ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA161

SIP4C/C++ AT CERN – STATUS AND LESSONS LEARNED

S. Jensen, J-C Bau, A. Dworak, M. Gourber-Pace, F. Hoguin, J. Lauener,
F. Locci, K. Sigerud, W. Sliwinski, CERN, Geneva, Switzerland

Abstract

After 4 years of promoting the Software Improvement Process for C/C++ (SIP4C/C++) initiative at CERN, we describe the current status for tools and procedures along with how they have been integrated into our environment. Based on feedback from four project teams, we present reasons for and against their adoption. Finally, we show how SIP4C/C++ has improved development and delivery processes as well as the first-line support of delivered products.

BACKGROUND

A C/C++ software improvement process (SIP4C/C++) has been promoted in the CERN Accelerator Controls group since 2011, addressing technical and cultural aspects of our software development work. A first paper was presented at ICALEPCS 2013 [1]. On the technical side, a number of off-the-shelf software products have been deployed and integrated, including Atlassian *Fisheye/Crucible* (code review), *Google test* and *Google mock* (unit test), *Valgrind* (memory debugging/profiling) and *SonarQube* (static code analysis). Likewise, certain in-house developments are now operational such as a generic Makefile, *Makefile.generic,* (compile/link/deploy), *CMX* (for publishing runtime process metrics) and *Manifest* (capturing library dependencies). In addition, SIP4C/C++ has influenced our culture by promoting integration of said products into our binaries and workflows.

Four projects have adopted SIP4C/C++ to various degrees:

CMW delivers C and C++ libraries providing transport facilities as extendible classes, letting the user create remotely accessible servers which expose data according to the device/property model employed in our controls system.

FESA a C++ framework based on CMW libraries, formalizing the creation of device/property-based servers.

SILECS resembles FESA, but focuses on letting users expose PLC data according to the device/property model.

TIMING provides and operates a number of executables used to sequence and synchronise CERN's accelerator complex, along with libraries for other developers to use.

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 into the software development process.

RESULTS

For each participating project, the SIP4C/C++ products and procedures mentioned above were evaluated in terms of uptake as either "Strong", "Medium" and "Weak". Reasons for and against adoption were collected along with suggestions for future improvements.

Common Build Tool

Status: The Makefile.generic is stable. Includes targets for compiling, linking, SVN commits with support for tags and branching, deploying, documentation, test execution and launch of the Valgrind memory debugger/profiler. Also, provides automatic generation of the Manifest (see below).

Uptake: Strong in all four projects, with all products managed using Makefile.generic

Pros: Essential for uniform approach to release management and testing, which in turn facilitates cross-project development teams. Once adopted, it greatly simplifies integration of new target platforms. It meets the requirements of many users and as it was implemented in-house, we can readily adapt it to future needs and new platforms.

Cons: Approaches the limits for what Make is intended for. Its complexity makes it hard to know what is possible and how to achieve it – in particular for inexperienced users. It was very time consuming to adopt and hence best suited for projects of a substantial size where the effort was found to be worthwhile. Projects risk losing time on refactoring if the Make system changes. Our implementation is non-standard and hence requires dedicated resources for evolution. The current solution depends on remote resources, i.e. network (NFS) access is required.

Future: Moving towards a higher-level service based on standard, open-source products (e.g. *cmake*) with support for dependency management would help decrease confusion and errors. There should be support for working offline, i.e. to download all resources once.

The Manifest, Dependency Capture

Status: Stable, with a few known issues, in particular problems in correctly navigating symbolic links. Dependency information is captured as XML at build-time, via Makefile.generic, and visualized as shown in Figure 1.

Uptake: Medium in FESA and TIMING, weak in CMW and SILECS. FESA uses the manifest ad hoc to manually spot end-user dependency conflicts, whereas TIMING parses the XML to automatically configure source directory paths for *gdb*. CMW saw little interest, finding the dependency information of committed Makefiles to be more reliable.

Pros: Provides useful information in case of certain unexplained run-time behaviours and it is effortless to use due to integration with Makefile.generic

Cons: The solution is non-standard. A system like *pkg-config* could be interesting, but would require a rewrite of Makefile.generic

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA161

Future: The mechanism should alert users on dependency conflicts, rather than silently capturing them – possibly by breaking the build process.

