An evaluation of tools for static checking of C++ code

E. Arderiu Ribera[1], *G. Cosmo*[1], *S. M. Fisher*[2], *S. Paoli*[1,3], *M. Stavrianakou*[1]

[1] CERN, Geneva, Switzerland
[2] RAL, UK
[3] Contact person

Abstract

The SPIDER-CCU (Code Check Utility) project was conducted at CERN by the IT-IPT group in collaboration with members of LHC experiments and IT division. Its purpose was to define a common C++ coding standard, and a tool to automatically check code against it.

After releasing the document "SPIDER - C++ Coding Standard" in August 1999 an evaluation of available commercial tools was conducted. Five commercial tools were evaluated over a period of three months. The evaluation was based on a process, and a set of criteria, defined in order to take into account all the needs of CERN and the LHC experiments.

The focus of the presentation will be on the method followed for the tools evaluation, and the results of the work.

Keywords: tools, C++, evaluation

1 Introduction

Static checking of code against an accepted standard will enable C++ developers in the HEP community to program better and more efficiently. This is especially important given that the majority of HEP developers are not professional programmers and the complexity of C++ as a programming language can only be mastered after months if not years of active coding.

The C++ coding standard [1] has been agreed upon by representatives of several major experiments and IT projects and includes 108 rules (naming, coding and stylistic). The aim of the subsequent evaluation was the identification of a suitable tool for the automatic checking of C++ against the standard.

Although quite frequently static analysis tools offer more QA functionality than mere rule-checking, the scope of the evaluation was limited to the latter, so as to allow meaningful and detailed comparisons among the different tools.

2 Evaluation of tools

2.1 Approach and selection of tools

In order to ensure meaningful and reliable results and to facilitate a future choice of a tool suitable for CERN needs, a pragmatic approach for the evaluation process was adopted. Potential users of the tool were involved in the definition of the evaluation criteria and the planning of the evaluation as well as the actual technical evaluations. To save time and effort a preselection on just technical merit was performed.

The candidate tools (see Table I) were selected based on all available information and taking into account time and resource constraints. Our first source was the OVUM Report [2] from which we selected three tools (Concerto/AuditC++, TestBed and QA C++) that were highly rated in the rule-checking section. CodeWizard and CodeCheck were selected based on prior experience in the team.

Table I: Evaluated tools

Tool	Vendor
CodeCheck 8.01 B1	Abraxas
QA C++ 3.1	Programming Research Ltd.
CodeWizard 3.0	Parasoft
Logiscope RuleChecker (Concerto/AuditC++) 3.5	CS Verilog S.A.
TestBed 5.8.4	LDRA Ltd.

Two other tools, namely the latest release of the Together/Enterprise CASE tool and a first prototype of a new coding rule check tool developed by a collaboration between ITC-IRST and the ALICE experiment were seen to have some promising features but were not yet in a state to be evaluated.

The detailed evaluation report is available from [3].

2.2 Evaluation environment and criteria

For the evaluation of the tools in terms of usability and reliability, real and representative C++ code produced in HEP was used. The main components were the GEANT4 toolkit for the simulation of HEP detectors and the passage of particles through matter, the “Event” package for the ATLAS experiment and an ATLAS C++ utility library known as “classlib”. The GEANT4 toolkit, which is publicly available and widely used in the HEP community, consists of more than 1 MLOC in more than 2000 source files, written by a large and varied community of developers, in different styles and levels of expertise. The ATLAS packages were chosen because, in addition to their complexity, they were well known to two members of the evaluation team.

The evaluation criteria have been organised in three main groups: operational, technical and managerial.

The operational aspects cover the issues of installation, deployment and upgrade of a centrally supported tool. The managerial aspects cover the issues of license and maintenance costs and vendor information.

The technical criteria cover all aspects related to the use of the tool as experienced by the end user:

- Coverage of standard: how many rules of the adopted standard can be checked by the shipped product and how many are potentially checkable if the tool allows the addition of new checks
- Addition of customised checks and level of difficulty to do that
- Other possible checks: does the tool provide other configured checks and what is their relevance
- Support of ANSI C++: does the tool support checking against the ANSI C++ standard
- Support of template libraries and in particular STL
- Robustness: does the tool correctly parse a large in good status or a complex and difficult to parse package
- Reliability: is the detection of item violations reliable
- Usability: how easy is it to learn and use the tool (learning time, tool interfaces, tool reports)
- Customisability. Is it possible to:
  - configure the tool to run on a package and save this configuration for later use

  - configure and use the tool reports e.g. sort warnings according to severity
  - only analyse parts of a package (selected set of files) and exclude external packages or parts of the package tree structure
  - integrate the tool in IDEs and configuration management systems
  - include or exclude selected checks
  - define or override the severity of the violations/warnings
  - parse a header file only once
- Performance: what is the time required to analyse a whole large program and a medium size program

Other criteria such as the quality and quantity of documentation in electronic and paper format, the quality of the tool WWW site and the quality of available support were also taken into account.

2.3 Evaluation results

Evaluation results are given mainly in terms of technical aspects for all five tools. Of these, only two, CodeWizard and QA C++, were preselected and fully evaluated.

CodeCheck had already been used and evaluated extensively and had shown severe problems in parsing real code making extensive use of template libraries like STL. Moreover, no enhancements in that direction were envisaged by the company. Other serious limitations included the amount of effort required for customisation and the implementation of new rules, as well as the absence of a license server or any way to monitor the tool usage. The tool was excluded from further evaluation.

Logiscope RuleChecker was found to be simple, easy to use and fast. However, the number of rules supplied and the possibility to add new rules were limited and the use of the proprietary language CQL made it difficult to exploit the flexibility offered in terms of report generation and quality. Therefore the tool was excluded from further evaluation.

TestBed was able to parse most of the code it was tested on but failed in the case of tricky code. The number of useful built-in checks was limited, there was no possibility of adding new checks and the report format was found rather poor. This tool was also excluded from further evaluation.

CodeWizard comes with at least 71 (vendor quote: 120) checks implemented, including most of the items described in S. Meyers books (Effective C++ and More Effective C++), some more items described in articles by S. Meyers and M. Klaus and other items referred to as Universal Coding Standards. As shipped, the tool was estimated to cover 24% (26/108 items) of the standard and would be configurable to cover 71% (77/108 items). Although no explicit checks against the ANSI C++ are performed, no serious problems were observed in parsing ANSI C++ code. Parsing code that used template libraries and in particular STL showed no problem whatsoever. The RuleWizard tool added in the recent release can in principle be used for adding customised checks (for naming and semantics but not yet layout) via a graphical interface. Although this utility seems flexible and powerful, it is at the moment insufficiently documented and in practice unusable. The reliability was specifically tested against 18 checks and was generally found to be good. The tool was found to be very robust (no crashes or undefined behaviour). Tool reports can be obtained in graphical and ASCII format of which the latter seems more efficient. Regarding usability, the tool was found easy to learn and use (with the exception of the RuleWizard). One annoying feature, however, is the repetition of parsing and error reporting for header files included from several sources. The tool works on single files and requires information for headers and libraries (-I and -D options) to be used. By using the makefile instead, any package can be analysed in a very straightforward way. It is possible to exclude specific parts of code and switch off and on individual checks. However, customisation of the reports is not really possible. Analysis of the GEANT4 Processes subpackage of the benchmark software has shown performance equivalent to that of the compiler.

QA C++ comes with about 500 (vendor quote: 650) checks implemented, a lot of which cover compliance to ISO C++. As shipped, the tool was estimated to cover 44% (48/108 items) of the adopted standard and would be configurable to cover 65% (70/108 items). The current product does not support STL; however, the STL stubs provided by the company allow partial analysis of code using STL and full support is foreseen for the next release. No specific reliability checking in terms of violation identification was performed. Extensive metric calculations and corresponding reports are available but were not in the scope of this evaluation. The tool is robust in parsing mostly successfully real code. The tool requires information for headers and libraries (-I and -D options) and provides a rather delicate configuration for it; integration with the makefile is not straightforward either. Apart from this, the tool is easy to learn and use. It provides a powerful GUI and a command line interface which are largely interchangeable. High quality, customisable reports can be obtained and displayed in various ways. The tool is also highly customisable in terms of inclusion/exclusion of code and rules and can be instructed to parse only once and cache header files. Performance is a weak point of the tool. Analysis of the GEANT4 Processes subpackage of the benchmark software has shown a factor of 2 slower performance compared to the compiler. It should be noted that the company is releasing a completely new version of the tool that was not yet available for our evaluation. Full ANSI C++ compliance including support for STL and an improved parser are envisaged.

3 Conclusions

The evaluation process devised for this project was found to be suited to the goals, pragmatic and efficient. The user involvement, the detailed planning of the two-phased approach and the special care taken in the definition of the evaluation criteria were the major factors that facilitated the completion of the project. Of the five tools initially considered, two, CodeWizard and QA C++, were preselected based on technical merits and were fully evaluated. The final choice would depend on the weight given to the various features of each tool, the relative cost, the needs of the institutes concerned and possibly the state of development of promising new tools.

References

1 S. Paoli, P. Binko, D. Burckhart, S.M. Fisher, I. Hrivnacova, M. Lamanna, M. Stavrianakou, H.-P. Wellisch “C++ Coding Standard - Specification”, CERN-UCO/1999/207, 20 October 1999.

2 OVUM Evaluates: Software Testing Tools, 1999, Ovum Ldt.

3 S. Paoli, E. Arderiu-Ribera, G. Cosmo, S.M. Fisher, A. Khodabandeh, G. H. Pawlitzek, M. Stavrianakou, “C++ Coding Standard - Check Tools Evaluation Report”, CERN, 17 December 1999 (restricted access, for availability please contact CERN IT-PST)