Verifying PETSc Vector Components Using CIVL

Venkata Dhavala[1] Jan Hückelheim[2] Paul D. Hovland[2] **Stephen F. Siegel**[1]

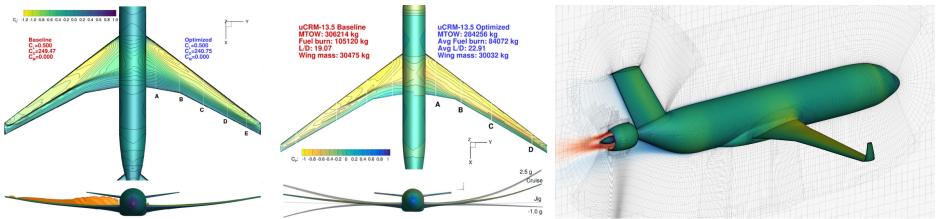
[1]University of Delaware, Newark DE, USA

[2]Argonne National Laboratory, Lemont IL, USA

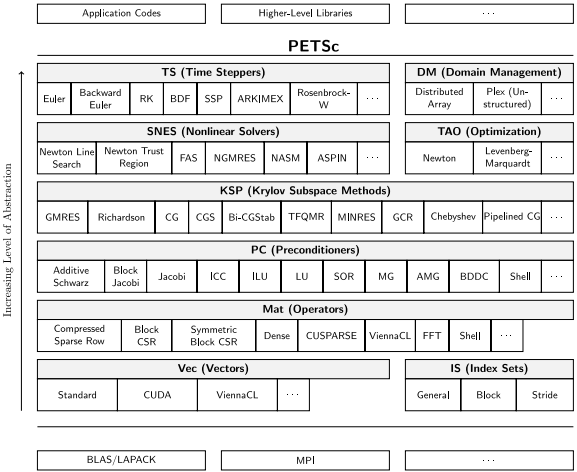
CAV 2025: 37th International Conference on Computer Aided Verification
Zagreb, Croatia
July 24, 2025

PETSc

- PETSc = Portable Extensible Toolkit for Scientific Computation
- library of scalable parallel algorithms for scientific applications modeled by PDEs
- used in numerous state-of-the-art scientific/HPC applications
  - acoustics, aerodynamics, air pollution, bone fractures, cancer treatment, cardiology, combustion, earthquake modeling, economics, fission & fusion, ocean dynamics, ...
- ~ 3000 citations to main paper (1997), ~ 6000 to users manual
- **DOE/NSF Workshop on Correctness in Scientific Computing 2023**
  - verification of PETSc identified as key challenge



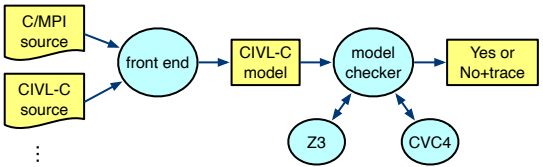
PETSc Code Characteristics



- $\sim 10^6$ non-comment lines of C code
- test suite executes over $13,000$ tests
- a complex object-oriented framework implemented by layers of macros and function pointers
- distributed data structures
  - MPI parallelism
- modular structure
- our project: target module **Vec** (Vectors)
  - used by almost all other modules

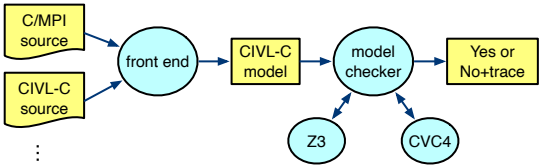
Verification Approach

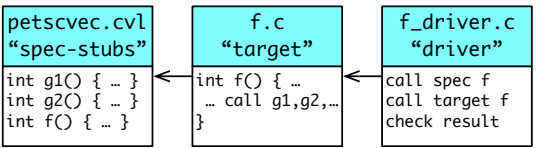
- use **CIVL Model Checker**: symbolic execution and bounded model checking
  - translates C/MPI into intermediate verification language **CIVL-C**
  - user can also write CIVL-C directly (e.g., libraries)
  - bound number of processes, vector lengths, other parameters



Verification Approach

- use **CIVL Model Checker**: symbolic execution and bounded model checking
  - translates C/MPI into intermediate verification language **CIVL-C**
  - user can also write CIVL-C directly (e.g., libraries)
  - bound number of processes, vector lengths, other parameters
- procedure-modular specification and verification
  - each function is verified independently of other functions
  - stubs and drivers
  - two new CIVL libraries
    - complex numbers and vectors

