call graph structure of the Bicycling example:



Taking *coast* as the root node and exploring backwards through the call graph, we generate the call graph tree for *coast*:

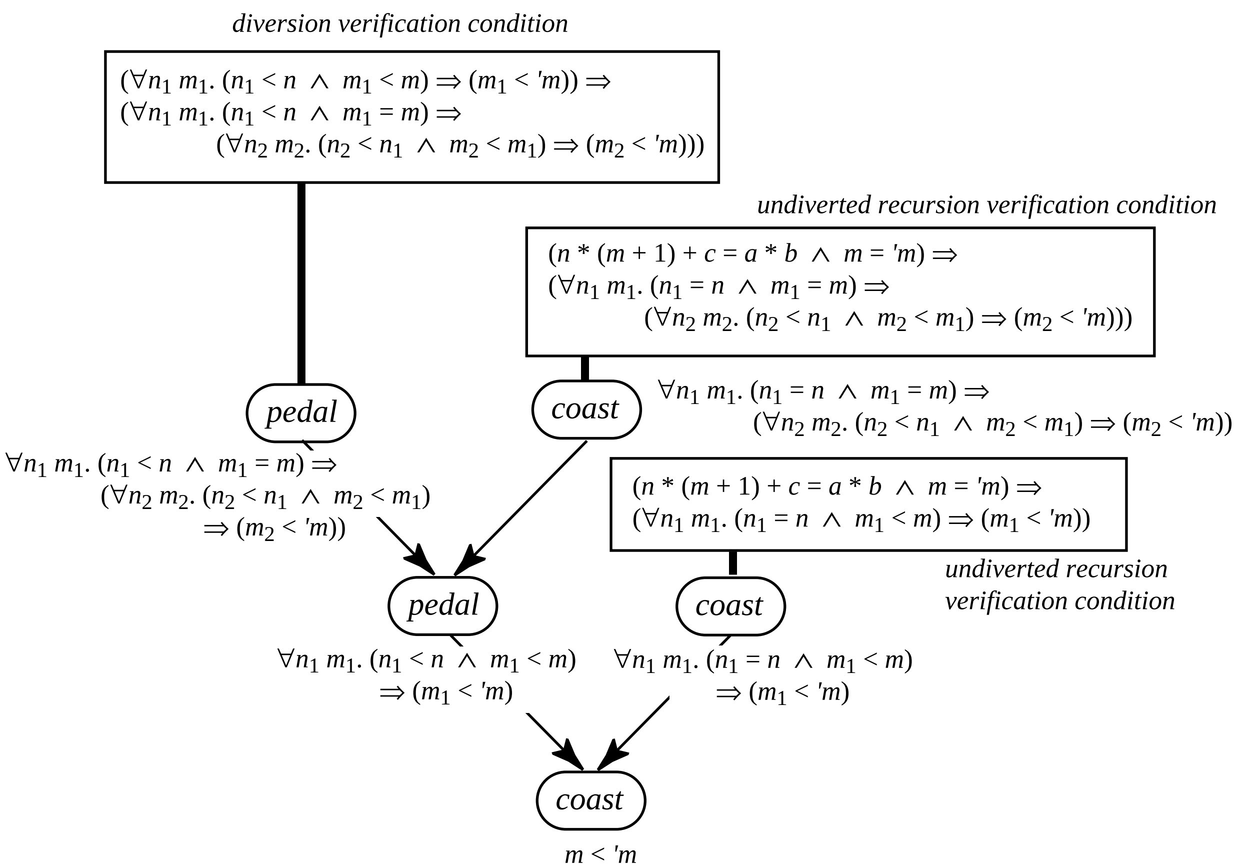


We end each branch of exploration when we encounter a duplicate of a node already present on the path back to the root.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Call Graph Verification Conditions

From this call graph tree, we generate verification conditions for each leaf node:



Leaves which duplicate the root produce *undiverted recursion verification conditions*.

