C++ software quality in the ATLAS experiment: tools and experience

C++ software quality in the ATLAS experiment: tools and experience

S Martin-Haugh[1], S Kluth[2], R Seuster[3], S Snyder[4], E Obreshkov[5], S Roe[6], P Sherwood[7] and G A Stewart[8]

[1] STFC – Rutherford Appleton Laboratory, U.K.
[2] Max Planck Institut für Physik (Werner Heisenberg Institut), Germany
[3] University of Victoria, Canada
[4] Brookhaven National Laboratory, U.S.A.
[5] University of Texas at Arlington, U.S.A
[6] CERN, Switzerland
[7] University College London, U.K.
[8] University of Glasgow, School of Physics and Astronomy, U.K.

E-mail: shaun.roe@cern.ch

Abstract. In this paper we explain how the C++ code quality is managed in ATLAS using a range of tools from compile-time through to run time testing and reflect on the substantial progress made in the last two years largely through the use of static analysis tools such as Coverity®, an industry-standard tool which enables quality comparison with general open source C++ code. Other available code analysis tools are also discussed, as is the role of unit testing with an example of how the GoogleTest framework can be applied to our codebase.

1. Introduction

The ATLAS Experiment [1] has developed its offline software based on the Gaudi framework [2]; this allows mainly C++ code components to be organized via Python job options into full simulation or data analysis jobs. The total codebase consists of roughly 3.8 million lines of code (LOC) in C++ and 1.4 million LOC in Python, and these are largely written by physicists with varying levels of coding experience. Currently there are approximately 420 developers spread over 140 teams, where each team covers a specific software domain (e.g. Inner detector tracking). Developers write code with a variety of tools and in different computing environments, are geographically distributed and generally write software only as part of a larger project or aim.

In such an heterogeneous development environment, code quality can be problematic. In the following discussion we introduce the tools which have been used or introduced over the last two years which facilitate the testing and (if necessary) correction of existing code, the promotion of good coding practices and overall to augment the awareness of coding quality issues in our C++ code.

2. Education

Our code developers come from a variety of educational backgrounds, and often they are students who have limited C++ experience but who are required to develop code in the context of their project. All newcomers are invited to take part in the software tutorials which take place two to three times per year and which teach general C++ skills as well as provide an introduction to the details of our specific framework. The tutorials are available online as TWiki [3] pages and are often followed independently of the formal classes.

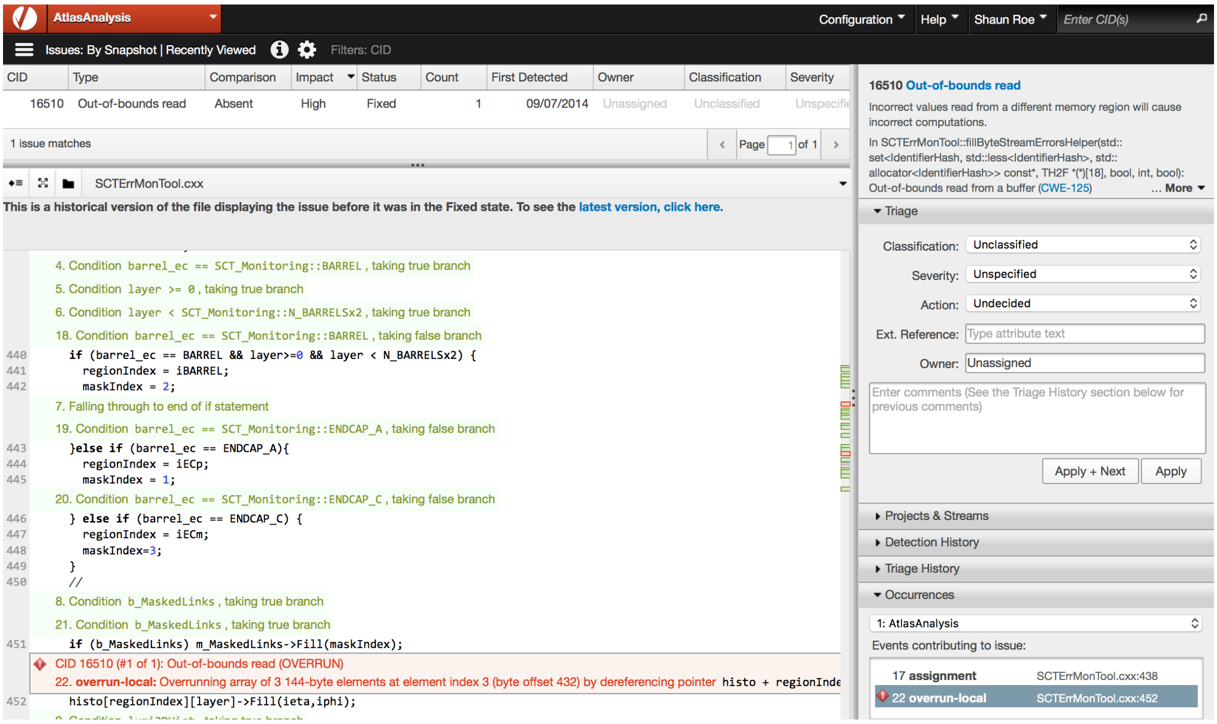
In addition to the tutorial and 'getting started' TWikis, there is a Software Quality page which aims to bring together resources (some of which are presented in this paper) to encourage awareness of code quality issues. The TWiki presents specific coding guidelines (including stylistic issues), books, web pages and software tools. For more advanced users, links are provided to code optimization resources.