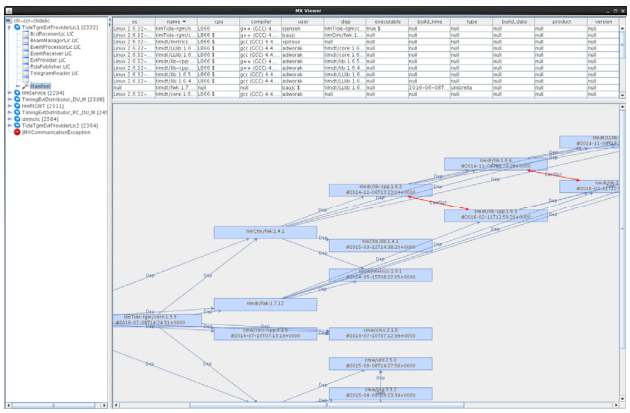

Figure 1: Dependency graph showing conflicts in red.

Unit Testing and Mocking

Status: Stable and available. Google test [2] and Google mock [3] targets are part of Makefile.generic and can therefore be triggered from our continuous integration service.

Uptake: Strong in FESA (~950 tests) and CMW (~3500 tests), Medium in TIMING, but includes C software (~250 tests), Weak in SILECS (~10 tests)

Pros: CMW described unit testing as "slow but safe", forcing developers to think about concrete, testable cases which solidifies code. It gives confidence that changes have not introduced regression. Tests often ramify further into the software than foreseen, which helps justify the investment. Several cases were reported where testing found bugs that would have compromised operation if deployed. Also, the approach fits the work habits of younger developers.

Cons: Implementation of tests takes time and the investment may be hard to justify, especially since benefits often appear only in the long term and are not always quantifiable. One project reported having a 1:1 ratio between lines-of-tested-code and test-code. One project found that it takes too much effort to manage dependencies. Implementing tests will always come second to fixing operational issues. Some (historical) code is not suited for unit testing without refactoring and it is uncertain if such an effort is justified. TIMING found unit testing to be most justified for libraries offered to many users, and less so when it comes to delivering executables. TIMING noted a risk in believing that if all tests pass, all is good – one should stay critical and not forget that tests do not cover all cases.

Future: Better integration and more automation in terms of dependency management is needed, allowing developers to focus on implementing the test code.

Continuous Integration

Status: The continuous integration (CI) service, provided by the Atlassian *Bamboo* [4] product in our case, is stable and available, triggered by commits to SVN and capable of invoking targets in Makefile.generic.

Uptake: Strong in FESA (~400 tests in ~20 plans, automatic launch on commit as fail-early is essential), CMW (~3500 tests in ~20 plans), preventing release if any test fails) and TIMING (~600 tests in ~20 plans) and weak in SILECS (~20 tests in ~10 plans)

Pros: TIMING reported value in repeating certain tests as sometimes a functionality may fail "only on the 101st call" due to changes in the execution environment. FESA noted the importance of "fail-early" achieved by having SVN commits immediately triggering Bamboo build plans. FESA noted that CI helps avoiding "operational testing" by first-adopter users and important for testing un-noticed ramifications of changes. TIMING uses CI as a sort of watchdog via a test which regularly calls an operational service known to fail periodically – thereby being notified on service failure. FESA and CMW reported that although implementation and maintenance is very time consuming, the operational stability gained justifies the investment.

Cons: CI was found to be very heavy to adopt initially (FESA reported 2 person-months of effort) and also to maintain due to current limitations in the integration with our environment. Implementing a test takes approximately the same time as implementing the feature being tested.

Future: There is a need to decrease the effort required to create and maintain test plans. This will be addressed via new targets in the common build tool.

Code Review

Status: We perform code reviews using the Atlassian FishEye/Crucible tool [5]. This service is stable and available.

Uptake: Strong in FESA and CMW (all changes reviewed by at least 2 persons within the week), Weak in TIMING and SILECS (only on a few occasions).