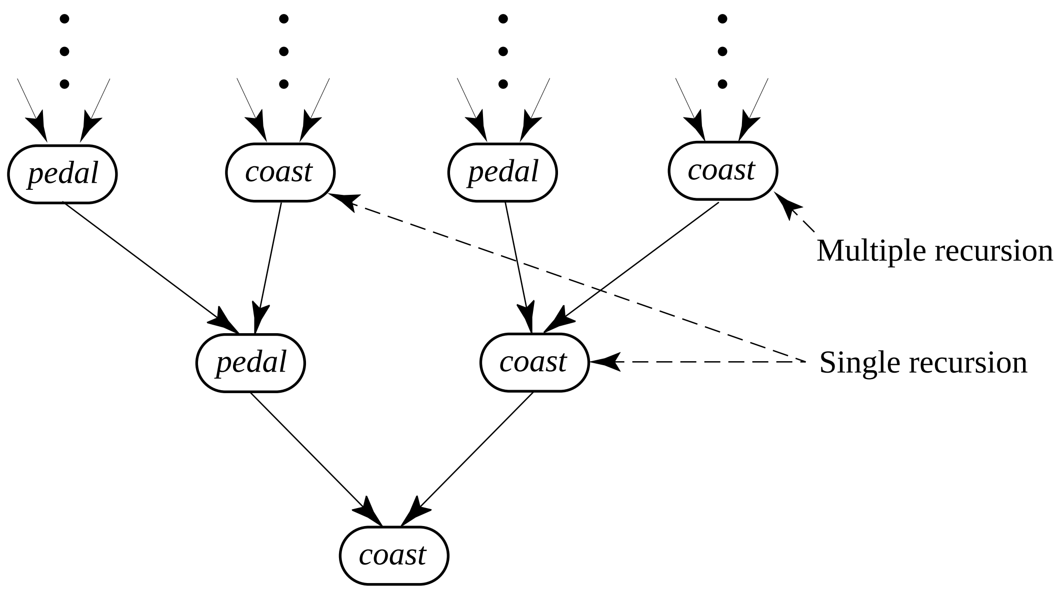
Leaves which do *not* duplicate the root produce *diversion verification conditions*.

Proof of Recursion

Must show progress for each and every path of recursion.

Must find all cycles, and develop expressions for progress across each.

But the full procedure call tree is infinite:



Identify nodes in the tree which duplicate the root. These are instances of *recursion*.

For each instance of recursion, examine its path to the root.

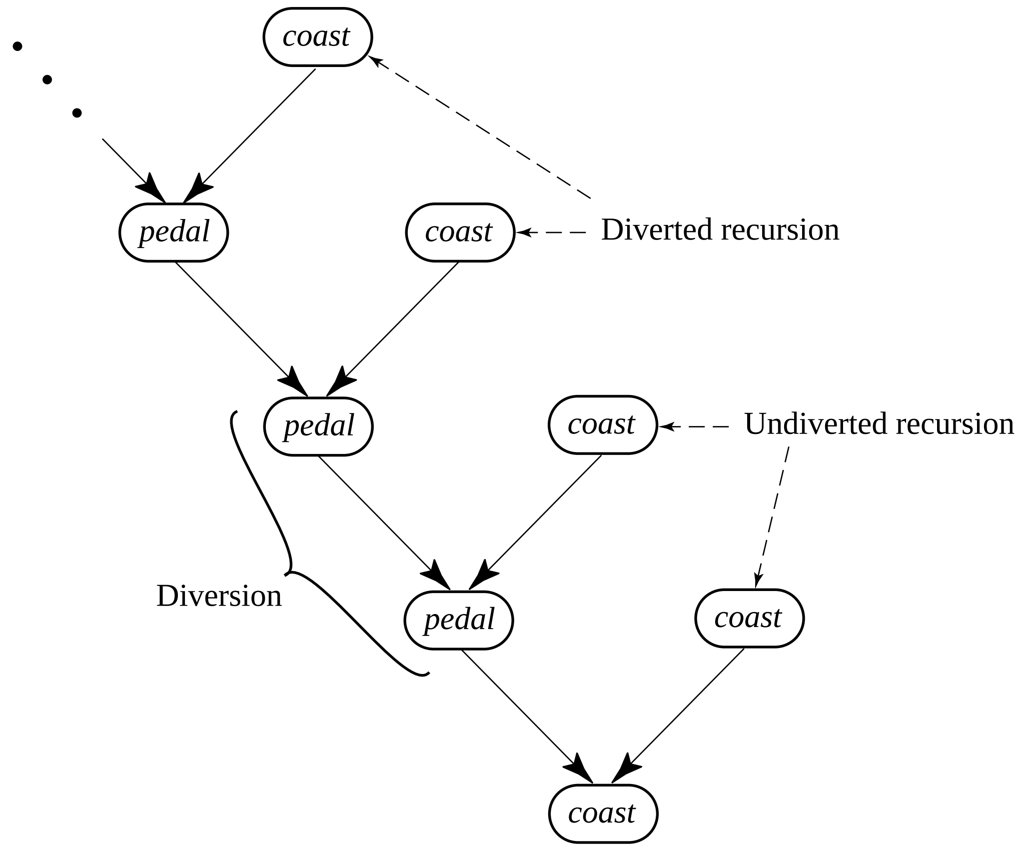
If the path contains another instance of recursion, this is an instance of *multiple recursion*. If it does not, this is an instance of *single recursion*.

