



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Jarosław Cierpich
Arkadiusz Kasprzak

kierunek studiów: **informatyka stosowana**

Rozbudowa i uaktualnienie oprogramowania systemu GGSS detektora ATLAS TRT

Opiekun: **dr hab. inż. Bartosz Mindur**

Kraków, styczeń 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelnia przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....
(czytelny podpis)

Spis treści

1. Wstęp	7
1.1. Wprowadzenie do systemu GGSS	7
1.2. Cel pracy	7
2. Zastosowane technologie	9
2.1. Język C++	9
2.2. Biblioteki	11
2.2.1. Rodzaje bibliotek	12
2.2.2. Biblioteki statyczne	12
2.2.3. Biblioteki współdzielone	15
2.3. Narzędzie CMake	19
2.4. Język Python	21
2.5. Powłoka systemu operacyjnego (Bash)	22
2.6. System kontroli wersji Git i portal Gitlab	24
2.7. Menadżer pakietów RPM	28
2.8. Technologie wirtualizacji i konteneryzacji	29
3. Stan początkowy projektu	33
3.1. Architektura	33
3.2. Budowanie	36
3.3. Dostarczanie i uruchamianie	37
3.4. Kontrola wersji	38
4. Stan docelowy projektu	39
4.1. Finalna wersja projektu	39
4.2. Stan oczekiwany w ramach projektu inżynierskiego	40
5. Ograniczenia dostępnej infrastruktury	41
5.1. Ograniczone uprawnienia w środowisku docelowym	41
5.2. Wersje kompilatorów i interpreterów	41
5.3. Wersja narzędzia budującego CMake	42

5.4. Związek projektu z wersją jądra systemu.....	42
6. Wykonane prace	43
6.1. Wykorzystanie funkcjonalności portalu GitLab wspierających zarządzanie projektem	43
6.2. Migracja projektu do systemu kontroli wersji Git i zmiany w architekturze	51
6.3. Zmiana sposobu budowania aplikacji.....	56
6.3.1. Zarys rozwiązania	56
6.3.2. Szablony CMake	57
6.3.3. Budowanie poszczególnych bibliotek.....	60
6.3.4. Budowanie aplikacji i całego projektu	61
6.3.5. Budowanie biblioteki DIM.....	63
6.4. Zastosowanie podejścia CI/CD	68
6.4.1. Możliwe sposoby implementacji podejścia CI/CD w projekcie GGSS	68
6.4.2. Opis działania GitLab CI/CD	69
6.4.3. Opis automatyzacji za pomocą GitLab CI/CD w projekcie GGSS	71
6.5. Budowanie i dystrybucja sterownika oraz aplikacji testującej	76
6.6. Maszyna wirtualna oraz konteneryzacja	79
6.7. Dokumentacja projektu.....	82
6.7.1. Język Markdown.....	82
6.7.2. Opis wykonanej dokumentacji	83
6.8. Pomniejsze prace.....	85
6.8.1. Integracja bibliotek napisanych w języku C z aplikacją w C++	85
6.8.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake .	86
7. Testy nowej wersji oprogramowania.....	87
7.1. Przebieg testu	87
7.2. Wyniki testu.....	92
8. Dalsza ścieżka rozwoju projektu.....	97
8.1. Wprowadzenie zautomatyzowanego systemu testowania projektu	97
8.2. Poprawa jakości kodu napisanego w języku C++.....	97
8.3. Automatyzacja procesu publikowania produktu	98
8.4. Migracja do języka Python 3	98
8.5. Przygotowanie dodatkowych interfejsów graficznych	98
9. Podsumowanie oraz wnioski	99
A. Dodatki/Appendixes	101

A.1. Porównanie początkowej i obecnej struktury projektu oraz kodu źródłowego	101
A.2. Adding modules to the project using existing CMake templates	105
A.3. Preparing the virtual machine to work as a runner	107

1. Wstęp

1.1. Wprowadzenie do systemu GGSS

Detektor ATLAS (*A Toroidal LHC ApparatuS*), znajdujący się w Europejskim Ośrodku Badań Jądrowych *CERN*, jest jednym z detektorów pracujących przy Wielkim Zderzaczu Hadronów (*LHC - Large Hadron Collider*). Pełni on kluczową rolę w rozwoju fizyki cząstek elementarnych, w szczególności badania przy nim prowadzone doprowadziły do potwierdzenia istnienia tzw. *bozonu Higgsa* w roku 2012 [1].

Detektor ATLAS charakteryzuje się budową warstwową - składa się z kilku pod-detektorów [2]. Jednym z nich jest Detektor Wewnętrzny (*Inner Detector*) składający się z trzech głównych elementów zbudowanych za pomocą różnych technologii. Elementy te, w kolejności od położonego najbliżej punktu zderzeń cząstek, to: detektor pikselowy (*Pixel Detector*), krzemowy detektor śladów (*SCT - Semiconductor Tracker*) oraz detektor promieniowania przejścia (*TRT - Transition Radiation Tracker*). Dokładny opis zasad działania całego detektora oraz poszczególnych jego komponentów wykracza poza zakres niniejszego manuskryptu.

W kontekście niniejszej pracy kluczowym jest System Stabilizacji Wzmocnienia Gazowego (*GGSS - Gas Gain Stabilisation System*) dla detektora TRT. Jego oprogramowanie jest zintegrowane [2] z systemem kontroli detektora ATLAS (*DCS - Detector Control System*). W skład systemu GGSS wchodzi zarówno urządzenia takie jak multiplexer i zasilacz wysokiego napięcia, jak i rozbudowana warstwa oprogramowania. Autorzy pracy zaprezentują opis zmian, jakie do tej pory wprowadzili w projekcie GGSS. Zmiany te obejmują m.in. sposób budowania aplikacji wchodzących w skład systemu, ale również automatyzacja prac związanych z jego utrzymaniem i użytkowaniem.

1.2. Cel pracy

Przed autorami postawiony został szereg celów do zrealizowania, związanych zarówno ze zdobyciem wymaganej wiedzy domenowej, jak i przeprowadzeniem modyfikacji oprogramowania systemu GGSS.

Jednym z nich było zapoznanie się z infrastrukturą informatyczną CERN-u. Praca z oprogramowaniem oparta jest tam o unikalny ekosystem, mający zapewnić bezpieczeństwo i stabilność

całej infrastruktury, co wiąże się z wieloma ograniczeniami dotyczącymi m.in. dostępu do komputerów produkcyjnych. Konieczne było więc uzyskanie odpowiednich uprawnień i zdobycie doświadczenia w pracy z tą infrastrukturą. Ze względu na domenę działania systemu GGSS celem było również zdobycie wiedzy na temat sposobu pracy przy dużych eksperymentach, na przykładzie eksperymentu ATLAS. Ponadto uczestnictwo w rozwoju projektu tego typu miało na celu nabycie przez autorów doświadczenia w pracy w międzynarodowym środowisku, jakim jest CERN. Kluczowym dla poprawnego przeprowadzenia prac było również zapoznanie się autorów z zastosowaniem i podstawami sposobu działania systemu GGSS.

Oprócz wyżej wymienionych czynności związanych ze zdobyciem podstawowej wiedzy domenowej, celem niniejszej pracy było przeprowadzenie modyfikacji w warstwie oprogramowania projektu GGSS. Do postawionych przed autorami zadań należało zaplanowanie prac i utworzenie wygodnego, nowoczesnego środowiska do zarządzania projektem informatycznym oraz prostego w rozwoju, intuicyjnego systemu budowania oprogramowania opartego o narzędzie *CMake*. Miało to na celu umożliwienie modularyzacji projektu tak, by każdy z komponentów mógł być niezależnie budowany. Ponadto zadaniem autorów była migracja projektu do systemu kontroli wersji *Git*, stanowiącego ogólnie przyjęty standard we współczesnych projektach informatycznych. W celu uproszczenia procedury wdrażania projektu w środowisku produkcyjnym celem autorów było również zautomatyzowanie procesu budowania i dystrybucji projektu. Na koniec, by umożliwić innym uczestnikom projektu sprawne korzystanie z nowych rozwiązań, przygotowana miała zostać dokumentacja w formie krótkich instrukcji oraz zestawów komend. Dokumentacja, z uwagi na międzynarodowy charakter środowiska w CERN, miała zostać napisana w języku angielskim.