Pros: FESA and CMW found nothing but positive effects, although the time investment is only recovered in the longer term. The value lies mostly in improving the software architecture and design, though some bugs have been found, leading to additional unit tests. CMW found value in conveying coding style to newcomers. TIMING reported interest since reviews help distribute knowledge of functionality between team members, which in turn will help removing single-point-of-failures in the team. However, TIMING reported reluctance to adopting a regular procedure due to operational pressure and resource scarcity combined with code size and complexity. SILECS reported interest and saw a big advantage in using a tool like FishEye/Crucible (Figure 2), bringing down the barrier to adopting the process and particularly expected value from homogenizing coding style.

Cons: The process is time consuming – for TIMING prohibitively so. FESA found the Crucible navigation features to be limited, without an "IDE feeling". Overall, code reviews demand discipline and there is risk of friction as some may take reviews personal.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA161

Future: The review process could be increasingly formalized, specifying criteria such as "maximum delay allowed" and "minimum participation for acceptance", though such criteria should likely be defined per project.

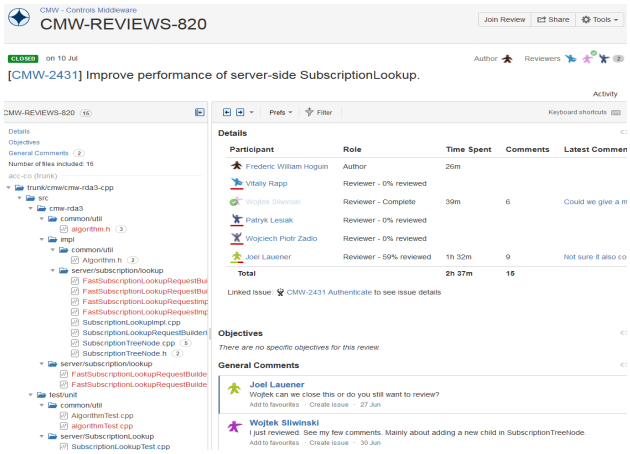

Figure 2: Code reviews.

Static Code Analysis

Since the 2013 article [1], we have moved from the commercial tool *Coverity* [6] to the open-source SonarQube [7] due to licensing considerations.

Status: Not yet fully integrated for C/C++.

Uptake: Not used actively in any project.

Pros: SILECS saw an interest for cleaning up the code base, but was so far unable to allocate time to do it. FESA noted the visual appeal of reports, but found them of little use as they mostly detect aesthetic problems rather than actual bugs.

Cons: The code base of the four projects is large and long-standing. Running a code analysis tool on it yields an overwhelming amount of errors and warnings – mostly of an aesthetic nature - too big to address.

Future: With better integration, automation and fine-tuned configuration, the tool may gain usefulness but we are not currently able to say if the effort is justified.

Memory Debugging/Profiling

Status: The Valgrind [8] installation is stable and available via targets in Makefile.generic.

Uptake: Strong in CMW (integrated as automated post-commit tests), weak in TIMING (due to a principle of not using any dynamic memory allocation), weak in FESA (ad hoc performance tests and spotting problems in client code), not used in SILECS.

Pros: FESA and CMW reported value by detecting slower forms of memory leaks, which may go undetected by unit tests as these execute over short periods of time.

Cons: FESA reported too many false positives due to use of shared memory. CMW reported many false positives caused by CMW's extensive use of timeout mechanisms combined with the fact that a binary executes slower under Valgrind – thereby triggering the timeouts.

Future: Automated memory debugging/profiling is being considered, but may be too complex to implement due to different configuration needs across projects. A mechanism to select between different configurations could be envisaged.

Runtime Process Metrics

Status: CMX [9] is stable and available, allowing executing processes to write internal variables to shared memory from where they can be accessed by various means, such as the "CMX Viewer" (Figure 3).

Uptake: Medium in TIMING and FESA, not used in CMW as similar functionality was already implemented. Not yet used in SILECS due to lack of time to implement.

Pros: Easy to use, providing for a good value-to-effort ratio. It provides non-intrusive diagnostics on a running process without the need to restart under debugger. Both TIMING and FESA reported most value when applied to libraries used by clients. FESA reported around 5 yearly cases of detecting client software too slow to consume certain events.

Cons: FESA found the integration with the control system to be sub-optimal. They expressed interest in a mechanism for (re-)setting CMX values as well as in support for data structures. TIMING noted that exposed metrics show the current state and that value lies in periodically logging CMX metrics to allow seeing trends over time.