The proof of full recursion simplifies to proving single recursion.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Proof of Single Recursion

Take the part of the tree which stops at each instance of single recursion:



Many leaves are duplicates of the root. This is still an infinite tree.

Consider each such leaf node, and examine its path to the root.

If the path contains internally duplicate nodes not the same as the root, this is called a *diversion*, and the leaf is an instance of *diverted recursion*.

The subtrees rooted at the two instances of *pedal* have identical structure.

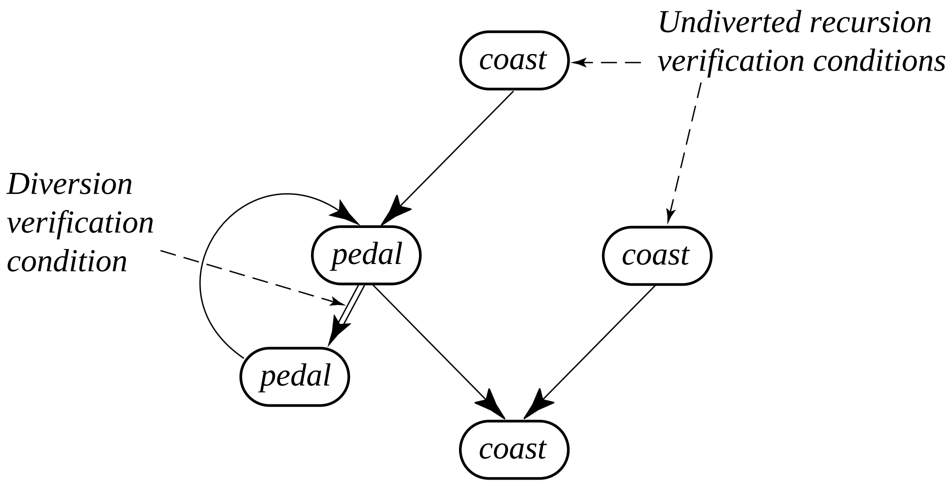
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Proof of Diverted Recursion

We can implicitly cover the infinite expansion of the tree by

bending the far endpoint of the diversion around and

connecting it to the near endpoint of the diversion.



The connection is established by a new *diversion verification condition*.

This VC is that the path condition at the *near* endpoint implies the path condition at the *far* endpoint.

This may seem *counterintuitive*, since the far endpoint is *previous* in time to the near endpoint.

This says that the diversion *does not interfere* with the proof of recursion. The path conditions do not lose progress going around the diversion.

☞ This reduces the proof burden to a finite number of VCs.

