AGH University of Science and Technology

Update and upgrade of the GGSS system for ATLAS TRT detector

Arkadiusz Kasprzak
Jarosław Cierpich

Supervisor: Bartosz Mindur

Agenda

Introduction to GGSS

- Gas Gain Stabilization System (GGSS) is a project integrated with ATLAS Detector Control System (DCS).
- It helps to ensure proper operation of the TRT (Transition Radiation Tracker) detector, which is a part of the ATLAS detector at CERN.
- GGSS consists of hardware and software layers.
- Today we will focus mainly on the software layer.
- Most of the codebase is written using C++. There is also some Python and Bash code.

GGSS software

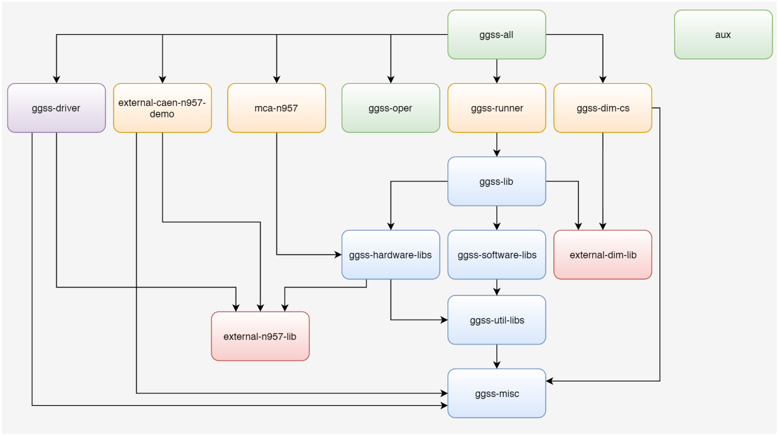

Figure: Software components of the GGSS project and their dependencies.

Tasks

- C++ codebase refactoring:
    - migration to C++11/14 (range-for loops, uniform initialization etc)
    - removing old, unused code
    - adding more comprehensive documentation
    - introducing TDD (Test Driven Development)
- CMake files refactoring
- creating tools for versioning and Git submodule handling
- creating Python library and apps for hardware testing (in progress)

Understanding the constraints

- The user must be able to build the project using the CERN infractructure
- Code refactoring had to be performed with this constraints in mind.
- Only old version of CMake (2.8) and GCC (C++11/14, but no 17/20) available.

Migration to C++11/14

Example of migration to C++11/14 - replacing iterator loop with range-for one. Below You can see the old code.

Listing 1: Example of old C++ code (before refactoring).

```
for ( XMLTag::NestedTags::const_iterator j = tag.getNestedTags().begin()
            ; j != tag.getNestedTags().end()
            ; j++
        )
{
    if((j->second->getName() == tagName)
        &&(j->second->getAttributeValue("id") == idValue))
        return j->second;
    else
        m_findTagById(*(j->second), tagName, idValue);
} // endfor
```

Migration to C++11/14

Listing 2: Example of new C++ code (after refactoring).

```cpp
const XMLTag::NestedTags& nestedTags = startingTag.getNestedTags();
for(const auto& nestedTag: nestedTags)
{
    if((nestedTag.second->getName() == tagName) &&
         (nestedTag.second->getAttributeValue("id") == idValue))
    {
        return nestedTag.second;
    }
}
```

- using range-for loop increases readability of the code
- `else` clause has been removed - result of the recursive function call was never used
- no need to use the `*` operator
- `nestedTag` is a better name than `j`

Removing old, unused code

- The project contained a lot of code (functions/methods) that were never used.
- Some of them could even be harmful if used.
- Below example shows two methods that have been removed (why?) from `QueueLimited` class (a queue with size limit).

Listing 3: Example of removed code.

```
// return the whole queue
const std::deque<T>& getQueue () const {
    return c;
}

// return the whole queue
std::deque<T>& getQueue () {
    return c;
}
```

Introducing Test Driven Development

- For unit tests, we are using `Boost.Test`, because it is simple and available when using CERN infrastructure.
- Component are tested during refactoring, we make sure that our changes do not introduce any new bugs.
- Each component can be tested separately.

Listing 4: Unit test example

```
/**
 * \brief Checks if proper exception is thrown when performing pop()
 *        operation on empty container.
 */
BOOST_AUTO_TEST_CASE(
    testIfExceptionIsThrownWhenTryingToPopFromEmptyContainer)
{
    QueueLimited<int> queue{};
    BOOST_CHECK_THROW(
        queue.pop(),
        QueueLimited<int>::ReadEmptyQueueException);
}
```

Continous Integration

- Practice widely used during modern software development.
- Developers integrate code into repository frequently.
- Each code contribution is automatically built and tested.
- This allows for quick error detection.
- We use GitLab CI/CD for building and testing the project.

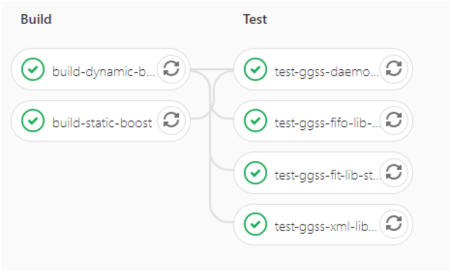

Figure: Example of CI pipeline used in the project.

CMake files refactoring

- GGSS uses CMake for managing the build process of the software.
- CMake is platform and compiler independent.
- CMake files have been slightly refactored to improve readability by using macros and functions.

Listing 5: Old version of CMake used for building *thread-lib*

```
set(target_name "thread")
if(NOT TARGET ${target_name})
    set(CMAKE_MODULE_PATH "${GGSS_MISC_PATH}")
    include(BuildLibrary)
    include(FindLibraryBoost)
    include(SetupDoxygen)
    include(SetupTests)

    # notice the need to set some variables before including the file
    set(dependency_prefix "${CMAKE_CURRENT_SOURCE_DIR}/..")
    set(dependencies "handle" "log")
    include(BuildDependencies)
endif()
```

CMake files refactoring

- Instead of including the CMake template files (which just pastes the code), we invoke `ggss_build_library` with named parameters.
- Unit tests and Doxygen support has been moved to `ggss_build_library` macro, because every library in the project uses them.

Listing 6: New version of CMake used for building *thread-lib*

```
set(CMAKE_MODULE_PATH "${GGSS_MISC_PATH}")
include(BuildLibrary)

ggss_build_library(
    TARGET_NAME "thread"
    DEPENDENCY_PREFIX "${CMAKE_CURRENT_SOURCE_DIR}/.."
    DEPENDENCIES "log" "sigslot"
)
```

Plans for future improvements

- Further code refactoring.
- Introducing new features, for example on-start and on-demand GGSS parameters update.
- Creating RPM package with whole project - easy deployment.