Code Quality Toolkit
Phase 1 Release

Personnel:

1. **Boyana Norris**, Associate Professor, University of Oregon (overall lead of activity)
2. **Shahid Hussain**, pro-tem researcher (design and implementation of analysis framework)
3. Kaley Chicoine, graduate student (implementation of static analysis)-left project on 3/30/21
4. **Bosco Ndemeye**, graduate student (implementation of static analysis)-joined project 3/31/21

norris@cs.uoregon.edu

Question for package developers:

What is your current PR/release workflow?

https://bit.ly/xSDK_01

Phase 1 release of code quality toolkit

Completion criteria:

1. Static and dynamic analysis framework for C, C++, and Fortran, open-source publicly available release

2. User documentation for the analysis framework

3. Documentation for xSDK developers on how to extend the framework with custom analyses

4. Example codes in xsdk-examples of applying this framework to several xSDK codes

Phase 1 release of code quality toolkit

Execution plan:

1. Design code quality analysis infrastructure based on LLVM for C and C++; incorporate both static and dynamic approaches.

2. Identify xSDK use cases for initial testing of new code analyses and apply the analyses to them

3. Extend analysis for Fortran (focus on FLASH)

4. In addition to general code quality metrics (e.g., cyclomatic complexity), define several examples of project-specific analyses

5. Add the example codes to the xsdk-examples test suite with documentation

Defects that program analyses can catch

Defects that result from inconsistently following simple, mechanical design rules.

–**Security:** Buffer overruns, improperly validated input.

–**Memory safety:** Null dereference, uninitialized data.

–**Resource leaks:** Memory, OS resources.

–**API Protocols:** improper use of APIs, incomplete/incorrect implementations

–**Exceptions:** Arithmetic/library/user-defined

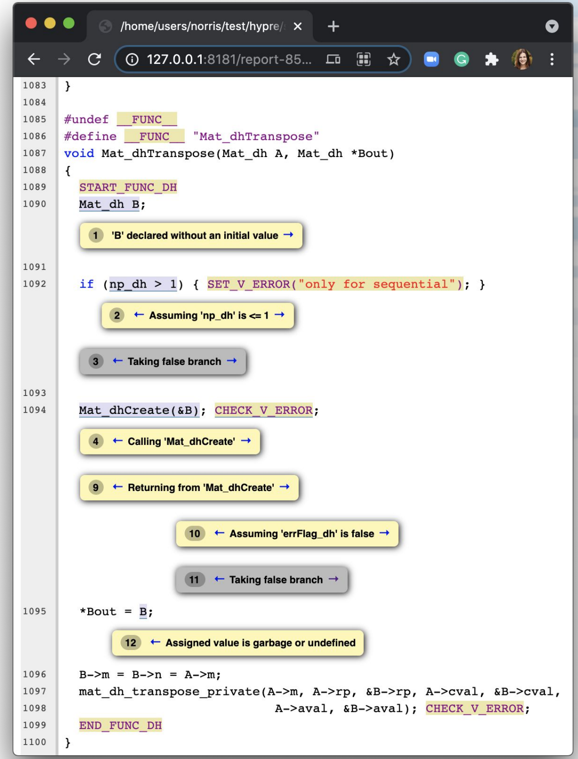
–**Encapsulation:** Accessing internal data, calling private functions.

–**Data races:** Two threads access the same data without synchronization

Key: check compliance to (mostly) simple, mechanical design rules.

General-purpose tools for code checking (bugs, style)

- C/C++
  - Run a bunch of general analyses with `scan-check` (wrapper around `clang --analyze`, which is uses the static analyzer below)
    - Minimally invasive, not very customizable
    - Works great with CMake and Autoconf builds
  - Clang static analyzer component: *extensible* analysis framework for bug finding
    - Can do more complex analyses (path-sensitive, inter-procedural analysis based on a symbolic execution technique)
    - Requires more compiler knowledge to extend
  - Clang-tidy: *extensible* (libTooling-based) framework for diagnosing typical programming errors or style issues
    - Checking and enforcing of simple coding conventions
    - Modular, provides API for implementing new checks
    - Relatively easy to integrate into Cmake
- Fortran
  - Flang (compiler front-end to LLVM)
  - Fortran-linter (limited)



Our goals and approach

Make it easy(-ish) to define and apply static and dynamic program analysis techniques to identify quality-related problems in xSDK codes.

How?