Niniejszy manuskrypt opisuje przede wszystkim prace związane z rozwojem oprogramowania przeprowadzone przez autorów. Praca opisuje stan początkowy projektu, założenia dotyczące stanu docelowego oraz wybrane, zdaniem autorów najważniejsze, zadania zrealizowane w ramach pracy z oprogramowaniem systemu GGSS.

2. Zastosowane technologie

Niniejszy rozdział zawiera krótki opis najważniejszych technologii i narzędzi używanych przez autorów podczas pracy z oprogramowaniem systemu GGSS. Przedstawione tu opisy zawierają podstawową wiedzę o sposobie działania i użytkowania tych technologii - szczegółowe przykłady przedstawione zostały w dalszej części pracy, w kontekście konkretnych rozwiązań zrealizowanych przez autorów w projekcie.

2.1. Język C++

C++ jest kompilowanym językiem programowania ogólnego przeznaczenia [3] opartym o statyczne typowanie. Został stworzony jako obiektowe rozszerzenie języka C (z którym jest w dużej mierze wstecznie kompatybilny), lecz wraz z rozwojem pojawiło się w nim wsparcie dla innych paradygmatów, w tym generycznego i funkcyjnego. Sprawilo to, że język ten stał się bardzo wszechstronny - pozwala zarówno na szybkie wykonywanie operacji niskopoziomowych, jak i na tworzenie wysokopoziomowych abstrakcji [3]. Dodatkową cechą wyróżniającą C++ wśród innych języków umożliwiającą programowanie obiektowe jest jego wysoka wydajność.

Standardy języka

W ciągu ostatnich kilku lat C++ przechodzi proces intensywnego rozwoju - od 2011 roku pojawiły się trzy nowe standardy tego języka, a kolejny przewidziany jest na rok 2020. Wspomniane nowe standardy to:

- C++11 - wprowadza funkcjonalności takie jak: wsparcie dla wielowątkowości, wyrażenia lambda, referencje do *r-wartości*, biblioteka do obsługi wyrażeń regularnych, dedukcja typów za pomocą słowa kluczowego *auto* czy pętla zakresowa. Standard ten uważany jest za przełom w rozwoju języka.
- C++14 - rozszerza zmiany wprowadzone w C++11. Nie zawiera tak wielu przełomowych zmian jak poprzedni standard - twórcy skupili się na poprawie błędów oraz rozwoju istniejących rozwiązań [4], np. dedukcji typu zwracanego z funkcji za pomocą słowa kluczowego *auto*.

- C++17 - wprowadza m.in. nowe typy danych (np. `std::variant`, `std::byte` i `std::optional`), algorytmy współbieżne, biblioteka *filesystem* przeznaczona do obsługi systemu plików oraz rozszerzenie mechanizmu dedukcji typów w szablonach na szablony klas [5]. Standard ten usuwa również pewne elementy uznane za przestarzałe, np. inteligentny wskaźnik `std::auto_ptr`, zastąpiony w standardzie C++11 przez inne rozwiązanie.

Zmiany wprowadzane w nowych standardach pozwalają na tworzenie czytelniejszego kodu, który łatwiej utrzymywać i rozwijać. Ma to znaczenie zarówno na poziomie pojedynczych instrukcji czy typów danych, jak i na poziomie architektury projektu. Listingi 2.1 oraz 2.2 przedstawiają przykład zmiany, jaka zaszła między standardem C++03, a C++11. Zaprezentowany kod realizuje w obu przypadkach iterację po zawartości kontenera typu `std::vector<int>` mającą na celu wypisanie na standardowe wyjście jego zawartości. Przykład ten, pomimo że bardzo prosty, dobrze obrazuje wzrost jakości i czytelności kodu w nowym standardzie.

Listing 2.1. Przykład kodu w języku C++ napisany z wykorzystaniem standardu C++03

```
// kontener zawierający 6 elementów typu int - inicjalizacja
// za pomocą tymczasowej tablicy, możliwa w standardzie C++03
// języka
int tmp_arr[] = {1, 2, 3, 4, 5, 6};
std::vector<int> a (tmp_arr, tmp_arr + 6);

// iteracja po zawartości kontenera w standardzie C++03
for (std::vector<int>::const_iterator it = a.begin(); it != a.end(); ++it) {
    std::cout << *it << " ";
}
```

Listing 2.2. Przykład kodu w języku C++ napisany z wykorzystaniem funkcjonalności ze standardu C++11 (zakresowa pętla for)

```
// kontener zawierający 6 elementów typu int - nowy
// sposób inicjalizacji
std::vector<int> a {1, 2, 3, 4, 5, 6};

// iteracja po zawartości kontenera w standardzie C++11 -
// przykład zastosowania zakresowej pętli for
for (const auto& elem: a) {
    std::cout << elem << " ";
}
```

Boost

Boost jest zestawem bibliotek dla języka C++, poszerzających w znacznym stopniu wachlarz narzędzi programistycznych dostarczanych przez język. Biblioteki wchodzące w skład Boost

dostarczają funkcjonalności takich, jak: wygodne przetwarzanie tekstu, zapewnienie interfejsu między C++ a językiem Python czy programowanie sieciowe [6]. Boost to projekt aktywnie rozwijany, bardzo popularny. Niektóre z bibliotek wchodzących w jego skład zostały przeniesione (nie zawsze w postaci identycznej względem oryginału) do standardu C++.

2.2. Biblioteki

Biblioteki są jednym z podstawowych narzędzi wprowadzających podział programu na niezależne komponenty oraz dających możliwość wielokrotnego użycia tego samego kodu. Stanowią więc zbiór funkcji, struktur itp. udostępnionych do użycia przez inne programy. Niniejsza część pracy skupia się na bibliotekach opisywanych z perspektywy języków C oraz C++. Opis dotyczył będzie rozwiązań związanych z systemami typu UNIX (nie zostanie poruszony sposób działania bibliotek na systemach Windows). Autorzy zdecydowali się opisać to zagadnienie szczegółowo z uwagi na fakt, iż architektura projektu GGSS w dużej mierze opiera się o mechanizm bibliotek. Rozważania teoretyczne wzbogacone zostaną więc prostym przykładem.

Opis przykładu

Przykład prezentujący działanie bibliotek został napisany w języku C i składa się z dwóch katalogów: *app*, zawierającego kod źródłowy programu, oraz *complex*, zawierającego kod źródłowy, który zostanie wykorzystany do stworzenia prostej biblioteki pozwalającej na wykonywanie podstawowych operacji na liczbach zespolonych. Listing 2.3 zawiera wynik polecenia *tree*, które wypisuje na standardowe wyjście strukturę katalogu z projektem. Autorzy zdecydowali się pokazać proces budowania bibliotek bez wykorzystania narzędzi automatyzujących ten proces (takich jak *CMake*), gdyż znajomość zasad działania tego mechanizmu okazała się dla nich bardzo pomocna podczas rozwiązywania problemów związanych z wykorzystaniem bibliotek w systemie GGSS, gdzie wspomniane narzędzia były już używane.

Listing 2.3. Struktura katalogów projektu stanowiącego bazę przykładu dotyczącego bibliotek.

```
user@host:~$ tree --charset=ascii
.
|-- app
|   '-- app.c
'-- complex
    |-- complex_number.h
    |-- complex_ops.c
    '-- complex_ops.h

2 directories, 4 files
```

2.2.1. Rodzaje bibliotek

Na systemach z rodziny UNIX wyróżniamy dwa podstawowy typy bibliotek: **statyczne** oraz **współdzielone (shared)**, nazywane również **dynamicznymi**. Podejścia te znacznie różnią się od siebie. Każde z nich oferuje pewne zalety względem drugiego, przez co oba pozostają dziś w użyciu.

