ReSoLVe Workshop

Need to explain Resolve. Usually we explain new ideas in terms of familiar ones.

South Pacific “There is nothing like a dame.”

There is nothing like Resolve.

There’s been nothing to evolve that is anything like Resolve.

Active for more than 30 years

Other approaches: example Larch web last modified 1996

Resolve website probably modified yesterday.

Some History:

1983: Bruce and STILE, Bill and data structures course concerned that textbooks said good things in the first 1 or 2 chapters and then reverted to the same old ignoring of the stated principles about formal reasoning.

30 years ago, our founding fathers brought forth on the campus of the Ohio State University a new approach to software.

Bill wrote his own course notes and used a new bold approach.

Beginning of RSRG as 788

2 requirements for reusing software:

1. Clear specification of what the software does.

2. The software is correct.

Need for name: ReFormS Reusable Formally Specified Software

Reforms doesn’t say anything about verification.

Other suggested names: VerPL Verifying Programming Language

VerSoft Verifiable Software

PLM Programming Language with Mathematics

Late 1980’s: serious discussion in Robinson Lab northwest corner room. (Robinson—graduate offices on the 4th floor. 2 students stuck in elevator for a holiday weekend.

Bruce led the discussion, Bill on sabbatical. No google+, no skype, email barely beginning, not effective.

The group decided “We resolve to come up with a name for our project.”

Someone said “How about RESOLVE?”

Mutterings about carpet cleaners,

Re: reusable

So: software

L: language

with

Verification

Perfect!

Verifying compiler at Clemson

Resolve Parts:

Formal (Mathematical) Specifications

Separation of Concerns: Math Units, Concepts, Realizations, Facilities

Library: Syntactically written Math theories including proofs

Precis

Concepts and Realizations already proven correct

Proof Checker

Compiler

Collection of Proof Rules that extend mathematical logic with rules that automatically convert programming constructs to mathematical assertions

Example:

### Assume ;

;

**if then** abs := z

**else** abs := -z

**endif**;

**Confirm** abs = ;

code: **Assume** B; code1; **Confirm** Q;

code; **Assume **B; code2; **Confirm** Q;

---------------------------------------------------------------------------

code; **If B then** code1 **else** code2; **endif**; **Confirm** Q;

(1) **Assume ;**

;

**Assume ;**

abs := z;

**Confirm** abs = ;

(2) **Assume ;**

**;**

**Assume ;**

abs := z;

**Confirm** abs = ;

code; **Confirm** Q**;**

**------------------------------------------------**

code; x := exp; **Confirm** Q;

(1) **Assume** y 0;

z := w/y;

**Assume** z 0;

**Confirm** z = ;

(2) **Assume** y 0;

z := w/y;

**Assume** ;

**Confirm** –z = ;

We now need a rule for **Assume** statements:

code; **Confirm** P  Q;

------------------------------------

code; **Assume** P; **Confirm** Q;

Applying the rule for **Assume**, we obtain:

(1) **Assume ;**

z := w/y;

**Confirm** z  0  z = ;

(2) **Assume** y  0;

z := w/y;

**Confirm** z < 0  -z = ;

Now we apply the assignment rule to each branch:

(1) **Assume** y  0;

**Confirm** w/y  0  w/y = ;

(2) **Assume ;**

**Confirm ;**

Another application of the **Assume** rule yields:

1. **Confirm** y  0  ;
2. **Confirm** y  0 **;**

To complete the proof we need a rule for **Confirm**:

Q

--------------

**Confirm** Q;

Applying the **Confirm** rule produces the following mathematical propositions:

1. y  0  
2. y  0  

Verifier

Verification Condition (VC) generator

Prover