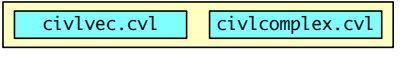




CIVL Vector Library Header File `civlvec.cvh` (excerpt)

- written in intermediate verification language CIVL-C
- functional model (no mutation)
- scalar type `STYPE` can be real or complex

```
/* The vector type. */
typedef struct $vec {
  int len;      // length of the vector
  STYPE data[]; // the elements of the vector
} $vec;
$vec $vec_make_from_dense(int len, STYPE *vals);
int $vec_len($vec vec);
bool $vec_eq($vec vec1, $vec vec2);
STYPE $vec_ith($vec vec, int i);
$vec $vec_add($vec vec1, $vec vec2);
$vec $vec_scalar_mul(STYPE a, $vec vec);
RTYPE $vec_norm($vec vec, int p);
$vec $vec_set($vec vec, int i, STYPE x);
```

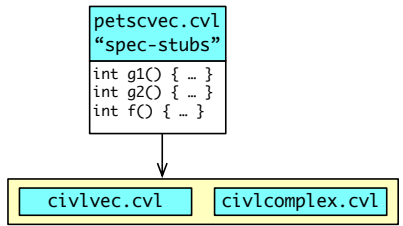


Spec-Stubs

- simple CIVL-C implementations of PETSc functions
- serve as **specifications** of PETSc functions
- serve as **stubs** for calls in PETSc functions

Example: $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$

```
PetscErrorCode VecAXPY(Vec y, PetscScalar alpha,
                       Vec x) {
  $assert(y->read_lock_count == 0);
  $assert(x->map->n == y->map->n);
  $vec c_x = CIVL_PetscToCivlVec(x),
     c_y = CIVL_PetscToCivlVec(y),
     c_z = $vec_add($vec_scalar_mul(alpha,c_x),c_y);
  CIVL_CivlToPetscVecCopy(c_z, y);
  return 0;
}
```



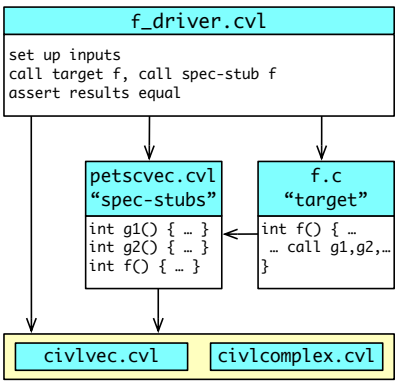
Spec-Stubs

- simple CIVL-C implementations of PETSc functions
- serve as **specifications** of PETSc functions
- serve as **stubs** for calls in PETSc functions

Example: $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$

```
PetscErrorCode VecAXPY(Vec y, PetscScalar alpha,
                       Vec x) {
  $assert(y->read_lock_count == 0);
  $assert(x->map->n == y->map->n);
  $vec c_x = CIVL_PetscToCivlVec(x),
     c_y = CIVL_PetscToCivlVec(y),
     c_z = $vec_add($vec_scalar_mul(alpha,c_x),c_y);
  CIVL_CivlToPetscVecCopy(c_z, y);
  return 0;
}
```

- functions called by target use the stub implementations
- a hand-written **driver** invokes target and spec



Case Study

- applied this approach to **61** core PETSc Vector functions
  - for all but 16
    - 1–5 MPI processes
    - 1–5 vector length
  - for 16 functions
    - path explosion due to branches on inputs
    - 1–4 MPI processes
    - 1–4 vector length
- total verification time: 4h35m
- 2 previously unknown defects discovered