■
Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Main VCG Function

Definition of VCG:

vcg (**program** *d* ; *c* **end program**) *q* =
    let ρ = *mkenv d* ρ_0 **in**
    let h_1 = *vcgd d* ρ **in**
    let h_2 = *vcgg* (*proc_names d*) ρ **in**
    let h_3 = *vcgc* **true** *c g0 q* ρ **in**
        h_1 & h_2 & h_3

Theorem of Verification of VCG:

vcg_THM:
    $\vdash \forall \pi\, q.\ \ WFp\ \pi\ \wedge$ **every close** $(vcg\ \pi\ q) \Rightarrow \pi[q]$

The ultimate theorem of the verification of the VCG:
    For all programs π and specifications *q*,
    If the program π is well-formed, and
    If every verification condition generated by *vcg* is true,
    Then the program π is totally correct with respect to the spec *q*.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Example – Bicycling Mutual Recursion

We submit the following text to the prototype VCG embedded in HOL:

```
g `^(||` program
            procedure pedal (;val n,m);
               global a,b,c;
               pre  n*m + c = a*b;
               post c = a*b;
               calls pedal with n < 'n /\ m = 'm;
               calls coast with n < 'n /\ m < 'm;
               recurses    with n < 'n;

               if n = 0 \/ m = 0
               then skip
               else
                  c := c + m;
                  if n < m then coast(;n - 1,m - 1)
                           else pedal(;n - 1,m)
                  fi
               fi
            end procedure;

            procedure coast (;val n,m);
               global a,b,c;
               pre  n*(m + 1) + c = a*b;
               post c = a*b;
               calls pedal with n = 'n /\ m = 'm;
               calls coast with n = 'n /\ m < 'm;
               recurses    with m < 'm;

               c := c + n;
               if n < m then coast(;n,m - 1)
                        else pedal(;n,m)
               fi
            end procedure;

            a := 7;  b := 12;  c := 0;
            pedal(;a,b)

         end program
         [ c = a*b ]
      `||)`;
```

Verification Conditions for Bicycling

Then `VCG_TAC` reduces this goal to nine verification conditions:

```
#e(VCG_TAC);;
OK..
9 subgoals
```

- Partial correctness and entrance progress for procedure *pedal*:

```
!'n n 'm m 'a a 'b b 'c c.
 (('n = n) /\ ('m = m) /\ ('a = a) /\ ('b = b) /\ ('c = c)) /\
      (n * m + c = a * b) ==>
 (((n = 0) \/ (m = 0))
    => (c = a * b)
     | ((n < m)
         => (((n - 1) * ((m - 1) + 1) + c + m = a * b) /\
                n - 1 < 'n /\ m - 1 < 'm)
          | (((n - 1) * m + c + m = a * b)
             /\ n - 1 < 'n /\ (m = 'm))))
```

- Partial correctness and entrance progress for procedure *coast*:

```
!'n n 'm m 'a a 'b b 'c c.
 (('n = n) /\ ('m = m) /\ ('a = a) /\ ('b = b) /\ ('c = c)) /\
      (n * (m + 1) + c = a * b) ==>
 ((n < m)
   => ((n * ((m - 1) + 1) + c + n = a * b) /\ (n = 'n) /\ m-1 < 'm)
    | ((n * m + c + n = a * b) /\ (n = 'n) /\ (m = 'm)))
```

- The value of the recursion expression of the procedure *pedal* strictly decreases across the undiverted recursion path *pedal* → *pedal*.

```
!n m c a b 'n.
        (n * m + c = a * b) /\ (n = 'n) ==>
        (!n1 m1. n1 < n /\ (m1 = m) ==> n1 < 'n)
```

- The value of the recursion expression of the procedure *pedal* strictly decreases across the undiverted recursion path *pedal* → *coast* → *pedal*.

```
!n m c a b 'n.
        (n * m + c = a * b) /\ (n = 'n) ==>
        (!n1 m1. n1 < n /\ m1 < m ==>
                 (!n2 m2. (n2 = n1) /\ (m2 = m1) ==> n2 < 'n))
```

Verification Conditions for Bicycling (cont.)