3. Static Analysis

Static analysis refers to analysis of the written code by a software tool to detect code which, while perfectly legal C++, may be logically flawed or contain other defects which mean that the program is not a reflection of the developer's intent. A common such defect in C++ is the mismatch between new/delete resulting in a memory leak. As compilers increase in complexity, we see that they are able to detect and warn about an increasing number of developer errors. Independent static analysis tools are nonetheless still able to detect a wider range of such defects. Over the last two years we have mainly used two such tools: Coverity® [4], a proprietary tool which has become an industry standard; and CppCheck [5], a more lightweight open source tool. Both provide similar information, although the Coverity® tool is clearly more comprehensive in its coverage. Both tools can provide an XML summary of the defects detected, and typically we generate a display of defect numbers per software team in a similar format for both tools.

3.1. Coverity®

Coverity® is run centrally twice per week over the entire C++ codebase; it typically takes about 24 hours to run over all the projects, and the resulting defects are each assigned an individual identification number and entered into a database which allows historical tracking of the defect, even if its detailed line position in the code changes. The Coverity® static analysis program is integrated with a database and a web interface which allows statistics to be generated, defects to be assigned to developers and comments to be entered; it is also possible to declare a defect as a 'false positive' or 'intentional' and rank its severity, or tell Coverity® to ignore the defect. A typical defect display is shown below. It illustrates how Coverity® displays the logic leading up to the detection of a defect.



CID	Type	Comparison	Impact	Status	Count	First Detected	Owner	Classification	Severity
16510	Out-of-bounds read	Absent	High	Fixed	1	09/07/2014	Unassigned	Unclassified	Unspecifie

1 issue matches

SCTErrMonTool.cxx

This is a historical version of the file displaying the issue before it was in the Fixed state. To see the latest version, click here.

```
    4. Condition barrel_ec == SCT_Monitoring::BARREL , taking true branch
    5. Condition layer >= 0 , taking true branch
    6. Condition layer < SCT_Monitoring::N_BARRELSx2 , taking true branch
    18. Condition barrel_ec == SCT_Monitoring::BARREL , taking false branch
  if (barrel_ec == BARREL && layer>=0 && layer < N_BARRELSx2) {
    regionIndex = iBARREL;
    maskIndex = 2;
    7. Falling through to end of if statement
    19. Condition barrel_ec == SCT_Monitoring::ENDCAP_A , taking false branch
  }else if (barrel_ec == ENDCAP_A){
    regionIndex = iECp;
    maskIndex = 1;
    20. Condition barrel_ec == SCT_Monitoring::ENDCAP_C , taking false branch
  } else if (barrel_ec == ENDCAP_C) {
    regionIndex = iECm;
    maskIndex=3;
  }
  //
    8. Condition b_MaskedLinks , taking true branch
    21. Condition b_MaskedLinks , taking true branch
  if (b_MaskedLinks) m_MaskedLinks->Fill(maskIndex);
    CID 16510 (#1 of 1): Out-of-bounds read (OVERRUN)
    22. overrun-local: Overrunning array of 3 144-byte elements at element index 3 (byte offset 432) by dereferencing pointer histo + regionInde
  histo[regionIndex][layer]->Fill(ieta,iphi);
```

16510 Out-of-bounds read

Incorrect values read from a different memory region will cause incorrect computations.