Future: A better integration with the controls system is desirable, facilitating for instance configuration of how CMX data is acquired and persisted centrally over time. However, this relates more to external tools and services than to CMX.

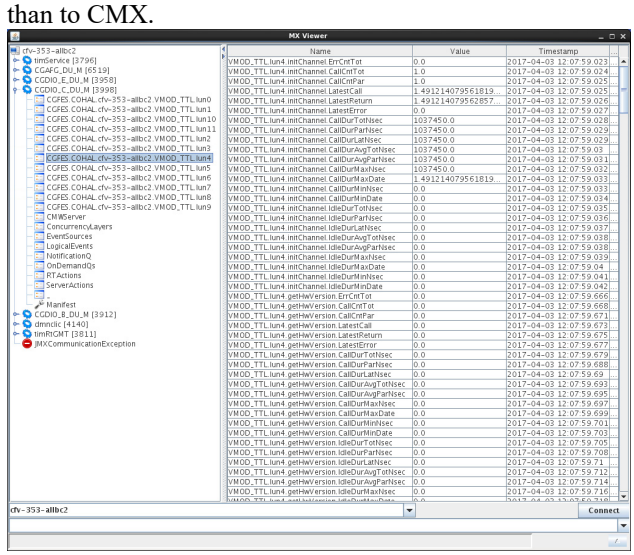

Figure 3: Run-time metrics via CMX.

CHALLENGES

The challenges encountered in applying the SIP4C/C++ procedures and tools can be summarized as follows:

Resources Driving SIP4C/C++

Understanding requirements, developing, integrating, adapting and evolving the various tools require significant

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA161

time and effort, in particular initially. Once operational, resources are needed for tool maintenance and evolution. Otherwise, project teams will refrain from using the tools. Time must be allocated from persons who well understand project needs and who are also knowledgeable about the tools employed. Finally, team leaders must dedicate time to repeatedly and actively promote the initiative.

Technical

Maximizing value and minimizing adoption barriers requires detailed knowledge about the individual and sometimes conflicting project needs. It is a fine balance to strike between providing simple, smooth and intuitive user experiences while covering the requirements of heterogeneous projects – both of which are essential for promoting uptake.

Cultural

Some team procedures are coupled to decade-old software and changing this may depend on refactoring the software. If operational pressure is high, projects struggle to create the "space" required to adopt these "finer" aspects of software development. Management must facilitate by supporting and encouraging that projects plan accordingly. Concerning code reviews, the challenge is to instil routine and discipline with respect to participation and deadlines. There seems to be consensus that this is beneficial.

Personal

It takes courage and an open mind to have one's software reviewed by peers. Code reviews can cause frictions as criticism may be perceived to be personal. Focusing on the common ownership of the code and knowledge-sharing aspects can mitigate such situations and hence must be encouraged by team leaders.

FUTURE PLANS

In parallel to continued promotion of SIP4C/C++, the barrier to adopting its procedures and tools must be minimized. Specifically, we see a need to improve the build/release chain to improve user experience and increase functionality, in particular with respect to dependency management. To address these concerns, a new activity was started named CODEINE.

CONCLUSION

After four years of applying SIP4C/C++ there is consensus in our project teams that the benefits - despite being difficult to quantify - undoubtedly justify the investment in the longer term: Code robustness has increased, leading to fewer operational incidents and we have seen increased cross-project knowledge sharing amongst developers.

Yet, we realize that flexibility is a must – in procedures and tools alike - in that each project has inherent characteristics influencing what is optimal and even possible. Leeway must be given in project planning for them to adopt SIP4C/C++. The initiative requires a continued, active effort for promotion and facilitation at management, technical and cultural levels.

REFERENCES

[1] K. Sigerud et al., "Tools and Rules to Encourage Quality for C/C++ Software", http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/papers/moppc087.pdf , http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/posters/moppc087_poster.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] Crucible, https://www.atlassian.com/software/crucible

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

[7] SonarQube, https://www.sonarqube.org/

[8] Valgrind, http://valgrind.org/

[9] F. Ehm *et al.*, "CMX - A Generic In-Process Monitoring Solution for C and C++ Applications", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, October 2013. https://gitlab.cern.ch/cmx/cmw-cmx/wikis/home