1. By integrating general **static** and **dynamic** program analyses into the xSDK development process
2. Creating easy interfaces to (and many examples of) custom analyses.

Why?

- Abstraction
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- Programs as data
  - Programs are just trees/graphs!
  - …and we have lots of ways to analyze trees/graphs

Static program analysis is...

Ensure everything is checked the same way.

Examples:
- clang-tidy
- Clang static analyzer

Systematic examination of an **abstraction** of program **state space.**

Only track "important" things...

Applies to all possible executions.

Dynamic program analysis is...

Instrumented code only.

Examples:

- Valgrind
- Clang/LLVM sanitizers (better!)

Partial examination of an **abstraction** of a **single** execution path at **runtime**.

Can capture information not available statically.

Applies to specific executions; can miss errors.

Compare with inspection, testing

- Pointers, Array Bounds, Interrupts
  - Testing
    - Errors typically on uncommon paths or uncommon input
    - Difficult to exercise these paths
  - Inspection
    - Non-local and thus easy to miss
    - Array allocation vs. index expression
    - Disable interrupts vs. return statement
- Example: Finding Race Conditions
  - Testing
    - Cannot force all interleavings
  - Inspection
    - Too many interleavings to consider
    - Check rules like “lock protects x” instead
      - But checking is non-local and thus easy to miss a case

Implementation approach: Part 1

Whenever possible, leverage existing compiler infrastructure:

- C/C++:
  - Static: Clang Static Analyzer (Clang CFG-based) and clang-tidy interfaces (Matcher-based)
    - Integrate existing checks into xSDK builds
    - Implement custom checkers based on coding policies (when available)
    - Produce even higher-level APIs to enable xSDK developers to extend checks
    - Check run during package *builds*
    - Questions/challenges:
      - How to deal with non-CMake builds? Spack support?
      - Output formats? (readable, understandable, actionable)
      - Minimize false positives
  - Dynamic: Python interfaces to the Clang sanitizer API
    - Used in our current implementation of PETSc development rule checking
    - Requires integration with build system and can be used during *testing*
    - Questions/challenges:
      - Minimize runtime overheads
      - Output formats for results? (readable, understandable, actionable)

Implementation approach: Part 1

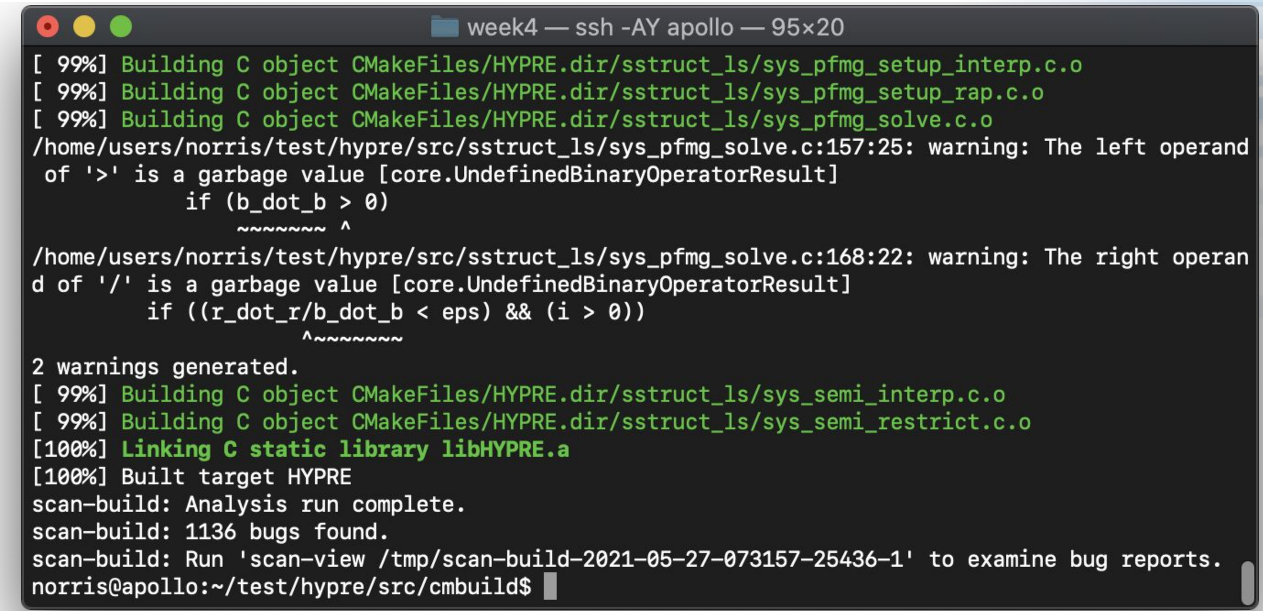
Whenever possible, leverage existing compiler infrastructure:

- Fortran:
  - Static: https://pypi.org/project/fortran-linter/ (extrermely limited)
  - Dynamic: ?

Yikes?

Example: Using scan-build with hypre

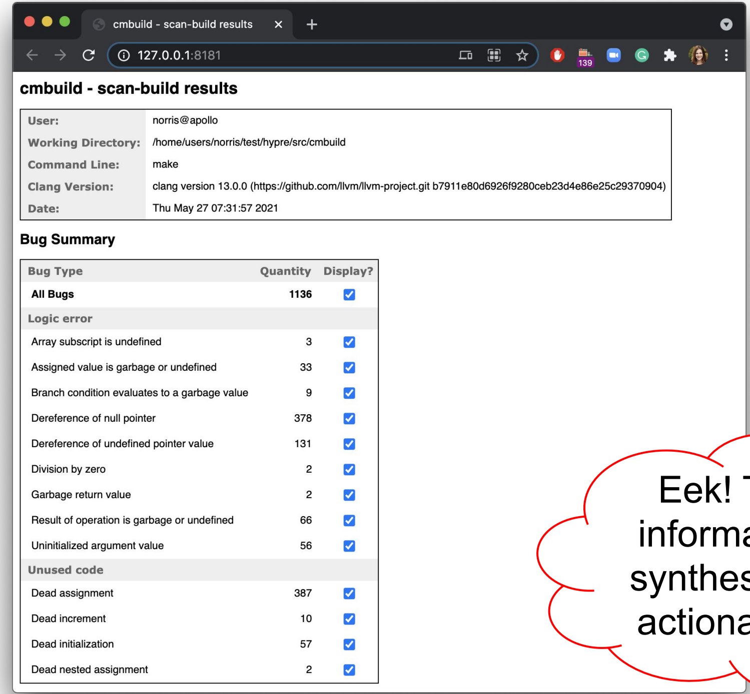
```
hypre/src/cmbuild$ scan-build cmake ..
hypre/src/cmbuild$ scan-build make
```



```
week4 — ssh -AY apollo — 95×20
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_setup_interp.c.o
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_setup_rap.c.o
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_solve.c.o
/home/users/norris/test/hypre/src/sstruct_ls/sys_pfmg_solve.c:157:25: warning: The left operand
 of '>' is a garbage value [core.UndefinedBinaryOperatorResult]
            if (b_dot_b > 0)
                ~~~~~~~ ^
/home/users/norris/test/hypre/src/sstruct_ls/sys_pfmg_solve.c:168:22: warning: The right operan
d of '/' is a garbage value [core.UndefinedBinaryOperatorResult]
         if ((r_dot_r/b_dot_b < eps) && (i > 0))
                     ^~~~~~~~
2 warnings generated.
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_semi_interp.c.o
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_semi_restrict.c.o
[100%] Linking C static library libHYPRE.a
[100%] Built target HYPRE
scan-build: Analysis run complete.
scan-build: 1136 bugs found.
scan-build: Run 'scan-view /tmp/scan-build-2021-05-27-073157-25436-1' to examine bug reports.
norris@apollo:~/test/hypre/src/cmbuild$
```

Example: hypre (cont.)

```
hypre/src/cmbuild$ scan-view /tmp/scan-build-2021-05-27-073157-25436-1
```