In SCTErrMonTool::fillByteStreamErrorsHelper(std::set<IdentifierHash, std::less<IdentifierHash>, std::allocator<IdentifierHash>> const*, TH2F *(*)[18], bool, int, bool): Out-of-bounds read from a buffer (CWE-125) ... More

Triage

Classification: Unclassified

Severity: Unspecified

Action: Undecided

Ext. Reference: Type attribute text

Owner: Unassigned

Enter comments (See the Triage History section below for previous comments)

1: AtlasAnalysis

Events contributing to issue:

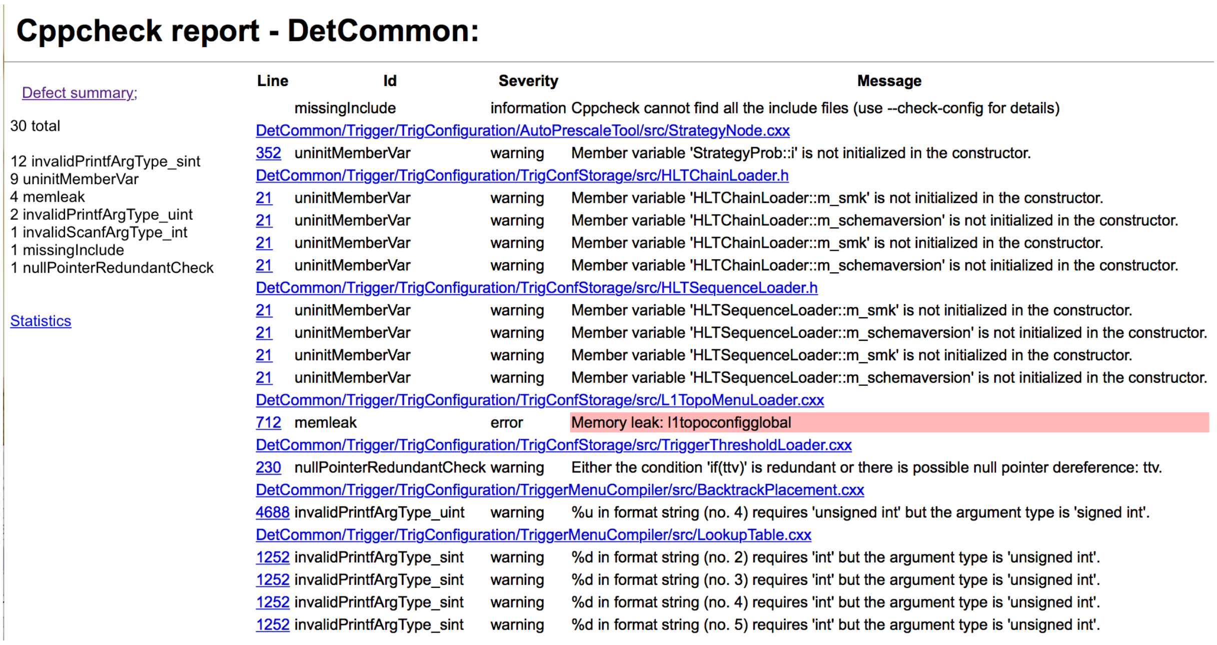
17 assignment	SCTErrMonTool.cxx:438
22 overrun-local	SCTErrMonTool.cxx:452

Figure 1: Web interface displaying Coverity® defect 16510

The Coverity database has a SOAP (Simple Object Access Protocol) interface allowing the extraction of defect information in XML format, however we have found that a simple one-line cURL command can also perform the same task, and our scripts generally use this method. The extracted XML is further processed by XSLT and Python scripts to generate emails to the developers and a web display ranking the number of defects per software team. A simple command line tool has also been introduced which queries the Coverity® results to produce a summary for an individual package, to be used by developers while they are in the process of editing code.

3.2. CppCheck

CppCheck is a much more lightweight program which can be run interactively over individual C++ files. It is also run centrally twice per week and typically takes about one hour to scan the entire codebase. The results are generated as a text file (we use the XML output option) and a web page with code highlights using the 'pygments' package [6], but the defects are not tracked week-to-week in the same way that Coverity® allows. While it is more prone to false positives than Coverity®, it *does* succeed in uncovering defects which Coverity® misses. The lightweight (and open source) nature of the program also means it can be run by individual developers as part of the development process, something which cannot be done by Coverity® due to both resource (it takes too long) and licensing issues.



Cppcheck report - DetCommon:

Defect summary;

30 total

12 invalidPrintfArgType_sint
9 uninitMemberVar
4 memleak
2 invalidPrintfArgType_uint
1 invalidScanfArgType_int
1 missingInclude
1 nullPointerRedundantCheck

Statistics

Line	Id	Severity	Message
	missingInclude	information	Cppcheck cannot find all the include files (use --check-config for details)
DetCommon/Trigger/TrigConfiguration/AutoPrescaleTool/src/StrategyNode.cxx			
352	uninitMemberVar	warning	Member variable 'StrategyProb::i' is not initialized in the constructor.
DetCommon/Trigger/TrigConfiguration/TrigConfStorage/src/HLTChainLoader.h			
21	uninitMemberVar	warning	Member variable 'HLTChainLoader::m_smk' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTChainLoader::m_schemaversion' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTChainLoader::m_smk' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTChainLoader::m_schemaversion' is not initialized in the constructor.
DetCommon/Trigger/TrigConfiguration/TrigConfStorage/src/HLTSequenceLoader.h			
21	uninitMemberVar	warning	Member variable 'HLTSequenceLoader::m_smk' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTSequenceLoader::m_schemaversion' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTSequenceLoader::m_smk' is not initialized in the constructor.
21	uninitMemberVar	warning	Member variable 'HLTSequenceLoader::m_schemaversion' is not initialized in the constructor.
DetCommon/Trigger/TrigConfiguration/TrigConfStorage/src/L1TopoMenuLoader.cxx			
712	memleak	error	Memory leak: l1topoconfigglobal
DetCommon/Trigger/TrigConfiguration/TrigConfStorage/src/TriggerThresholdLoader.cxx			
230	nullPointerRedundantCheck	warning	Either the condition 'if(ttv)' is redundant or there is possible null pointer dereference: ttv.
DetCommon/Trigger/TrigConfiguration/TriggerMenuCompiler/src/BacktrackPlacement.cxx			
4688	invalidPrintfArgType_uint	warning	%u in format string (no. 4) requires 'unsigned int' but the argument type is 'signed int'.
DetCommon/Trigger/TrigConfiguration/TriggerMenuCompiler/src/LookupTable.cxx			
1252	invalidPrintfArgType_sint	warning	%d in format string (no. 2) requires 'int' but the argument type is 'unsigned int'.
1252	invalidPrintfArgType_sint	warning	%d in format string (no. 3) requires 'int' but the argument type is 'unsigned int'.
1252	invalidPrintfArgType_sint	warning	%d in format string (no. 4) requires 'int' but the argument type is 'unsigned int'.
1252	invalidPrintfArgType_sint	warning	%d in format string (no. 5) requires 'int' but the argument type is 'unsigned int'.

Figure 2: CppCheck generated web page

3.3. Reporting and defect history

The results from the Coverity® scan are matched to the development teams and emails are sent once per week (typically on Monday) to the teams indicating what defects have been detected. The XML files from each scan are used to update a 'league table' showing the number of defects per team in the format shown in figure 3a. This competitive comparison was also initially seen as a motivation for the teams to reduce the number of defects assigned to them. The absolute numbers of defects were also recorded, and the historical trend is shown in figure 3b. Note that the switching of compiler versions and subsequent upgrade to the Coverity® package introduced a significant disruption to our defect detection, and during this time we relied more on CppCheck.

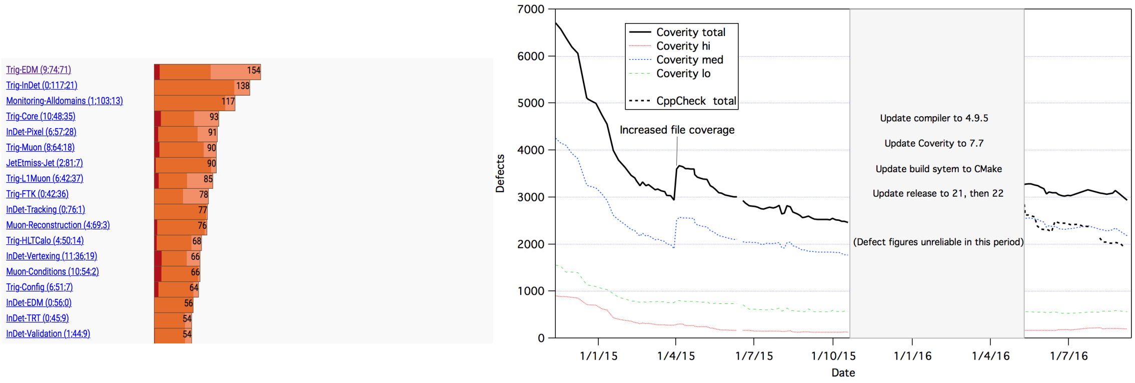


Figure 3 (a) Typical 'league table' of defects per team, and (b) Historical trend of defects in the project

3.4. Other tools and code metrics

Various other open-source C++ - checking tools were investigated by the software quality group, including OCLint [7], Include-What-You-Use [8] and Lizard [9].

OCLint seems to be a promising suite, being able to detect stylistic issues and 'code smells' such as long sections of cut-and-pasted code between compilation units; unfortunately it underwent a long hiatus in its support from 2013-2016 but now looks worthy of further investigation.

Include-What-You-Use is much more focussed in its aim: it helps to tidy up '#include' statements and ensure that forward declarations are used where possible; this is beneficial in terms of compilation times and dependencies between packages.

Lizard is a tool providing code metrics such as cyclomatic complexity (CC) and LOC per function; it helps to identify 'code smells' and when applied to our code it certainly helped to find egregious examples of unreadable code, however measures such as CC have never been unambiguously correlated with defective or unmaintainable code and it seems to highlight code worthy of review rather than explicitly defective code. The CC remains a good measure of the number of test cases necessary to thoroughly test a piece of code.

4. Compile and Run-Time Defect Detection

Compilers differ in their defect detection capabilities, and it has been clearly beneficial to expose our code to a variety of compilers (e.g. Clang and different versions of gcc) during the development process. We are aided in this by the use of a nightly build system which builds the entire code with different compilers, on different platforms and in both debug and optimized builds. An automated reporting process notifies developers in the case of a build failure.

Our usual compiler in this period was gcc49, and gcc plug-ins were also written which identified certain violations of our coding guidelines, such as proscribed inheritance structures or naming conventions. Additional tools that were investigated include the 'Undefined Behaviour Sanitizer'[10], the 'Address Sanitizer' [11] and 'Thread Sanitizer' [12]; of these, the Undefined Behaviour Sanitizer was retained and is used regularly in debug builds. It gives clear error messages at run time when undefined behavior is detected (such as left-shift of a negative number).

4.1. Unit tests and Run Time Testing (RTT)

Unit tests have long existed in our code as part of the core software, but very few subsystem developers have written unit tests routinely as part of their development process. One of the perceived barriers to comprehensive unit testing is the fact that in many cases, complex objects (e.g. a Track) are required as input to a given method in order to test it. Over the last year, the GoogleMock/GoogleTest

framework [13] has been introduced and explored as a means to overcome that complexity by introducing dummy or ‘mock’ objects with interfaces that allow unit tests to be written more easily.
In the example below, a complex ‘Jet’ object is mocked to provide dummy calls to pt and eta, the two sections illustrating the header and implementation.

```
#ifndef GMOCKDEMO_XAODJET_JET_H
#define GMOCKDEMO_XAODJET_JET_H
// GTest/GMock include(s):
#include <gmock/gmock.h>
#include "allNecessaryIncludes.h"
namespace xAOD {
   /// Mock class for xAOD::Jet
   class Jet : public virtual IParticle {
   public:
      /// Constructor, setting how the object should behave:
      Jet();
      MOCK_CONST_METHOD0( pt, double() );
      MOCK_CONST_METHOD0( eta, double() );
      /// The cached 4-momentum of the mock object
      TLorentzVector m_p4;
   };
}
#endif // GMOCKDEMO_XAODJET_JET_H

// Local include(s):
#include "xAODJet/Jet.h"
#include "utilities.h"
namespace xAOD {
   Jet::Jet() {
      // Generate a random 4-momentum for the jet:
      const double pt  = randomPt();
      const double eta = randomEta();
      const double phi = randomPhi();
      const double m   = randomMass();
      m_p4.SetPtEtaPhiM( pt, eta, phi, m );
      // Set up the IParticle functions to return these properties:
      EXPECT_CALL( *this, pt() )
         .WillRepeatedly( testing::Invoke( [this]() -> double {
                  return this->m_p4.Pt();
               } ) );
      EXPECT_CALL( *this, eta() )
         .WillRepeatedly( testing::Invoke( [this]() -> double {
                  return this->m_p4.Eta();
               } ) );
    }

} // namespace xAOD
```

