TOOLS AND RULES TO ENCOURAGE QUALITY FOR C/C++ SOFTWARE

Katarina Sigerud, Wojciech Sliwinski, Vito Baggiolini, Jean-Claude Bau, Stephane Deghaye, Jeremy Nguyen Xuan, Xavier Piroux, Gennady Sivatskiy, Ilia Yastrebov, CERN, Geneva, Switzerland

Abstract

Inspired by the success of the software improvement process for Java projects, in place since several years in the CERN accelerator Controls group, it was agreed in 2011 to apply the same principles to the C/C++ software developed in the group, an initiative we call the Software Improvement Process for C/C++ software (SIP4C/C++). This paper will present the SIP4C/C++ initiative in more detail, summarizing our experience and the future plans.

BACKGROUND

In view of improving the quality and integrity of the products released in operations, the CERN accelerator Controls group decided in 2009 to apply a systematic approach to quality assurance (QA). The aim was to introduce QA activities as an integral part of the development cycle and to standardize and unify between the projects with regards to deliverables, deployment and release procedures. We call this initiative SIP, the Software Improvement Process and it was first applied for the Java projects in the group [1].

The C/C++ software in the group is developed by several projects in separate sections. Most of the projects were already applying some quality assurance techniques but there was no common effort in their approaches and several aspects of quality assurance were not addressed, e.g. static code analysis, unit testing or continuous integration.

Inspired by the success for the Java software, it was agreed in 2011 to apply the same principles to the C/C++ software developed in the group, an initiative we call the Software Improvement Process for C/C++ software (SIP4C/C++).

OBJECTIVES

The objectives of the SIP4C/C++ initiative are: 1) agree on and establish best software quality practices, 2) choose tools for quality, and 3) integrate these tools in the software development process.

RESULTS

After a year we have reached a number of concrete results. In the areas of standard quality assurance practices like unit testing, static code analysis and continuous integration, we have investigated the available tools and agreed on a common set of tools. In addition to this, we have implemented manifest file generation with dependency information and runtime in-process metrics. To automate the use of these tools, we have implemented a common build tool based on GNU Make, which standardizes the way to build, test and release the C/C++ binaries, libraries and executables.

A Common Build Tool

All software projects participating in the SIP4C/C++ initiative have joined forces and agreed to share a common software build and release process based on a common Makefile. This Makefile, called Make.generic, contains a common set of targets used by all projects. Besides the fundamental ones, it has targets to run the unit tests, to run Valgrind memory profiler, to build the demo programs, to generate code documentation, to produce a Manifest file as described in the next section, to create a SVN tag or branch and to deploy the binaries into the binary repository. Make.generic also standardizes certain configurations: it defines a set of compilation flags agreed on by the SIP4C/C++ members; it enforces a common directory structure for both the source code repository and for the binary repository; a common naming convention and versioning scheme for all released products.

The Manifest

In our software development process, we want to include certain build-time information into a binary, namely its name, version, build time, creator, compiler, OS, CPU architecture, etc. We call this information a “manifest”, inspired by the Java manifest. The manifest must be easy to retrieve at runtime using the CMX API (see the chapter related to the runtime metrics). It should also be retrievable without having to execute a binary and for binaries that cannot be executed e.g. libraries. Furthermore, if an executable contains several statically linked libraries, the manifest of all those libraries must be contained in the resulting executable. This is vital e.g. for troubleshooting scenarios, where we need to examine the versions of all libraries used to build the resulting executable. Before we started using manifests, the only information available about a given binary was contained in the file name and the file system location where it was stored (file name, creation time, uid of the creator). This solution was much less robust and powerful, because the information got lost if a library was copied to another location or linked into an executable. Sample manifest information is shown in Figure 1 below.

Copyright © 2014 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-139-7

```
[          ] /user/                    /cmw-proxy-cpp/build/bin/L866 > ident cmw-proxy
cmw-proxy:
     $Manifest: product=cmw-proxy;version=2.1.8;build_date=2013-08-14 15:42:20;user=        ;c
ompiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866
;executable=true $
     $Manifest: product=cmw-log-stomp;version=1.5.0;build_date=2013-05-27 14:49:45;user=
ns;compiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=
L866 $
     $Manifest: product=cmw-stomp;version=2.1.0;build_date=2013-05-27 14:39:04;user=        ;c
ompiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866
 $
     $Manifest: product=cmw-util;version=1.5.0;build_date=2013-05-27 13:47:38;user=        ;co
mpiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866
$
     $Manifest: product=cmw-log;version=2.7.6;build_date=2013-05-27 14:03:24;user=        ;com
piler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866 $
     $Manifest: product=cmw-rbac;version=5.3.0;build_date=2013-05-28 15:23:41;user=        ;co
mpiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866
$
     $Manifest: product=cmw-rda;version=2.12.1;build_date=2013-05-28 15:56:28;user=        ;co
mpiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_64;cpu=L866
$
     $Manifest: product=cmw-directory-client;version=1.4.0;build_date=2013-05-27 16:20:20;user
=        ;compiler=g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3);os=Linux 2.6.32-358.2.1.el6.x86_
64;cpu=L866 $
```

Figure 1: Example Manifest as extracted by `ident`.

Our Manifest implementation is based on the RCS identity keywords and the Linux `ident` command. This tool relies on putting special strings into the binary. Make.generic generates two files for this purpose: a header file with constants used to expose the manifest information at run-time, and a simple text file that is attached to the end of the binary using the `objcopy` utility. The latter approach was necessary for libraries, because the compiler interpreted the manifest variables as "unused code" and removed it.

Unit Testing, Mocking and CI

Most of the C/C++ projects applied some testing policy to their software, however as the tests could be heavy to run, and there was little or no automation, they were not always run before committing the changes and releasing the product.

As for the Java software, we wanted to profit from the benefits of unit testing for the C/C++ software, therefore we investigated the available frameworks for the C/C++ software that could facilitate testing. The different frameworks were evaluated based on criteria that were grouped into fundamental, mandatory and desirable features. Based on this evaluation, we decided on the Google Test framework [2]. The example output from running unit tests is shown in Figure 2 below.

```
[ RUN      ] TestAccessRulesMap.TestAuthorization20Roles
[       OK ] TestAccessRulesMap.TestAuthorization20Roles (28 ms)
[ RUN      ] TestAccessRulesMap.TestAccessMapSize1
[       OK ] TestAccessRulesMap.TestAccessMapSize1 (22 ms)
[ RUN      ] TestAccessRulesMap.TestAccessMapSize2000
[       OK ] TestAccessRulesMap.TestAccessMapSize2000 (50 ms)
[ RUN      ] TestAccessRulesMap.TestRoles1
[       OK ] TestAccessRulesMap.TestRoles1 (19 ms)
[ RUN      ] TestAccessRulesMap.TestRoles20
[       OK ] TestAccessRulesMap.TestRoles20 (28 ms)
[----------] 13 tests from TestAccessRulesMap (168 ms total)

[----------] Global test environment tear-down
[==========] 45 tests from 10 test cases ran. (11030 ms total)
[  PASSED  ] 36 tests.
[  FAILED  ] 9 tests, listed below:
[  FAILED  ] TestExpiration.TestDefaultLifetime
[  FAILED  ] TestExpiration.TestDefaultMasterLifetime
[  FAILED  ] TestExpiration.TestCustomLifetime
[  FAILED  ] TestExpiration.TestCustomMasterLifetime
[  FAILED  ] TestLocationLogin.TestLocationLogin
[  FAILED  ] TestRolePicker.TestRolePicker
[  FAILED  ] TestExplicitLogin.TestBasic
[  FAILED  ] TestExplicitLogin.TestAdvanced
[  FAILED  ] TestRelogin.TestRelogin

 9 FAILED TESTS
```

Figure 2: Example output from running C++ unit tests written using the Google Test framework.

To be able to fully isolate the class being tested, it is important to mock out functionality in adjacent classes. For this purpose, we agreed to use the Google Mock framework [3].

For the Java projects, we are relying on a Continuous Integration (CI) server (Atlassian Bamboo [4]) for early detection of problems, especially between dependent projects. With unit tests available also for the C/C++ projects, we wanted to gain the same benefits. As running of the tests is integrated with the common build tool, they can be triggered from our CI server, after committing the changes to the source code repository. Next, a build report is generated and notification is sent to the responsible in case of build or test failures. See Figure 3.

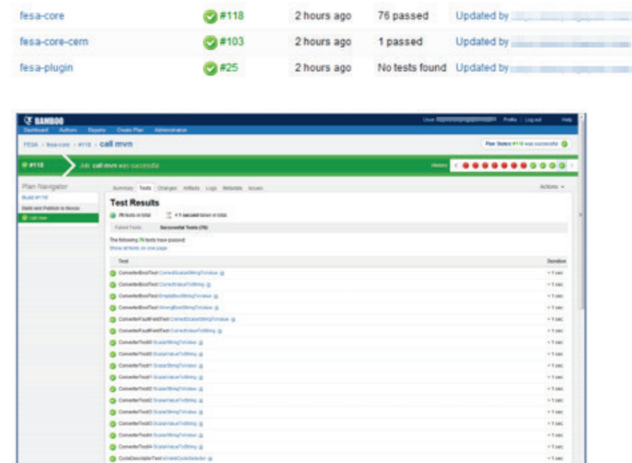

Figure 3: Test results in the CI server.

Static Code Analysis

Code reviews by peers are very beneficial but also very time consuming, therefore most projects can only afford to apply this kind of review to the most complex parts of the codebase. Instead, we could benefit from running static code analysis tools over all our codebase to automatically spot the most common mistakes and bug patterns. For C/C++ projects, we are using the Coverity tool [5]. The experience so far is that it can detect many common programming mistakes, so any project could benefit from performing a regular code analysis, Fig. 4.

```
78 //-----------------------------------------------------------------------
79
80 string StringUtils::toString(float number, size_t digitsAfterDot)
81 {
82     char buf[50];
CID 10635: Calling risky function (SECURE_CODING)
[VERY RISKY]. Using "sprintf" can cause a buffer overflow when done incorrectly. Because sprintf()
assumes an arbitrarily long string, callers must be careful not to overflow the actual space of the
destination. Use snprintf() instead, or correct precision specifiers.
83     sprintf(buf, "%.*f", static_cast<int>(digitsAfterDot), number);
84     return buf;
85 }
86
87 //-----------------------------------------------------------------------
88
89 string StringUtils::toString(double number, size_t digitsAfterDot)
90 {
91     char buf[360];
CID 10634: Calling risky function (SECURE_CODING) [select defect]
92     sprintf(buf, "%.*f", static_cast<int>(digitsAfterDot), number);
93     return buf;
94 }
95
```

Figure 4: Example Coverity report with highlighted code defect.

Copyright © 2014 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-139-7

Another type of static code analysis for C/C++ projects is the compile options. As it is a good practice to keep the same flags across the projects depending on each other, we have agreed on a set of the compile options common to all C/C++ projects in the group. These options are integrated with the Make.generic build tool.

It must be noted that compiler options, Coverity and other static code analysis tools must be carefully configured. It is often even necessary to fine-tune the analysis rules for specific projects. Without that, these tools will flood the developer with hundreds of warning messages, many of which even turn out to be "false-positives". Without this configuration effort static analysis tools can even be counter-productive.

Runtime In-process Metrics

The knowledge of the internal, runtime state of the operational processes is essential for problem diagnostics as well as for constant monitoring for pre-failure recognition. The CMX library follows similar principles as JMX (the Java Management Extensions) and it provides similar monitoring capabilities for C/C++ applications. It was implemented as a lightweight C/C++ library, providing a sub-set of JMX's extensive functionality. It allows registering and exposing runtime information as simple counters, floating point numbers or character data that can subsequently be used by external diagnostics tools for checking thresholds, sending alerts or trending. CMX uses shared-memory technology to ensure non-blocking read/update actions, which is an important requirement for real-time processes. CMX was integrated with DIAMON [6] - CERN's Diagnostic and Monitoring system. Detailed CMX architecture, design and characteristics are outlined in a separate paper [7].

CHALLENGES

The main challenge for the SIP4C/C++ working group was to agree on common standards and tools. In most cases the involved projects already had well-established routines and tools, especially in the area of build and deploy. Thanks to the collaborative spirit of all parties involved, we have already achieved the concrete results described above and the projects have made the necessary changes to adhere to what has been agreed.

Another challenge has been to identify appropriate tools. With our current criteria for choosing tools (open-source, easy to use, active developer community, good documentation), the choice is quite limited.

FUTURE PLANS

The next step for the SIP4C/C++ initiative will be to draw conclusions from the early adopters of the Coverity tool and decide if its use should be extended to all C/C++ projects in the group.

All projects should be encouraged to use the CMX library for exposure of runtime, in-process metrics and easy integration with the DIAMON monitoring system.

As seen for the Java projects, the unit test coverage is an important metric to encourage developers to do quality assurance. The SIP4C/C++ working group should identify and agree on a common tool to facilitate the code coverage analysis. An initial investigation has been done but no conclusions have been drawn yet.

REFERENCES

[1] K. Sigerud et al, "The Software Improvement Process – Tools and Rules to Encourage Quality", ICALEPCS'11, Grenoble, France, October 2011, THBHMUST04, p. 1212 (2011); http://accelconf.web.cern.ch/AccelConf/icalepcs2011/papers/proceedings.pdf.

[2] GoogleTest: https://code.google.com/p/googletest/

[3] GoogleMock: https://code.google.com/p/googlemock/

[4] Bamboo: http://www.atlassian.com/software/bamboo

[5] Coverity: http://www.coverity.com/products/coverity-save.html

[6] W. Buczak et al, "DIAMON2 – Improved Monitoring of CERN's Accelerator Controls Infrastructure", ICALEPCS'13, San Francisco, CA, USA, October 2013.

[7] F. Ehm et al, "CMX - A Generic In-Process Monitoring Solution for C and C++ Applications", ICALEPCS'13, San Francisco, CA, USA, October 2013.

ISBN 978-3-95450-139-7

Copyright © 2014 CC-BY-3.0 and by the respective authors