cmbuild - scan-build results

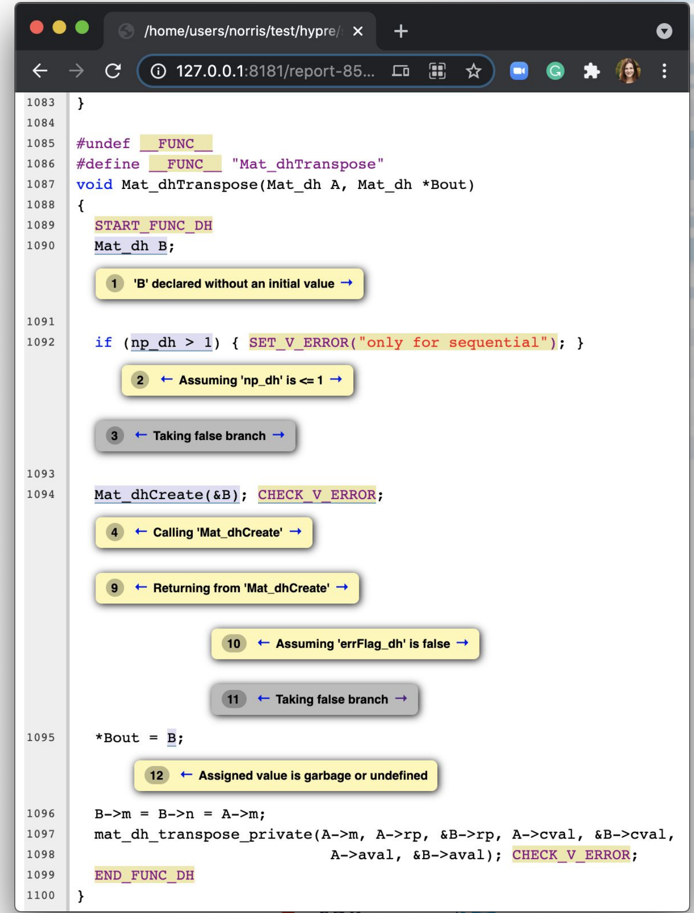
User:	norris@apollo
Working Directory:	/home/users/norris/test/hypre/src/cmbuild
Command Line:	make
Clang Version:	clang version 13.0.0 (https://github.com/llvm/llvm-project.git b7911e80d6926f9280ceb23d4e86e25c29370904)
Date:	Thu May 27 07:31:57 2021

Bug Summary

Bug Type	Quantity	Display?
All Bugs	**1136**	☑
Logic error		
Array subscript is undefined	3	☑
Assigned value is garbage or undefined	33	☑
Branch condition evaluates to a garbage value	9	☑
Dereference of null pointer	378	☑
Dereference of undefined pointer value	131	☑
Division by zero	2	☑
Garbage return value	2	☑
Result of operation is garbage or undefined	66	☑
Uninitialized argument value	56	☑
Unused code		
Dead assignment	387	☑
Dead increment	10	☑
Dead initialization	57	☑
Dead nested assignment	2	☑

details

Eek! Too much information, must synthesize a more actionable report.




```
}

#undef __FUNC__
#define __FUNC__ "Mat_dhTranspose"
void Mat_dhTranspose(Mat_dh A, Mat_dh *Bout)
{
  START_FUNC_DH
  Mat_dh B;
      1 'B' declared without an initial value →

  if (np_dh > 1) { SET_V_ERROR("only for sequential"); }
      2 ← Assuming 'np_dh' is <= 1 →
      3 ← Taking false branch →

  Mat_dhCreate(&B); CHECK_V_ERROR;
      4 ← Calling 'Mat_dhCreate' →
      9 ← Returning from 'Mat_dhCreate' →
      10 ← Assuming 'errFlag_dh' is false →
      11 ← Taking false branch →
  *Bout = B;
      12 ← Assigned value is garbage or undefined
  B->m = B->n = A->m;
  mat_dh_transpose_private(A->m, A->rp, &B->rp, A->cval, &B->cval,
                           A->aval, &B->aval); CHECK_V_ERROR;
  END_FUNC_DH
}
```

UNIVERSITY OF OREGON Argonne NATIONAL LABORATORY

What about project-specific requirements?

Do you need to be a compiler expert to implement new program checks?

Thankfully -- **no!**

Implementation approach: Part 2

Develop custom static and dynamic checking based on xSDK requirements.

- C/C++
  - Static: use and extend existing APIs (Clang static analyzer, clang-tidy); implement custom AST traversals and matchers (more details next)
  - Dynamic: use Clang sanitizer APIs; python for simplicity and easy of extensions by xSDK developers

- Fortran
  - Static: need to write new Flang-based checkers *only* for things that Fortran developers actually care about
  - Dynamic: do we need anything?