- The diversion of *coast* in *coast* → *coast* → *pedal* does not interfere with the recursive progress of the procedure *pedal*.

```
!n m 'n.
        (!n1 m1. (n1 = n) /\ (m1 = m) ==> n1 < 'n) ==>
        (!n1 m1.
          (n1 = n) /\ m1 < m ==>
          (!n2 m2. (n2 = n1) /\ (m2 = m1) ==> n2 < 'n))
```

- The diversion of *even* in *pedal* → *pedal* → *coast* does not interfere with the recursive progress of the procedure *coast*.

```
!n m 'm.
   (!n1 m1. n1 < n /\ m1 < m ==> m1 < 'm) ==>
   (!n1 m1. n1 < n /\ (m1 = m) ==>
            (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < 'm))
```

- The value of the recursion expression of the procedure *coast* strictly decreases across the undiverted recursion path *coast* → *pedal* → *coast*.

```
!n m c a b 'm.
        (n * (m + 1) + c = a * b) /\ (m = 'm) ==>
        (!n1 m1.
          (n1 = n) /\ (m1 = m) ==>
          (!n2 m2. n2 < n1 /\ m2 < m1 ==> m2 < 'm))
```

- The value of the recursion expression of the procedure *coast* strictly decreases across the undiverted recursion path *coast* → *coast*.

```
!n m c a b 'm.
        (n * (m + 1) + c = a * b) /\ (m = 'm) ==>
        (!n1 m1. (n1 = n) /\ m1 < m ==> m1 < 'm)
```

- Total correctness for the main body:

```
7 * 12 + 0 = 7 * 12
```

We have proven these nine VCs in HOL; five of the VCs were solved by Richard Boulton's `ARITH_CONV` decision procedure.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Resulting HOL Theorem

Proving the nine VC's in HOL results in the following theorem:

```
|- program
      procedure pedal(var ;val n,m);
         global a,b,c;
         pre n * m + c = a * b;
         post c = a * b;
         calls pedal with n < 'n /\ m = 'm;
         calls coast with n < 'n /\ m < 'm;

         recurses with n < 'n;

         if n = 0 \/ m = 0 then skip
         else
            c := c + m;
            if n < m then coast(;n - 1,m - 1)
                     else pedal(;n - 1,m) fi
         fi
      end procedure;

      procedure coast(var ;val n,m);
         global a,b,c;
         pre n * (m + 1) + c = a * b;
         post c = a * b;
         calls pedal with n = 'n /\ m = 'm;
         calls coast with n = 'n /\ m < 'm;

         recurses with m < 'm;

         c := c + n; if n < m then coast(;n,m - 1)
                              else pedal(;n,m) fi
      end procedure;

      a := 7; b := 12; c := 0; pedal(;a,b)
   end program
   [c = a * b]
```

☞ This is a genuine HOL theorem of total correctness!

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Secure and Insecure Versions of VCG

Most of the time of applying the VCG_TAC is consumed in
-- conversion to evaluate well-formedness of program (WF)
-- conversion to calculate verification conditions of program (VCG)

These conversions were re-implemented in ML, using the same algorithm already verified in HOL logic, for faster execution:

Example	Secure WF	Insecure WF	Ratio
ex1	1.791 s	0.002 s	900
ex2	1.282 s	0.002 s	650
ex3	3.578 s	0.005 s	700
ex4	13.412 s	0.009 s	1500
ex5	3.486 s	0.004 s	900
ex6	4.653 s	0.005 s	900
ex7	12.406 s	0.010 s	1200

Example	Secure VCG	Insecure VCG	Ratio
ex1	7.171 s	0.010 s	720
ex2	12.842 s	0.012 s	1070
ex3	48.907 s	0.031 s	1580
ex4	388.763 s	0.147 s	2640
ex5	31.183 s	0.022 s	1420
ex6	965.904 s	0.350 s	2760
ex7	1085.146 s	0.420 s	2580

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Conclusions

about the mechanically verified VCG:

- The VCG verification encapsulated a level of semantic reasoning, proving it once, rather than repeating it for each application.
- Intricacies of semantics of termination require mechanical proof.
- There is a substantial difference between partial and total correctness involving procedures, which has not been generally appreciated.

about the new method of proving termination of procedures:

- The new method is more general and flexible than prior proposals, and more powerful, which enables more intuitive proofs.
  It is compositional and readily mechanized in a VCG.

about a trustworthy VCG:

- This VCG substantially decreases the difficulty of proving programs totally correct, and does so with a very high level of security.
- A VCG itself must be trustworthy for the proofs to be trustworthy.
- This level of trusworthiness is now demonstrated to be feasible, by the presentation of this mechanically verified VCG.

Peter Vincent Homeier
UNIVERSITY OF PENNSYLVANIA — DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE