C++ Software Quality in the ATLAS Experiment

Stefan Kluth, Stewart Martin-Haugh, Emil Obreshkov, Shaun Roe, Rolf Seuster, Scott Snyder, Graeme Stewart

Tools in use
Experience
Progress, Pleasures and Pitfalls
Compilation * Static Analysis * run-time behaviour
continuous testing

TOOLS

Gcc-plugin tctoolkit

Coverity

Cppcheck

Clang Ubsan

PRQA
ATLAS code

- 2 million lines of code
- 6 million developers
- ~420 domains
- 140 'domains'
- e.g. Tracking-Fitting
- 2400 individual packages
- (gcc 4.9)
Compiler checks

Code should always, where possible, compile without warnings.
It is beneficial to expose the code to different compilers; e.g. **Clang builds** give additional info:
- Mismatched struct/class; Unused private class members; Mismatched header guard.
- `std::abs(eta < 2.5)`
- `if ((bitfield && 0x0011) != 0) ...`
- `!val==10`

**gcc plug-ins**
- Check for inheritance structure already in place
- Coming soon: naming convention checks
Static Analysis

Attempts to follow your program and points out possible errors; some examples:

- ‘new’ without delete
- scalar ‘delete’ on array object
- array indices out-of-bounds
- possible cut-and-paste errors
- suspect indentation

The examples are drawn from Coverity®, one of the most widely used static analysers.
12598 09/07/2014 (High) Resource leak :/
ForwardDetectors/ALFA/ALFA_CLinkAlg/src/ALFA_CLinkAlg.cxx in function "execute"
Pitfalls: Coverity’s classification may not correspond to your own; e.g. **Uninitialised members** are classified ‘high’, as are obvious **resource leaks**; however a **faulty assignment operator** which can also leak resources will be ‘low’. **Parse errors** can occur and are classified ‘low’ but may mask many other defects.
Industry report 2012

“Coverity’s analysis found an average defect density of .69 for open source software projects that leverage the Coverity Scan service, and an average defect density of .68 for proprietary code developed by Coverity enterprise customers. Both have better quality as compared to the accepted industry standard defect density for good quality software of 1.0 [defects/kloc].”
Coverity® pleasures and pitfalls

“spending hours with a memory profiler could save you 5 minutes looking at the coverity report” 😊

False positives (very few): 😞
e.g. “restoring ostream format”
“parse errors” (particularly complicated templates)

Near misses:

```
792  //set the foreign key
6. return_constant: Function call variableType() returns 4.
CID 11595 (#2-1 of 2): Out-of-bounds read (OVERRUN)
7. overrun-local: Overrunning array of 4 bytes at byte offset 4 by dereferencing pointer &"_FK"[variableType()].
793  pixel_columns.createForeignKey(variableType<T>()+"_FK","FK",m_pixmaptable,"FK");
794  // create indices
```

Misses: if (m_Analogue[1][strip]<0.0000001 || m_Analogue[1][strip]>-0.0000001){ //
Other Static Analysers

Installed, to be integrated into reporting system:
cppcheck *(running routinely since Nov ’15)*
  Open source, easily configurable, quick (~1 hr)
  somewhat noisy, more false positives, but does find additional defects. Output to XML/static web page. Options for performance (e.g. “use pre-increment”), style.

Include-what-you-use
  Easy to set up; scope is limited to tidying up #includes; web interface available for reports (R. Seuster).

OCLint
  Open source, based on Clang static analysis tools; ~12 hr scan with integrated result-to-web generation.