Implementation Details for Part 2

Program analysis tools (static or dynamic)

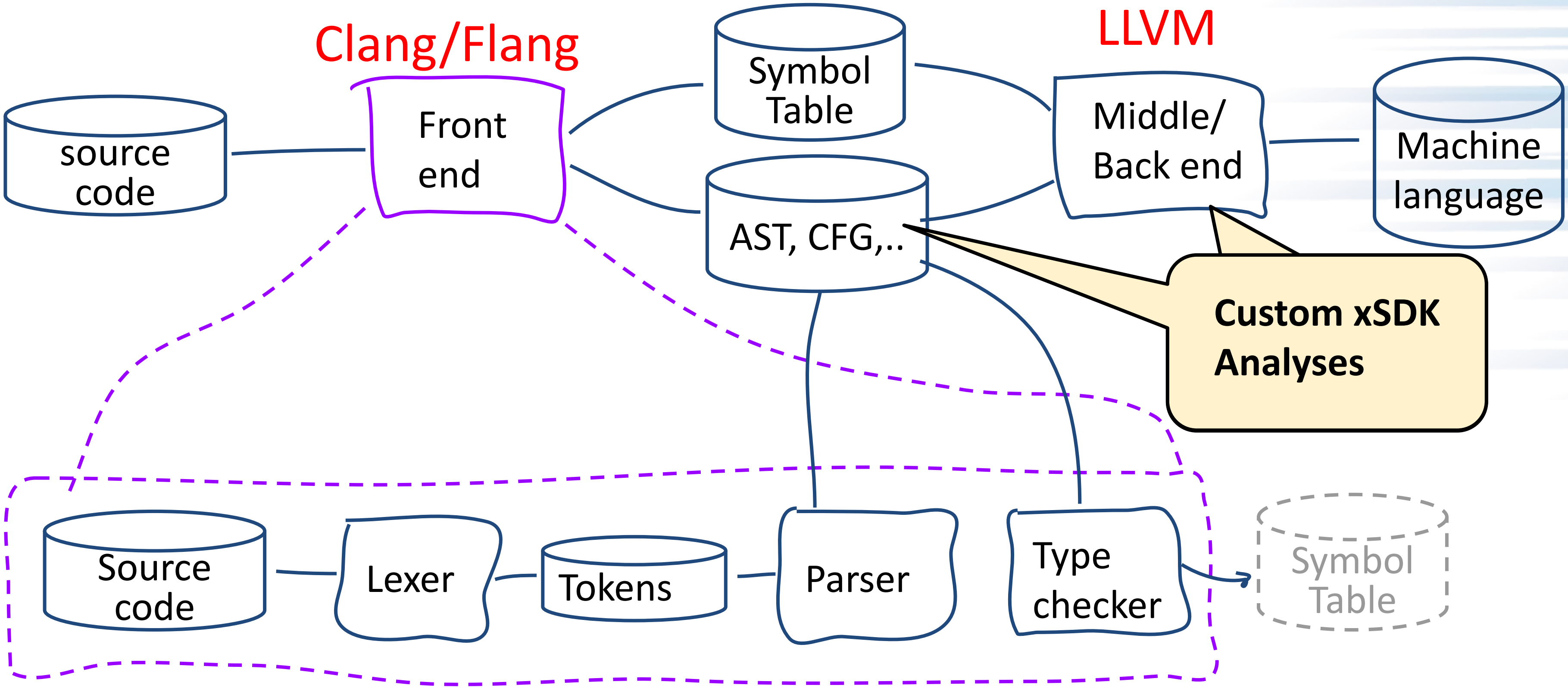
Abstraction

- Elide details of a specific implementation.
- Capture semantically relevant details; ignore the rest.