Figure 4: Code example of a mock ‘Jet’ object (simplified)

The Google test framework in addition implements a simple and uniform reporting mechanism that can be incorporated into the build process. The initial uptake by developers has been slow, but continues to grow.
‘Run Time Testing’(RTT) in our context refers to more holistic tests in which plots of physics quantities may be produced and compared from build to build, based upon an input reference dataset. The outputs are compared to detect possible changes which may impact physics results, and are used to qualify both infrastructure changes (operating system, processor, compiler) and detailed software changes. The RTT specification is an XML file in the package detailing what job is to be run and the

method of comparison, and is run nightly as an appendix to the build process; developers are similarly notified when a test fails.
Our code repository is soon expected to move to Git [14] and code committal will include a review process; build and unit testing is expected to take on increased importance as part of this review.

5. Documentation and Formatting

We encourage developers to use Doxygen [15] to document their code; this is an easy-to-use but quite flexible form of markup in the code, and allows automatic extraction to a searchable web interface which can also generate dependency and class diagrams. The documentation stays close to the code (as opposed to being in a separate file) so is less likely to be out-of-date.

We have not mandated a fixed format for code in terms of indentation or brace styles, but have rather requested that developers maintain a consistent style within their own package. This is difficult to enforce so we have also made the 'uncrustify' tool available, which is a post-edit reformatting tool with an easily understood configuration file. While generally very good, we have discovered a few examples where this tool has produced erroneous changes and we continue to explore other options.

6. Social Issues

Reports from the various tools mentioned here are available as web pages and linked from the software quality page, however these are not regularly consulted by developers. Developers are under time constraint to produce functional code and the extra work implied by quality issues (e.g. initializing all variables, providing documentation) often takes second place. Notifications of specific errors by email to the developers have more success, as witnessed by the reduction in the number of Coverity® defects (figure 3) over the last two years. It is also noticeable that the introduction of *any* new tool which promotes discussion of code quality has the effect, at least temporarily, of increasing the effort invested in quality issues. It is therefore important to maintain interest and promote discussion and awareness on a continuous basis by regular presentations to the community.

7. Summary and Plans

Over the last two years, the ATLAS Software Quality Group has investigated various tools which aid developers to improve software quality. The static analysis tool Coverity® has proved to be extremely useful, and has resulted in many defects being found and fixed, as shown in figure 3b. Other static analysis tools have also been successfully introduced and some (CppCheck) continue to be regularly used. Our code is more stable as a result, and in this period (for example) over 670 resource leaks were identified and fixed. However, these tools are only useful in checking individual lines and sections of code; their use does not necessarily mean that the code is 'good'. 'Code smell' detectors such as Lizard and OCLint help to identify code which should be reviewed in terms of its overall design. Other tools which are now in regular use include the run-time sanitizer UBSan (detecting undefined behavior) and uncrustify (for formatting). Unit testing in the GoogleTest framework and using GoogleMock objects has been successfully introduced and the usage is expected to grow.

The ATLAS software repository will move to Git in 2017, requiring the use of routine code review as part of the code committal process; quality testing and metrics are expected to become an integral part of the review process, and we expect to continue to research and promote use of such tools in our environment.

References

[1] The ATLAS Collaboration 2008 *Journal of Instrumentation* **3** S08003
[2] Barrand G et al. 2001 *Comput. Phys. Commun.* **140** 45-55
[3] TWiki – The Open Source Enterprise Wiki: http://www.twiki.org
[4] Software Testing and Static Analysis Tools (Synopsis): http://www.coverity.com
[5] Cppcheck – A tool for static C/C++ code analysis: http://cppcheck.sourceforge.net
[6] Pygments: http://pygments.org
[7] OCLint: http://oclint.org
[8] Include-What-You-Use: https://include-what-you-use.org
[9] Terryin/lizard, A simple code complexity analyser: https://github.com/terryyin/lizard
[10] GCC Undefined Behaviour Sanitizer: https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan/
[11] Address Sanitizer: https://github.com/google/sanitizers/wiki/AddressSanitizer
[12] Thread Sanitizer: https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
[13] Google/Googletest: https://github.com/google/googletest
[14] Git: https://git-scm.com
[15] Doxygen: http://www.stack.nl/~dimitri/doxygen/