2.2.2. Biblioteki statyczne

Koncepcja stojąca za bibliotekami statycznymi jest bardzo prosta [7] - są to archiwa zawierające w sobie kolekcję plików obiektowych (*.o). Do tego typu bibliotek dołączone muszą być pliki nagłówkowe zawierające m.in. deklaracje funkcji stanowiących interfejs programistyczny pomiędzy biblioteką, a używającym ją programem. Cechą bibliotek statycznych odróżniającą je od bibliotek dynamicznych jest fakt, że są one dołączane do plików obiektowych głównego programu w czasie linkowania - stanowią więc część wynikowego pliku wykonywalnego.

Tworzenie biblioteki statycznej

Rysunek 2.1 przedstawia schematycznie proces tworzenia bibliotek statycznych. Składa się on z dwóch etapów [8]:

- kompilacja plików źródłowych biblioteki do postaci obiektowej za pomocą *gcc* (listing 2.4). Wynikiem powinny być pliki o rozszerzeniu *.o odpowiadające wykorzystanym plikom źródłowym.

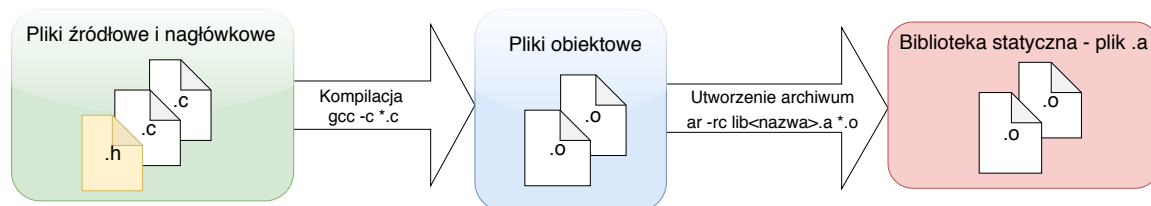
Listing 2.4. Kompilacja plików źródłowych biblioteki do postaci obiektowej - polecenie oraz jego wynik

```
user@host:~/complex$ gcc -c *.c
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o
```

- utworzenie archiwum zawierającego pliki obiektowe za pomocą programu *archiver* (listing 2.5). Wynikiem powinien być plik o rozszerzeniu *.a. Podczas tworzenia biblioteki należy nadać jej odpowiednią nazwę, zgodną z formatem *lib<nazwa>.a*.

Listing 2.5. Utworzenie biblioteki statycznej z plików obiektowych - polecenie oraz jego wynik

```
user@host:~/complex$ ar -rc libcoml.a *.o
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o  libcoml.a
```



Rys. 2.1. Proces tworzenia biblioteki statycznej z uwzględnieniem komend koniecznych do wykonania poszczególnych etapów [7]

Zawartość powstałego archiwum można zbadać za pomocą komendy `ar -t` - wyświetla ona wszystkie pliki obiektowe wchodzące w skład danej biblioteki. Istnieje również możliwość wypisania symboli - służy do tego narzędzie `nm`. Użycie tych narzędzi na wykonanym przez autorów przykładzie ilustruje listing 2.6. Wynikiem polecenia `nm` są tam dwa symbole, oznaczające dwie udostępnione dla użytkowników biblioteki funkcje (dodawanie i odejmowanie liczb zespolonych).

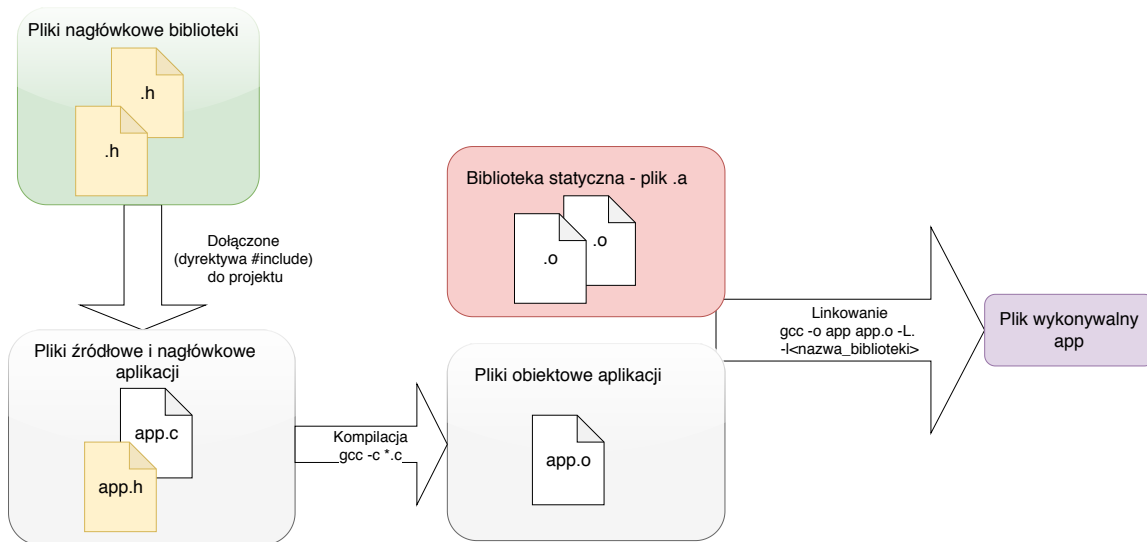
Listing 2.6. Użycie poleceń `ar -t` oraz `nm` na bibliotece statycznej.

```
user@host:~/complex$ ar -t libcoml.a
complex_ops.o
user@host:~/complex$ nm libcoml.a

complex_ops.o:
0000000000000000 T add_complex_numbers
0000000000000091 T subtract_complex_numbers
```

Dołączanie utworzonej biblioteki do programu

Rysunek 2.2 przedstawia schematycznie proces dołączania utworzonej biblioteki statycznej do programu. Proces ten składa się z następujących etapów:



Rys. 2.2. Proces dołączania biblioteki statycznej do programu z uwzględnieniem komend koniecznych do wykonania poszczególnych etapów [7]

- dołączenie do źródeł programu plików nagłówkowych zawierających deklaracje stanowiące interfejs między programem a biblioteką (dyrektywa *include* - listing 2.7).

Listing 2.7. Plik *app.c* zawierający dyrektywę *include* dołączającą plik nagłówkowy zawierający deklaracje funkcji z biblioteki statycznej

```

#include "../complex/complex_ops.h"
#include <stdio.h>

int main(void) {

    complex_number a = {2.5, 3.7};
    complex_number b = {3.5, 0};

    complex_number c = add_complex_numbers(a, b);
    complex_number d = subtract_complex_numbers(a, b);

    printf("%lf %lf\n", c.real, c.imaginary);
    printf("%lf %lf\n", d.real, d.imaginary);

    return 0;
}
  
```

- kompilacja plików źródłowych programu do postaci plików obiektowych za pomocą *gcc* (listing 2.8). Wynikiem powinien być zestaw plików obiektowych odpowiadający wykorzystanym plikom źródłowym.

Listing 2.8. Kompilacja plików źródłowych głównego programu do postaci obiektowej.

```
user@host:~/app$ gcc -c *.c
user@host:~/app$ ls
app.c  app.o
```

- połączenie plików obiektowych i biblioteki w plik wykonywalny za pomocą *gcc* (listing 2.9). Opcja *-L* pozwala określić ścieżkę do dołączanej biblioteki, natomiast *-l* - nazwę biblioteki (bez przedrostka *lib* i rozszerzenie *.a*).

Listing 2.9. Linkowanie plików obiektowych programu z biblioteką statyczną, wynik uruchomienia programu obrazujący poprawne działanie przykładu.

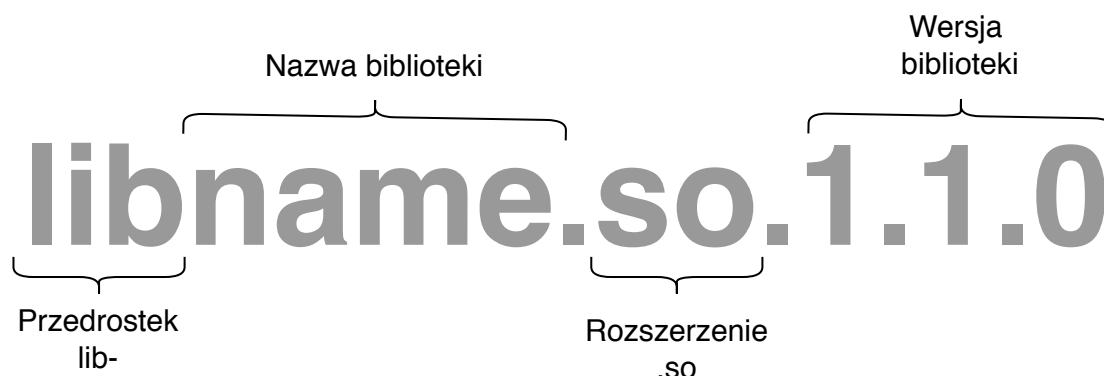
```
user@host:~/app$ gcc -o app app.o -L../complex -lcom1
user@host:~/app$ ls
app  app.c  app.o
user@host:~/app$ ./app
6.000000 3.700000
-1.000000 3.700000
```

2.2.3. Biblioteki współdzielone

Działanie bibliotek współdzielonych jest bardziej skomplikowane. Z uwagi na fakt, iż w projekcie GGSS biblioteki współdzielone są używane jedynie w postaci zewnętrznych zależności, przedstawiony opis dotyczył będzie jedynie podstaw tworzenia i działania tych bibliotek. Główna różnica w stosunku do bibliotek statycznych polega na zmianie czasu, w którym biblioteka jest dołączana do programu. W przypadku bibliotek współdzielonych dzieje się to w trakcie ładowania programu do pamięci. Oznacza to, że biblioteki te nie są częścią pliku wykonywalnego programu, dzięki czemu programy te mają mniejszy rozmiar i nie ma konieczności ich ponownej kompilacji w wypadku wprowadzenia zmian w bibliotece. Ponadto kod bibliotek tego typu jest ładowany do pamięci tylko raz - przy pierwszym użyciu biblioteki, i może być współdzielony. Podobnie jak wcześniej do tego typu bibliotek dołączone muszą być pliki nagłówkowe zawierające *m.in.* deklaracje funkcji stanowiących interfejs programistyczny pomiędzy biblioteką, a używającym ją programem.

Nazwy i wersjonowanie bibliotek współdzielonych

Podczas pracy z bibliotekami współdzielonymi istotne jest przestrzeganie narzuconych konwencji dotyczących sposobu nadawania im nazw oraz ich wersjonowania. Rys. 2.3 przedstawia poszczególne części składające się na poprawną nazwę pliku biblioteki. Składa się ona z przedrostka *lib*, po którym następuje właściwa nazwa biblioteki. Następnie po kropce powinno znaleźć się rozszerzenie *.so* a po nim, rozdzielone kropkami, liczby określające wersję biblioteki.



Rys. 2.3. Elementy poprawnej nazwy biblioteki współdzielonej

Wersja biblioteki składa się z trzech liczb [9]:

- nadrzędny numer wersji (*major version number*) - różnica na tym poziomie oznacza zmianę interfejsu skutkującą brakiem kompatybilności między wersjami bibliotek
- podrzędny numer wersji (*minor version number*) - różnica na tym poziomie oznacza zwykle zachowaną kompatybilność między wersjami biblioteki
- numer wydania (*release number*) - opcjonalny

Poza nazwą samego pliku bibliotekę współdzieloną określają również dwie inne nazwy [10]. Pierwsza z nich to tzw. **soname**. Stanowi ona podzbiór nazwy przedstawionej na Rys. 2.3, bez dwóch ostatnich numerów wersji (przykład: *libname.so.1*). Nazwa *soname* stanowi zwykle nazwę dowiązania symbolicznego do pliku biblioteki. Druga z nazw to tzw. **linker name** - stanowi ona nazwę biblioteki bez numeru wersji (tzn. np. *libname.so*).

Tworzenie biblioteki współdzielonej

Proces tworzenia biblioteki współdzielonej przedstawiony został na Rys. 2.4. Składa się on z następujących etapów [11]:

- kompilacja plików źródłowych biblioteki do postaci obiektowej za pomocą *gcc* (listing 2.10).
Konieczne jest zastosowanie flagi **-fPIC**, pozwalającej na wygenerowanie tzw. **Position**

Independent Code - jest to taki kod maszynowy, który po załadowaniu do pamięci może być współdzielony przez kilka procesów. Wynikiem tej operacji są pliki obiektowe (*.o)

Listing 2.10. Kompilacja plików źródłowych biblioteki (zastosowanie flagi -fPIC)

```
user@host:~/complex$ gcc -fPIC -c *.c
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o
```

- utworzenie obiektu współdzielonego w procesie linkowania (listing 2.11). W tym celu należy zastosować flagę **-shared**. Należy również przekazać do wynikowego pliku *nazwę so* (*so-name*). Wartość pola *SONAME* w pliku biblioteki można następnie sprawdzić za pomocą narzędzia *objdump* z flagą *-p*. Podobnie jak w przypadku biblioteki statycznej możliwe jest również zbadanie występujących w niej symboli za pomocą narzędzia *nm* - z uwagi na bardziej skomplikowaną naturę samego pliku ich lista jest jednak dłuższa.

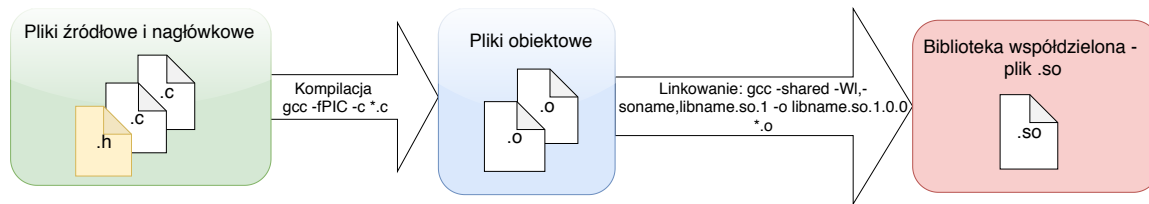
Listing 2.11. Utworzenie biblioteki współdzielonej, zbadanie wartości pola *SONAME* w utworzonym pliku

```
user@host:~/complex$ gcc -shared -Wl,-soname,libcoml.so.1 -o ↵
    libcoml.so.1.0.0 *.o
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o  ↵
    libcoml.so.1.0.0
user@host:~/complex$ objdump -p libcoml.so.1.0.0 | grep SONAME
    SONAME                libcoml.so.1
```

- utworzenie odpowiednich dowiązań symbolicznych. Istnieje kilka sposobów na linkowanie bibliotek współdzielonych. W zależności od zastosowanego sposobu potrzebne mogą okazać się inne dowiązania. W tym przypadku autorzy posłużyli się narzędziem **ldconfig** z flagą *-n* do wygenerowania dowiązania odpowiadającego nazwie *soname* biblioteki (listing 2.12).

Listing 2.12. Utworzenie za pomocą programu *ldconfig* dowiązania symbolicznego wskazującego na bibliotekę

```
user@host:~/complex$ ldconfig -n .
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o  ↵
    libcoml.so.1  libcoml.so.1.0.0
```



Rys. 2.4. Proces tworzenia biblioteki współdzielonej z uwzględnieniem koniecznych poleceń [7]

Dołączanie utworzonej biblioteki do programu

Proces dołączania utworzonej biblioteki współdzielonej do programu składa się z następujących etapów:

- dołączenie do źródeł programu plików nagłówkowych zawierających deklaracje stanowiące interfejs między programem a biblioteką (dyrektywa *include*, identycznie jak w przypadku biblioteki statycznej)
- kompilacja plików źródłowych programu do postaci obiektowej (również analogicznie jak w przypadku bibliotek statycznych)
- linkowanie w celu uzyskania pliku wykonywalnego (listing 2.13). W zaprezentowanym przykładzie użyte zostało stworzone wcześniej dowiązanie symboliczne *libcoml.so.1*.

Listing 2.13. Linkowanie w celu uzyskania pliku wykonywalnego zależnego od biblioteki współdzielonej

```
user@host:~/app$ gcc -o app app.o -L../complex -l:libcoml.so.1
user@host:~/app$ ls
app  app.c  app.o
```

Istnieje możliwość wyświetlenia, od jakich bibliotek współdzielonych jest zależny dany plik wykonywalny. Można to osiągnąć za pomocą narzędzie **ldd** [12], co zostało ukazane na listingu 2.14. Aktualnie stworzona przez autorów biblioteka jest oznaczona jako nie znaleziona (*not found*). Z tego też powodu próba uruchomienia programu zakończy się niepowodzeniem.

Listing 2.14. Użycie polecenia *ldd* i pierwsza, nieudana próba uruchomienia aplikacji zależnej od biblioteki współdzielonej

```
user@host:~/app$ ldd app
        linux-vdso.so.1 (0x00007ffffc16b6000)
        libcoml.so.1 => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3e5e8b0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3e5f000000)
user@host:~/app$ ./app
./app: error while loading shared libraries: libcoml.so.1: cannot open shared
        object file: No such file or directory
```

Uruchamianie programu zależnego od biblioteki współdzielonej

Poprzednia próba uruchomienia biblioteki zakończyła się porażką. Wynika to z tego, że program **loader** (linker dynamiczny) nie zna ścieżki, pod którą powinien znaleźć bibliotekę. Jednym z rozwiązań tego problemu jest dodanie tej ścieżki do zmiennej środowiskowej **LD_LIBRARY_PATH** [13]. Odpowiednie polecenie oraz poprawne uruchomienie programu ilustruje listing 2.15.

Listing 2.15. Dodanie ścieżki zawierającej bibliotekę do zmiennej *LD_LIBRARY_PATH* i poprawne uruchomienie przykładowej aplikacji

```
user@host:~/complex$ export LD_LIBRARY_PATH=$(pwd):$LD_LIBRARY_PATH
user@host:~/complex$ cd ../app/
user@host:~/app$ ./app
6.000000 3.700000
-1.000000 3.700000
```

2.3. Narzędzie CMake

CMake (*Cross-platform Make*) to narzędzie pozwalające na konfigurację procesu budowania oprogramowania (aplikacji oraz bibliotek) w sposób niezależny od platformy. Jego działanie opiera się na generowaniu pliku budującego natywnego dla określonej platformy [14] (dla systemów z rodziny UNIX jest nim *Makefile*) na podstawie przygotowanego przez użytkownika pliku *CMakeLists.txt*. Takie podejście w znacznym stopniu ułatwia tworzenie aplikacji multiplatformowych oraz pozwala na intuicyjne zarządzanie zależnościami w projekcie. Domyślnie CMake pracuje z językami C i C++, natomiast nowe wersje narzędzia wpierają ponadto m.in. język C# czy technologię CUDA [15]. Narzędzie to jest rozwijane i wspierane przez firmę *Kitware*.

Plik CMakeLists.txt

Jak wspomniano wyżej działanie narzędzia CMake opiera się na przygotowanym przez użytkownika pliku (lub zestawie plików rozmieszczonych w strukturze katalogów projektu) *CMakeLists.txt*. Plik ten zawiera polecenia napisane w specjalnie do tego celu przygotowanym języku

skryptowym. Użytkownik może za jego pomocą m.in. określać jakie pliki wykonywalne mają zostać wygenerowane podczas procesu budowania, wskazać lokalizację plików źródłowych czy określić zależności między komponentami projektu oraz bibliotekami zewnętrznymi.

Prosty przykład

Listing 2.16 zawiera przykład prostego pliku *CMakeLists.txt*, pozwalającego na zbudowanie napisanej w języku C++ obiektowej wersji klasycznego programu *Hello world*. Przykład ilustruje zastosowanie podstawowych poleceń CMake do określenia minimalnej wersji narzędzia, standardu języka C++, wynikowego pliku wykonywalnego oraz potrzebnych plików nagłówkowych.

Listing 2.16. Przykład prostego pliku CMakeLists.txt przeznaczonego do budowania programu napisanego w C++

```
# Określenie minimalnej wersji CMake
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)

# Określenie standardu języka C++
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Nazwa oraz wersja projektu
project(Hello VERSION 1.0)

# Dodanie pliku wykonywalnego, który powinien powstać
# wskutek procesu budowania
add_executable(Hello Main.cpp)

# Dodanie do projektu katalogu include wraz ze znajdującym się
# wewnątrz niego plikiem nagłówkowym
target_include_directories(Hello PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include")
```

Wersje CMake

CMake jest narzędziem, który w ciągu ostatnich kilku lat przechodzi gruntowne zmiany. Starsze wersje (np. 2.8) oparte są o prosty system zmiennych [16], co wprowadza szereg trudności w zarządzaniu dużymi projektami z wielopoziomowymi drzewami zależności. Dodatkowym problemem tych wersji jest również brak dobrze zdefiniowanych tzw. *dobrych praktyk* oraz nieprzystępna dla początkujących dokumentacja. Współczesne wersje narzędzie CMake (zwykle za takie uznaje się nowsze od wersji 3.0) opierają się na innym podejściu, opartym na strukturze projektu [16], co było przyczyną pojawienia się dla nich wyżej wspomnianych *dobrych praktyk*. Zalecane jest więc, by nowe projekty prowadzone były właśnie z użyciem nowszych wersji narzędzia.

Narzędzia CTest i CPack

CMake oferuje również możliwość konfiguracji sposobu testowania projektu. Służy do tego narzędzie *CTest*, dystrybuowane razem z podstawowym narzędziem CMake. Innym przydatnym modulem jest *CPack* - narzędzie to służy przygotowywaniu pakietów instalacyjnych z oprogramowaniem. Użycie obu wymienionych narzędzi polega na umieszczeniu w pliku *CMakeLists.txt* kilku przeznaczonych do tego komend.

2.4. Język Python

Python jest nowoczesnym, wysokopoziomowym językiem programowania, wspierającym takie paradygmaty jak programowanie obiektowe czy imperatywne. Działanie Pythona opiera się na dynamicznym systemie typów. Z założenia Python jest językiem przyjemnym w użytkowaniu, co przyczyniło się do jego dużej popularności [17]. Python jest szeroko stosowany jako język skryptowy - takie też zastosowanie znalazł w projekcie GGSS.

Prosty przykład

Listing 2.17 przedstawia prosty przykład skryptu napisanego w języku Python w wersji 3. Kod ten stanowi uproszczoną wersję skryptu zaprezentowanego w dalszej części pracy, którego zadaniem jest zbudowanie aplikacji w zależności od przekazanych przez użytkownika argumentów. Przykład prezentuje prosty skrypt przyjmujący jeden z dwóch możliwych argumentów i wypisujący informację na temat otrzymanego argumentu na standardowe wyjście.

Listing 2.17. Przykład prostego skryptu napisanego w języku Python 3 - przetwarzanie argumentów podanych przez użytkownika do skryptu

```
import argparse

## Definicja funkcji w języku Python
def parse_command_line_arguments():
    parser = argparse.ArgumentParser()
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument("-s", "--staticboost",
        help="Use static Boost linking.", action="store_true")
    group.add_argument("-d", "--dynamicboost",
        help="Use dynamic Boost linking.", action="store_true")
    return parser.parse_args()

if __name__=="__main__":
    arguments = parse_command_line_arguments()
    print(arguments)
```

Wersje języka Python

Python funkcjonuje w dwóch wersjach: Python 2 oraz 3. Wersje te nie są ze sobą w pełni kompatybilne, tzn. pewne funkcjonalności Pythona 2 nie są dostępne w Pythonie 3 i odwrotnie. Różnice znaleźć można również np. w domyślnym sposobie kodowania łańcuchów znakowych (ASCII w Pythonie 2, Unicode w Pythonie 3) oraz w wyniku dzielenia (za pomocą operatora `/`) dwóch liczb całkowitych (w Pythonie 2 wynikiem jest liczba całkowita, w Pythonie 3 liczba zmiennoprzecinkowa typu *float*) [18]. Ponadto zakończenie oficjalnego wsparcia Pythona w wersji 2 przewidziane jest na styczeń 2020 roku [19] - co w momencie pisania niniejszej pracy (grudzień 2019) jest terminem niedalekim i miało kluczowe znaczenie w czasie podejmowania pewnych decyzji projektowych.

Zewnętrzne biblioteki

Jedną z największych zalet Pythona jest bardzo duża liczba bibliotek zewnętrznych tworzonych przez społeczność Pythona. Rozbudowują one język o wiele nowych funkcjonalności, np. przetwarzanie plików HTML czy wykonywanie obliczeń numerycznych. W niniejszej pracy zastosowanych zostało kilka tego typu bibliotek, m.in. *Beautiful Soup* [20] do wspomnianego wyżej przetwarzania dokumentów w formacie HTML. Omówienie ich działania na przykładach znaleźć można w dalszej części pracy - przy opisie konkretnego ich zastosowania.

2.5. Powłoka systemu operacyjnego (Bash)

Powłoka systemu jest programem, którego głównym zadaniem jest udostępnienie interfejsu umożliwiającego łatwy dostęp do funkcji systemu operacyjnego. Nazwę *powłoka* zawdzięcza temu, że jest warstwą okalającą system operacyjny. Najczęściej spotykanym rodzajem powłoki są tzw. interfejsy z wierszem poleceń (ang. command-line interface). Polecenia wprowadzane są do nich w modzie interaktywnym, tj. wykonywane są one w momencie wprowadzenia końca linii.

Listing 2.18. Komenda wypisująca tekst na standardowe wyjście wykonana z linii poleceń

```
user@host:~$ echo "interfejs z linią poleceń"
interfejs z linią poleceń
user@host:~$
```

Bash, czyli **Bourne Again Shell** jest powłoką systemu początkowo napisaną dla systemu operacyjnego GNU. Obecnie Bash jest kompatybilny z większością systemów Unixowych, gdzie zwykle jest powłoką domyślną oraz posiada kilka portów na inne platformy, tj.: MS-DOS, OS/2, Windows [21]. Oprócz pełnienia wyżej wymienionej funkcji, Bash jest również językiem programowania pozwalającym na tworzenie skryptów, które są kolejną metodą wprowadzania poleceń do powłoki systemu.

Korzystając z języka skryptowego powłoki Bash jesteśmy w stanie zawrzeć dodatkową logikę podczas wykonywania komend. Wspiera on takie struktury jak: instrukcje warunkowe, pętle, operacje logiczne oraz arytmetyczne. Aby wykorzystać Bash w skrypcie należy na początku pliku zamieścić zapis `#!/bin/bash`, gdzie `/bin/bash` to ścieżka do pliku interpretera Bash. Zachowanie skryptu jesteśmy w stanie uzależnić od argumentów wykonania. Ich obsługa odbywa się za pomocą zapisu `$?`, gdzie `?` jest to numer porządkowy argumentu liczony od 0.

Listing 2.19. Skrypt wykorzystujący argumenty wejściowe, instrukcję warunkową oraz polecenie echo

```
#!/bin/bash
if [ $1 == "argumenty" ]; then
    echo "Argument 0.: $0"
    echo "Argument 1.: $1"
else
    echo "Nieznane polecenie"
fi
```

Listing 2.20. Przykład działania Skryptu z Listingu 2.19

```
user@host:~$ /home/user/prostySkrypt.sh argumenty
Argument 0.: /home/user/prostySkrypt.sh
Argument 1.: argumenty
```

Bash posiada wiele poleceń, które pozwalają na wykonywanie zarówno podstawowych, jak i bardziej zaawansowanych czynności, np.: obsługa plików, obsługa systemu katalogów, zarządzanie kontami, uprawnieniami, itd.

Bash posiada również wiele zaawansowanych funkcjonalności, które pozwalają na kontrolowanie przepływu informacji w trakcie wykonywania poleceń. Przykładem jest wpisywanie tekstu do pliku ukazane na Listingu 2.21.

Listing 2.21. Przykład zapisu tekstu do pliku

```
user@host:~$ echo "Ten napis zostanie zapisany do pliku plik.txt" > plik.txt
user@host:~$ cat plik.txt
Ten napis zostanie zapisany do pliku plik.txt
```

W celu zapisania tekstu do pliku należy na standardowe wyjście przekazać napis za pomocą komendy **echo**, a następnie przekierować za pomocą zapisu `>`, który poprzedza nazwę pliku docelowego. W wyniku działania zawartość pliku **plik.txt** zostanie nadpisana, a w przypadku gdy takiego pliku nie ma, to zostanie on utworzony i uzupełniony o napis.

2.6. System kontroli wersji Git i portal Gitlab

System kontroli wersji Git jest oprogramowaniem służącym do śledzenia i zarządzania zmianami w plikach projektowych. W przypadku Git'a, aby zarejestrować pliki projektowe w celu ich śledzenia należy wykonać kilka czynności. Po pierwsze wymagane jest utworzenie repozytorium. Sprowadza się ono do wykonania odpowiedniej komendy Git'a wewnątrz folderu projektu, tj. **git init**. Podczas działania komendy wewnątrz folderu, w którym wywołaliśmy ww. polecenie, inicjowany jest ukryty folder **.git**. Jest on odpowiedzialny za przechowywanie konfiguracji dla tego repozytorium oraz zapisywanie informacji o wszystkich zmianach dokonanych w projekcie.

Listing 2.22. Inicjalizacja repozytorium git

```
user@host:/ścieżka/do/projektu$ git init
Initialized empty Git repository in /ścieżka/do/projektu
user@host:/ścieżka/do/projektu$ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
```

Taka inicjalizacja nie spowoduje żadnego dodatkowego działania oprócz utworzenia repozytorium. Żadne pliki nie są jeszcze poddawane rewizji. W celu rejestracji plików należy wykonać jeszcze kilka kroków. Pierwszym z nich jest wykonanie komendy **git add**, która poprzedza nazwę plików lub folderów, które chcemy poddać wersjonowaniu. Elementy te zostają dodane do tzw. poczekalni, czyli są one kandydatami do utworzenia kolejnej rewizji. Przydatną komendą w tym przypadku jest również **git status** pozwalająca na sprawdzenie obecnego stanu repozytorium. Wyświetla ono krótkie podsumowanie nt. nowych plików, usuniętych plików oraz plików zmodyfikowanych. Informuje nas również o tym, które pliki są brane pod uwagę do utworzenia kolejnej rewizji.

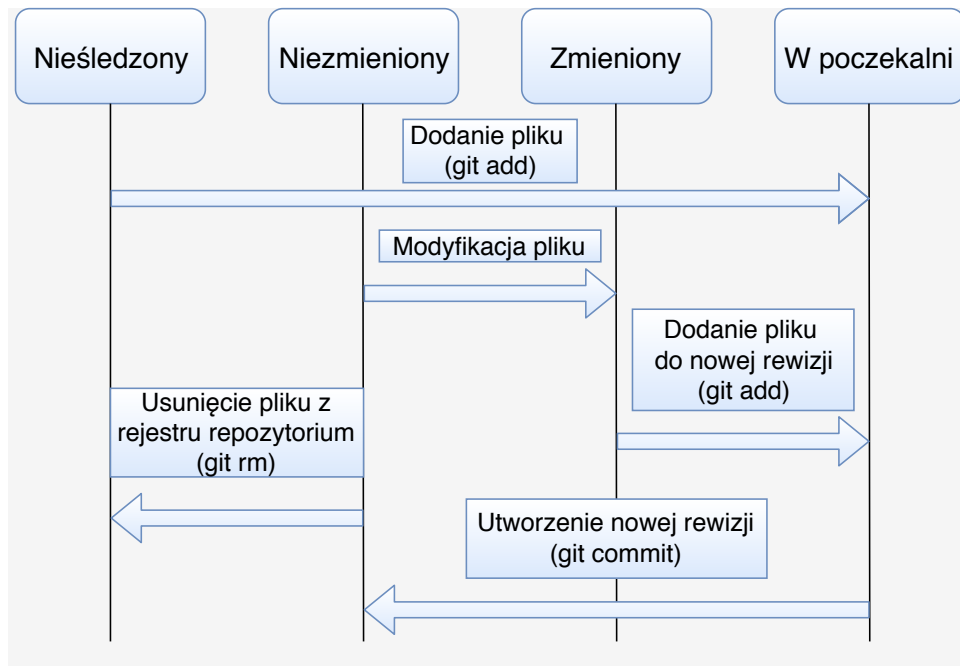
Listing 2.23. Dodawanie elementów do poczekalni

```
user@host:/ścieżka/do/projektu$ git add plik1 folder1 && git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   folder1/plik3
    new file:   folder1/plik4
    new file:   plik1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    plik2
```

Rys. 2.5. Możliwe stany pliku w repozytorium [22]

Tworzenie nowej wersji w ramach repozytorium odbywa się za pomocą komendy **git commit**. Sprowadza się do 'zamrożenia' obecnych wersji plików zarejestrowanych do rewizji oraz przypisanie im wspólnego, unikalnego dla każdej z nich, identyfikatora. Git udostępnia komendy pozwalające na przeglądanie oraz przywracanie plików do wcześniej utworzonych wersji. Listing 2.24 przedstawia utworzenie nowej wersji oraz wyświetlenie podsumowania o utworzonych do tej pory rewizjach.

Listing 2.24. Utworzenie nowej rewizji

```

user@host:/sciezka/do/projektu$ git commit -m "Pierwsza rewizja"
user@host:/sciezka/do/projektu$ git log
commit 1d2445e961beb25940dffa9d73963f887ee553ad
Author: user <user@localdomain>
Date:   Wed Dec 4 17:29:55 2019 +0100

    Pierwsza rewizja
  
```

Przejścia między rewizjami nie powodują utraty danych, gdyż zachowywana jest informacja o stanie plików dla każdej z nich, co ukazuje Listing 2.25. Tworzona jest nowa rewizja zawierająca dodatkowo **plik2**, natomiast po powrocie do poprzedniej wersji plik ten nie występuje. Gdy powrócimy do nowszej wersji ponownie pojawi się **plik2**.

Listing 2.25. Podsumowanie rewizji, powrót do starszej wersji

```
user@host:/sciezka/do/projektu$ git add plik2
user@host:/sciezka/do/projektu$ git commit -m "Druga rewizja"
user@host:/sciezka/do/projektu$ git log
commit b58836df55fc2a8eb2a43aa96273853776924807
Author: user <user@localdomain>
Date:   Wed Dec 4 17:30:10 2019 +0100

    Druga rewizja

commit 1d2445e961beb25940dffa9d73963f887ee553ad
Author: user <user@localdomain>
Date:   Wed Dec 4 17:29:55 2019 +0100

    Pierwsza rewizja

user@host:/sciezka/do/projektu$ ls
folder1 plik1 plik2
user@host:/sciezka/do/projektu$ git checkout ↵
1d2445e961beb25940dffa9d73963f887ee553ad
user@host:/sciezka/do/projektu$ ls
folder1 plik1
user@host:/sciezka/do/projektu$ git checkout ↵
b58836df55fc2a8eb2a43aa96273853776924807
user@host:/sciezka/do/projektu$ ls
folder1 plik1 plik2
```

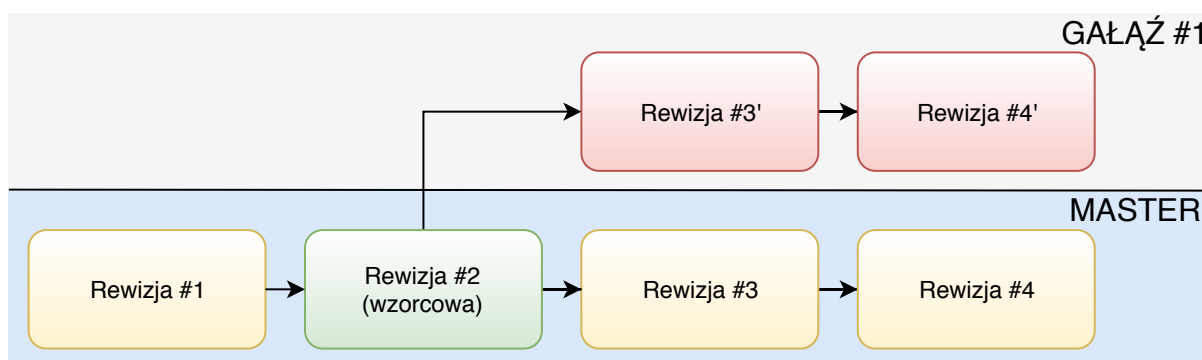
Głównym celem portalu **GitLab** jest udostępnienie środowiska do przechowywania repozytoriów Git'owych na zdalnych serwerach. Pozwala to na uniezależnienie się od maszyny na której pracujemy, zwiększa bezpieczeństwo plików źródłowych poprzez umieszczenie kopii na zdalnym serwerze oraz wspiera zespołową pracę nad kodem.

Ze względu na to, że portale typu **GitLab** są traktowane jako podstawowe narzędzie do wspólnej pracy nad kodem, to rozwinęły one wiele narzędzi wspomagających organizację oraz śledzenie pracy. Oprócz ww. funkcji **Gitlab** dostarcza wiele narzędzi do wspomagania procesu zapewniania jakości, jak i automatyzacji dostarczania kodu.

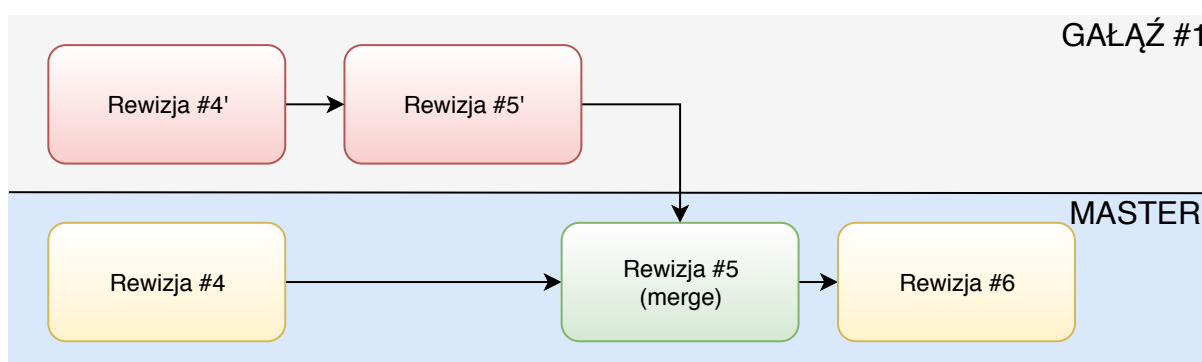
Git posiada specjalne komendy pozwalające na przekazywanie oraz pobieranie repozytoriów z ww. portali, czyli:

- **git clone**, która pozwala na pobranie repozytorium z portalu oraz zainicjalizowanie go lokalnie
- **git pull** - za pomocą tej komendy możemy zaktualizować repozytorium lokalne do najnowszej rewizji, która znajduje się na portalu
- oraz komenda **git push** aktualizująca zdalne repozytorium do najnowszej lokalnej wersji