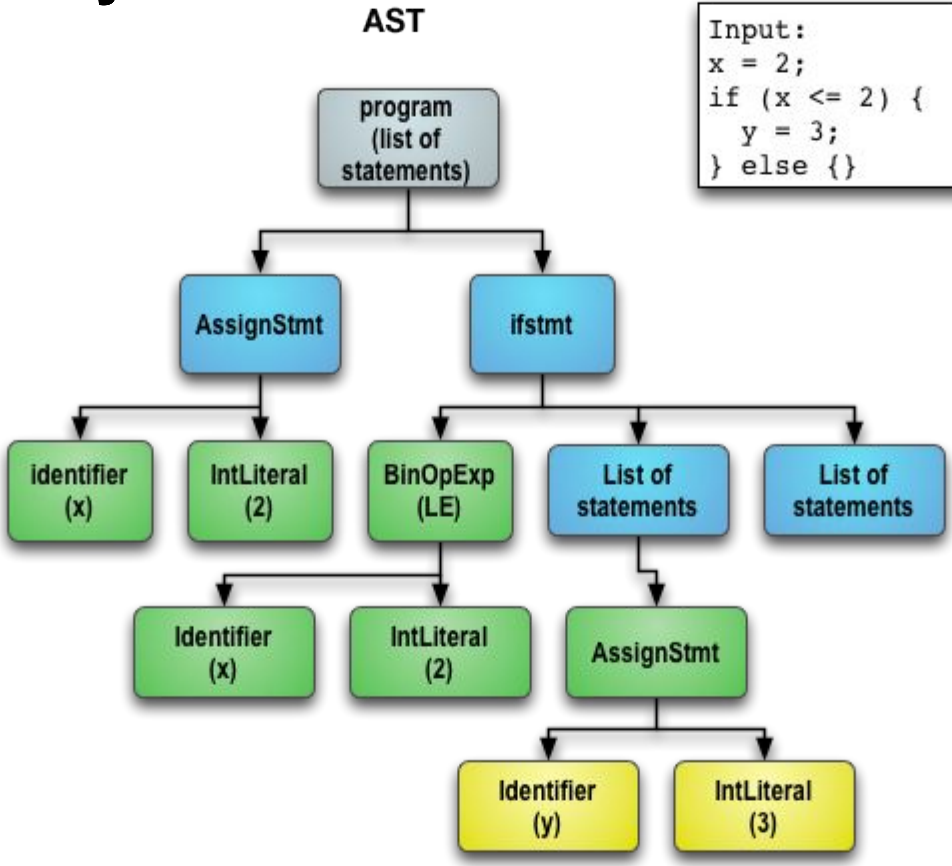
Programs as data

- Programs are just trees/graphs!
- …and we have lots of ways to analyze trees/graphs

Structure of a compiler



Abstract Syntax Trees




```
Input:
x = 2;
if (x <= 2) {
  y = 3;
} else {}
```

Beyond ASTs

- Interprocedural analyses require more than just AST traversals

- Other program representations required include
  - Call graphs
    - Challenging in C codes with virtual function pointers, e.g., PETSc
  - Control-flow graphs

- Main challenge:
  - How to enable customizable inter-procedural checks

Coding Rule Violations in HPC Packages: PETSc Case Study

PETSc developer rules: https://petsc.org/release/developers/style/

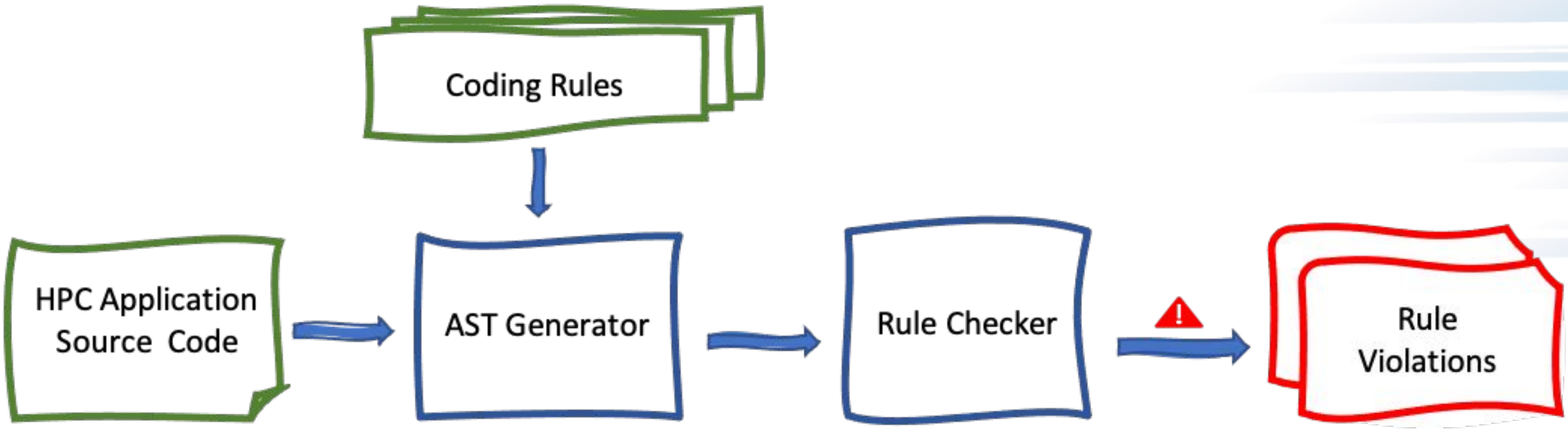
Consider this subset:

Rule	Description
Rule-1-A	All function names consist of acronyms or words, each of which is capitalized, for example, KSPSolve() and MatGetOrdering().
Rule-1-B	All enum types consist of acronyms or words, each of which is capitalized, for example, KSPSolve() and MatGetOrdering().
Rule-2-A	All macro variables are named with all capital letters. When they consist of several complete words, there is an underscore between each word. For example, MATFINALASSEMBLY.
Rule-2-B	All enum elements are named with all capital letters. When they consist of several complete words, there is an underscore between each word. For example, MATFINALASSEMBLY.
Rule-3	Functions that are private to PETSc (not callable by the application code) either. • have an appended `_Private` (for example, `StashValues_Private`, or • have an appended _Subtype (for example, MatMultSeq_AIJ.
Rule-4	Function names in structures (for example, `_matops`) are the same as the base application function name without the object prefix and are in small letters. For example, `MatMultTranspose()` includes the structure name `multtranspose`.
Rule-5	Names of implementations of class functions should begin with the function name, an underscore, and the name of the implementation, for example, `KSPSolve_GMRES()`.
Rule-6	Do not use `if (rank == 0)` or `if (v == NULL)` or `if (flg == PETSC_TRUE)` or `if (flg ==PETSC_FALSE)`. Instead, use `if (!rank)` or `if (v)` or `if (flg)` or `if (flg)`.
Rule-7	Do not use `#ifdef` or `#ifndef`. Rather, use `#if defined(...` or `#if defined(...`. Better, use `PetscDefined()`.

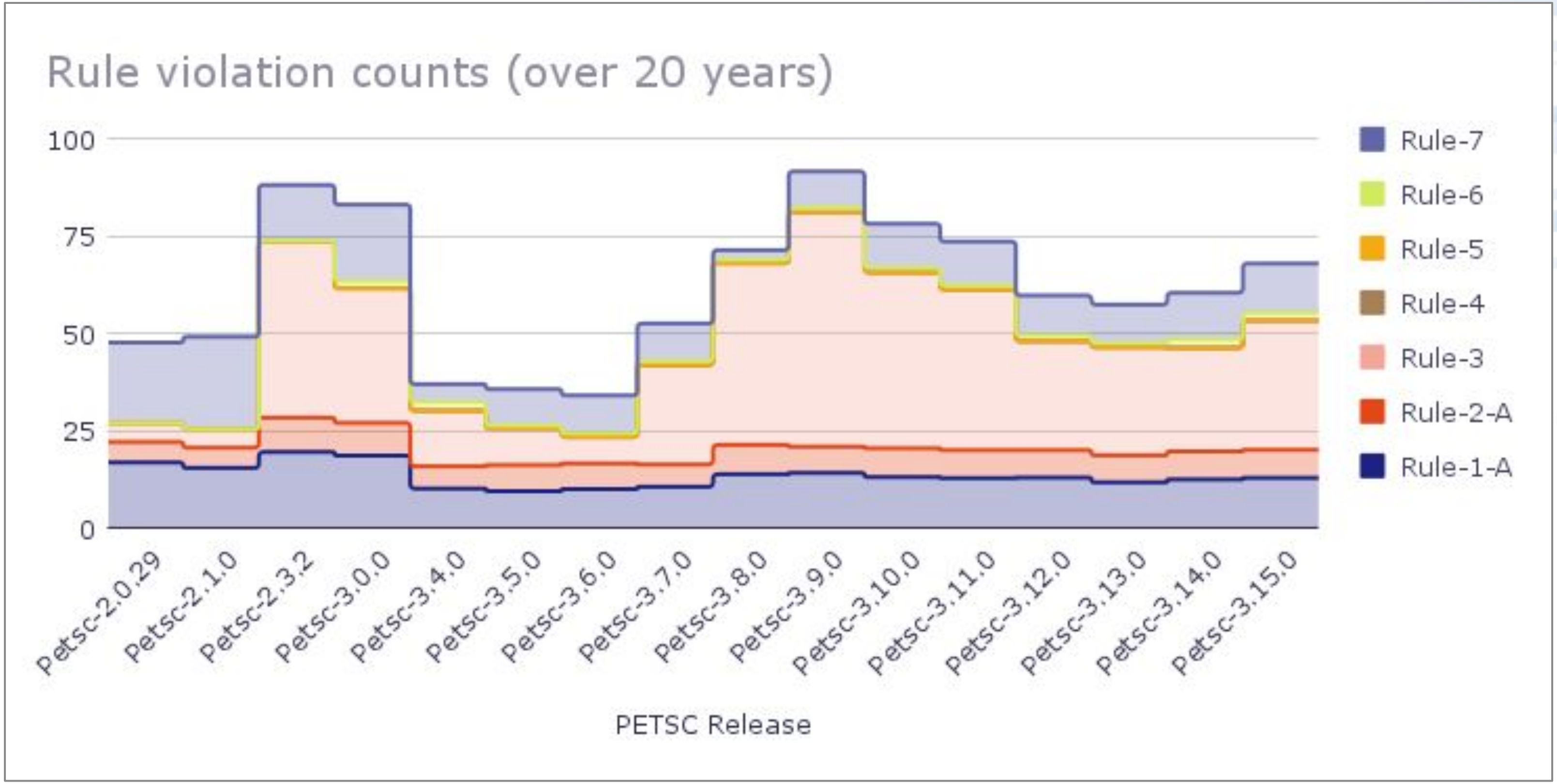
Examples of developer rule violations (PETSc 3.15)