Investigated:
  PRQA: Nice system, flexible, industry standards, can enforce naming conventions; $$, difficult to integrate in our build system
cppcheck

Top level web display

<table>
<thead>
<tr>
<th>Line</th>
<th>Id</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>440</td>
<td>missingInclude</td>
<td>information</td>
<td>Cppcheck cannot find all the include files (use --check-config for details)</td>
</tr>
<tr>
<td>399</td>
<td>coutCerrMisusage</td>
<td>error</td>
<td>Invalid usage of output stream: '&lt;&lt; std::cout'.</td>
</tr>
<tr>
<td>258</td>
<td>nullPointer</td>
<td>warning</td>
<td>Possible null pointer dereference: vot - otherwise it is redundant to check it against null.</td>
</tr>
<tr>
<td>161</td>
<td>mismatchAlloc Dealloc</td>
<td>error</td>
<td>Mismatching allocation and deallocation: disto</td>
</tr>
<tr>
<td>75</td>
<td>uninittMemberVar</td>
<td>warning</td>
<td>Member variable 'Calibrator::resHist' is not initialized in the constructor.</td>
</tr>
<tr>
<td>493</td>
<td>memleak</td>
<td>error</td>
<td>Memory leak: hist</td>
</tr>
<tr>
<td>650</td>
<td>invalidscf_libc</td>
<td>portability</td>
<td>scanf without field width limits can crash with huge input data on some versions of libc.</td>
</tr>
<tr>
<td>706</td>
<td>invalidscf_libc</td>
<td>portability</td>
<td>scanf without field width limits can crash with huge input data on some versions of libc.</td>
</tr>
<tr>
<td>707</td>
<td>invalidscf_libc</td>
<td>portability</td>
<td>scanf without field width limits can crash with huge input data on some versions of libc.</td>
</tr>
<tr>
<td>708</td>
<td>invalidscf_libc</td>
<td>portability</td>
<td>scanf without field width limits can crash with huge input data on some versions of libc.</td>
</tr>
<tr>
<td>898</td>
<td>uninittStructMember</td>
<td>error</td>
<td>Uninitialized struct member: startdata.det</td>
</tr>
</tbody>
</table>

..linked to annotated code

```cpp
this->cd(1);
ModuleTuple->Draw("y:x","resHist10(40,-1200,1200,40,-1200,1200)",selectionA,"colz");
TH2F *resHist10 = (TH2F*)gPad->GetPrimitive("resHist10");
if (!resHist10) throw string("Variable not found!");
if (variable == "res") { // String literal compared with variable 'variable'. Did you intend to use strcmp() instead?
  resHist10->GetZaxis()->SetRangeUser(0.135, 0.150);
```
Include-what-you-use

Choose first the project, then follow path to your favourite package

…advise on which ‘includes’ to add, remove or can be replaced by a forward class declaration.
Run-time Sanitizers

Implemented in nightly tests of DBG builds:

• **UBSan (undefined behaviour)**
  Clear warnings; easy to implement:

  ```
  HepMcParticleLink.h:72:51: runtime error: left shift of 200001 by 16 places cannot be represented in type ’int’
  ```

Investigated:

• **ASan (address)**
  Similar to the debug tool Valgrind but much faster and catches more; output maybe a bit intimidating...
  ABI incompatible.

• **TSan (thread)**
  Currently not useful ‘out of the box’ for ATLAS
"If all you have is a hammer, everything looks like a nail"
- Psychology of Science (Maslow, 1966)

Coding to solve the line-by-line problems revealed by these tools might produce better code, but it won’t forcibly produce good code.

“Code smell” detectors (tentative results so far):

- **TCToolkit**
  
  ~1 hr to scan release, produces >50Mb html.
  
  - code duplication detector (looks useful)
  
  - token tag cloud (“how noisy” the code is)
  
  - class co-occurrence matrix (interdependencies)
Continuous testing

‘ATN’ tests run nightly in all releases;
  • Unit tests configured in a dedicated SVN test directory for the package
  • XML file describes expected output; maybe simple return code, or comparison with a reference file.

‘RTT’ tests
  • Produce plots of physics quantities (momenta, track parameters etc) and compared with reference plots
  • Reviewed by shifters, who report each week.
Summary

• All of these tools are useful as developer aids;
  • Coverity in particular reveals many programming errors and maintainability issues

• Public league tables help motivate groups.
  • None of these will replace peer review or make up for a lack of education (another important topic) or experience.

• ATLAS will continue to use these tools, integrating the reports from different analyses, and keep an eye on new tools as they become