VecAXPBY	VecMAXPY_Seq	VecNorm_Seq
VecAXPBYPCZ	VecMDot	VecNormalize
VecAXPBYPCZ_Seq	VecMDot_MPI	VecPointwiseDivide
VecAXPBY_Seq	VecMDot_Seq	VecPointwiseDivide_Seq
VecAXPY	VecMTDot	VecPointwiseMax
VecAXPY_Seq	VecMTDot_MPI	VecPointwiseMaxAbs
VecAYPX	VecMTDot_Seq	VecPointwiseMaxAbs_Seq
VecAYPX_Seq	VecMax	VecPointwiseMax_Seq
VecConjugate_Seq	VecMaxPointwiseDivide	VecPointwiseMin
VecCopy	VecMaxPointwiseDivide_MPI	VecPointwiseMin_Seq
VecCopy_Seq	VecMaxPointwiseDivide_Seq	VecPointwiseMult
VecDot	VecMaxPointwiseDivide_orig	VecPointwiseMult_Seq
VecDotRealPart	VecMax_MPI	VecScale
VecDot_MPI	VecMax_Seq	VecScale_Seq
VecDot_Seq	VecMin	VecSwap
VecGetSize	VecMin_MPI	VecSwap_Seq
VecGetSize_MPI	VecMin_Seq	VecTDot
VecGetSize_Seq	VecNorm	VecTDot_MPI
VecGetValues	VecNormAvailable	VecTDot_Seq
VecMAXPBY	VecNorm_MPI	VecWAXPY
VecMAXPY	VecNorm_MPI_orig	VecWAXPY_Seq

Defect 1: `VecMaxPointwiseDivide` missing global reduction

- given $\mathbf{v} = (v_1, \ldots, v_n)$ and $\mathbf{w} = (w_1, \ldots, w_n)$
- returns $\max_{1 \le i \le n} (w_i = 0 \ ? \ |v_i| \ : \ |v_i / w_i|)$
- every process should return the same global maximum
- original implementation
  - each process computes its local max and returns it — WRONG!
  - CIVL assertion violation when `nprocs` ≥ 2
  - reported to PETSc developers, who committed a fix
- corrected implementation
  - each process computes its local max
  - all processes engage in a global reduction (`MPI_Allreduce`) and return result
  - fixed version verifies successfully

Defect 2: `VecNorm_MPI`: incorrect `NORM_FROBENIUS` fall-through

- `VecNorm_MPI` computes the **norm** of a distributed vector
- $||\mathbf{v}||_p = (\sum_{i=1}^{n} |v_i|^p)^{1/p}$, $||\mathbf{v}||_\infty = \max_{i=1}^{n} |v_i|$
- each process computes the norm of its part, adjustments are made if necessary, then a global reduction
- an enumeration is used to specify the norm `type` for a matrix or vector
  - `NORM_1`, `NORM_2`, `NORM_FROBENIUS`, ...
- for vectors, Frobenius norm $=$ 2-norm

```
...
MPI_Op op = MPIU_SUM;
...
PetscCall(VecNorm_SeqFn(xin, type, z));
switch (type) {
case NORM_1_AND_2:
  z[1] *= z[1];
  zn = 2;
  break;
case NORM_2:
  z[0] *= z[0];
case NORM_FROBENIUS:
case NORM_1:
  break;
case NORM_INFINITY:
  op = MPIU_MAX;
  break;
}
...
```

Defect 2: `VecNorm_MPI`: incorrect `NORM_FROBENIUS` fall-through

- `VecNorm_MPI` computes the **norm** of a distributed vector
- $||\mathbf{v}||_p = (\sum_{i=1}^{n} |v_i|^p)^{1/p}$, $||\mathbf{v}||_\infty = \max_{i=1}^{n} |v_i|$
- each process computes the norm of its part, adjustments are made if necessary, then a global reduction
- an enumeration is used to specify the norm `type` for a matrix or vector
  - `NORM_1`, `NORM_2`, `NORM_FROBENIUS`, ...
- for vectors, Frobenius norm = 2-norm
- bug had been present for at least 2 years

```
...
MPI_Op op = MPIU_SUM;
...
PetscCall(VecNorm_SeqFn(xin, type, z));
switch (type) {
case NORM_1_AND_2:
  z[1] *= z[1];
  zn = 2;
  break;
case NORM_2:
  z[0] *= z[0];
case NORM_FROBENIUS:
case NORM_1:
  break;
case NORM_INFINITY:
  op = MPIU_MAX;
  break;
}
...
```

Conclusions

1. symbolic execution + bounded model checking proved effective for verifying/debugging vector functions in a state-of-the-art C/MPI numerical library
2. 61 functions verified; 2 previously undiscovered defects found
   - up to 4 or 5 MPI processes, vectors of length 4 or 5
3. a procedure modular approach, using stubs and drivers, was key
4. branch explosion is a challenge to scaling for some functions
   - concurrency explosion less of a challenge due to use of deterministic subset of MPI
5. specification is enabled by an abstract vector library with functional interface

All artifacts online:

https://github.com/verified-software-lab/civl-petsc