PETSc Rule	PETSc Construct	Description	path	Line	Column
Rule-1	Function	PetscErrorCode PETSCMAP1(VecScatterBeginMPI3Node)(VecScatter ctx,Vec xin,Vec yin,InsertMode addv,ScatterMode mode)	/home/users/shussain/PETSC4/petsc-3.14.3/src/vec/vscat/impls/mpi3/vpscat.h	249	16
Rule-2	Macro	#define mpi_reduce_scatter PETSC_MPI_REDUCE_SCATTER	/home/users/shussain/PETSC4/petsc-3.14.3/include/petsc/mpiuni/mpiunifdef.h	118	2
Rule-3	Function (Defnition)	PETSC_INTERN PetscErrorCode MatFactorFactorizeSchurComplement_Private(Mat);	/home/users/shussain/PETSC4/petsc-3.14.3/include/petsc/private/matimpl.h	494	29
	Function (Call)	PETSC_EXTERN PetscErrorCode MatFactorFactorizeSchurComplement(Mat);	/home/users/shussain/PETSC4/petsc-3.14.3/include/petscmat.h	1245	29
Rule-4	Function in Structure	PETSC_EXTERN PetscErrorCode DMDAVecGetArray(DM,Vec,void *)	/home/users/shussain/PETSC4/petsc-3.14.3/include/petscdmda.h	113	29
	Function in Base application	ierr = VecGetArray(y,yv)	/home/users/shussain/PETSC4/petsc-3.14.3/include/petscvec.h'	545	10
Rule-5	Function	ierr = PetscFEPushforwardGradient(fe, fegeom, 1, interpolantGrad);	'/home/users/shussain/PETSC4/petsc-3.14.3/include/petsc/private/petscfeimpl.h'	332	10
Rule-6	If	if (p == 0) return node;	/home/users/shussain/PETSC4/petsc-3.14.3/src/dm/impls/plex/gmshlex.h	231	3
Rule-7	Macro	#ifndef PETSC4PY_COMPAT_MUMPS_H	/home/users/shussain/PETSC4/petsc-3.14.3/src/binding/petsc4py/src/include/compat/mumps.h	1	1

Checking for violations of the PETSc developer rules



Example results for a subset of the rules



What next?

- General static analysis tools:
  - Create documentation on how xSDK packages can integrate them into their development workflows
  - Provide example workflows for some of the xSDK packages (C, C++, Fortran)

- Custom analyses: redesign/reimplement the collection of scripts and C++ code to make this into a usable toolkit
  - The collection of PETSc rules would be a good example for C codes
  - Will create similar examples for other languages (next is C++, then Fortran)
  - Combine static and dynamic analysis results (currently they are separate)
  - Consider any of the xSDK code policies that are specific enough to enable automation

Extra

Example ‘make commit’ workflow (C/C++, but hopefully possible in some form for Fortran):

- clang-format passes and reformats the code
- clang-tidy passes and enforces coding conventions
- clang static analyzer compiles debug and production builds (check errors)
- xSDK specific analysis for debug and production build (check errors)
- debug/production builds get compiled and unit tests launched (check errors)
- production build + unit tests run under valgrind (check errors)
- production build with ASan/MSan/TSan gets compiled and unit test launched (check errors)
- production build with --coverage gets compiled and unit test launched against llvm-cov (write unit-test coverage stats)