Bardzo ważnym aspektem technologii **Git** jest mechanizm gałęzi (ang. *branch*). Pozwala on na równoległą pracę nad jedną częścią kodu. Polega to na tym, że w momencie utworzenia nowej gałęzi tworzymy tak na prawdę osobny łańcuch rewizji, których jedyną wspólną częścią jest rewizja wzorcowa, czyli wersja na podstawie której została utworzona gałąź, co przedstawia Rysunek 2.6. W celu utworzenia nowej gałęzi należy wykorzystać komendę **git branch <nazwa-gałęzi>**, a w celu zmiany aktualnie używanej gałęzi **git checkout <nazwa-gałęzi>**. Główna gałąź projektu nazywa się **master**. Zmiany na jednej gałęzi nie wpływają w żaden sposób na inną gałąź do momentu tzw. **merge**, czyli złączenia dwóch gałęzi, co ukazuje rysunek 2.7. Ważnym aspektem łączenia gałęzi jest rozwiązywanie konfliktów, które pojawiają się w momencie, gdy łączone gałęzie wprowadzały zmiany do tego samego pliku, a system rozwiązujący konflikty nie był w stanie wykonać swojego zadania automatycznie. W takim przypadku wymagana jest akcja programisty, aby w gałęzi wynikowej zachować odpowiedni kod.



Rys. 2.6. Schemat tworzenia rewizji w ramach nowej gałęzi



Rys. 2.7. Schemat złączania gałęzi w ramach polecenie **git merge**

Jest to tylko podstawowy opis technologii jaką jest **Git**, przykłady bardziej zaawansowanych zastosowań pojawią się w części manuskryptu przeznaczony na prezentację wykonanych prac w ramach projektu.

2.7. Menadżer pakietów RPM

Menadżer pakietów jest zbiorem oprogramowania, które w sposób automatyczny zarządza instalacją, aktualizacją, konfiguracją oraz usuwaniem programów komputerowych [23]. Ze względu na to, że procesy te różnią się w zależności od systemów operacyjnych oraz ich dystrybucji istnieje wiele menadżerów pakietów.

Zastosowanie technologii zarządzania pakietami pozwala znacząco zmniejszyć próg wejścia wynikający z użycia wcześniej niewykorzystywanego oprogramowania. Pozwala on odejść od żmudnego procesu ręcznej instalacji zależności oraz konfiguracji środowiska. Dzięki menadżerom wszystko jest wykonywane automatycznie. Jeżeli w trakcie procedur nie wystąpi żaden problem, to pakiet, którego zleciliśmy instalację, powinien być od razu gotowy do działania. Jeżeli domyślna konfiguracja, jaka zostanie nam zapewniona w podczas działania menadżera pakietów, nie będzie dla nas odpowiednia możemy dokonać jej modyfikacji po procesie instalacji.

RPM, czyli RedHat Package Manager jest darmowym, open-source'owym menadżerem pakietów dla systemów z rodziny RedHat oraz SUSE, czyli m.in.:

- RedHat Linux
- CentOS
- Fedora
- openSUSE

RPM jest domyślnym menadżerem pakietów dla ww. dystrybucji. Obsługuje on pakiety w ramach formatu **.rpm**. Pakiety **.rpm** zawierają w sobie wiele ważnych elementów. Po pierwsze wewnątrz nich przechowywane są dane aplikacji, czyli: pliki wykonywalne, dokumentacja, testy, konfiguracja.

Kolejnym ważnym elementem są informacje o zależnościach, czyli innych wymaganych pakietach, które pozwalają na automatyzację procesu instalacji. W momencie, gdy któraś z zależności jest niespełniona menadżer pakietów stara się odnaleźć, w bazie danych pakietów, odpowiedni wpis, aby **pobrać** oraz **zainstalować** brakujące oprogramowanie. Trzecim ważnym elementem jest logika pakietu, która jest podstawą do realizacji akcji wykonywanych przez menadżer pakietów, dostarczona w postaci skryptów powłoki, np. **bash**, zaszytych wewnątrz pliku *.rpm.

Listing 2.26 pokazuje przykładową zawartość pakietu, czyli moduł kernela **modułAplikacji**, plik w formacie **JSON** pozwalający na konfigurację aplikacji oraz plik wykonywalny **aplikacji**, natomiast listing 2.27 pokazuje przykładową logikę pakietu RPM w postaci skryptów shell. Skrypty te nie zawierają ani kopiowania, ani usuwania plików zawartych w pakiecie, proces ten odbywa się automatycznie na podstawie ścieżek ukazanych na Listingu 2.26 w trakcie instalacji/dezinstalacji.

Listing 2.26. Przykładowa zawartość pakietu RPM

```
user@host:~$ rpm -qpl pakiet.rpm
/ścieżka
/ścieżka/do
/ścieżka/do/konfiguracjaAplikacji.json
/ścieżka/do/aplikacji
/usr
/usr/lib
/usr/lib/modules
/usr/lib/modules/3.10.0-862
/usr/lib/modules/3.10.0-862/extra
/usr/lib/modules/3.10.0-862/extra/modułAplikacji.ko
```

Listing 2.27. Skrypty pakietu RPM

```
user@host:~$ rpm -qp --scripts pakiet.rpm
preinstall program: /bin/sh
postinstall scriptlet (using /bin/sh):

#!/bin/sh
echo "Post-instajacja przykładowego pakietu"
echo "Przypisywanie uprawnień"
chmod 777 /ścieżka/do/konfiguracjaAplikacji.json
echo "Ładowanie modułu aplikacji"
/sbin/modrpobe modułAplikacji

preuninstall scriptlet (using /bin/sh):
#!/bin/sh
echo "Odinstalowywanie przykładowego pakietu"
echo "Usuwanie modułu aplikacji"
/sbin/rmmmod modułAplikacji

postuninstall program: /bin/sh
```

2.8. Technologie wirtualizacji i konteneryzacji

Wirtualizacja, czyli proces uruchamiania instancji wirtualnego systemu komputerowego odseparowanego od rzeczywistego systemu komputerowego oraz jego sprzętu (ang. hardware). Pozwala na uruchomienie **wielu różnych** systemów operacyjnych na jednym komputerze **jednocześnie**. Wykorzystywany przede wszystkim do separacji środowisk dla aplikacji, czy też całych systemów. Pozwala na uruchomienie oprogramowania nieprzystosowanego do naszego systemu operacyjnego, wystarczy utworzyć instancję maszyny wirtualnej z odpowiednim systemem operacyjnym. Aplikacje uruchamiane w takiej instancji zachowują się tak, jakby znajdowały się na **odseparowanym komputerze** z własnym, dedykowanym systemem operacyjnym, biblio-

tekami oraz innym oprogramowaniem. Dużym plusem jest pełna separacja instancji uruchomionych na systemie gospodarza. Procesy jednej instancji **nie mają wpływu** na drugą instancję [24].

Proces wirtualizacji odbywa się za pomocą oprogramowania, które nazywa się hipernadzorcą (ang. *hypervisor*). Odpowiada on za zapewnienie środowiska, które pozwoli na uruchomienie maszyny wirtualnej. Wyróżniane są dwa rodzaje hipernadzorców. Pierwsze z nich bazują na wspomaganiu procesu przez fizyczny sprzęt, co pozwala na częściowe ominięcie systemu operacyjnego gospodarza, dzięki czemu narzut na wydajność jest mniejszy, natomiast drugie bazują na rozwiązaniach aplikacyjnych, dzięki czemu można je uruchamiać bez wsparcia sprzętowego, natomiast są znacznie mniej wydajne.

Konteneryzacja jest procesem utworzenia odseparowanego kontenera, czyli ustandaryzowanej jednostki, która zawiera w sobie oprogramowanie oraz zależności wymagane do uruchomienia aplikacji, w celu której została utworzona [25]. Kontenery są tworzone na podstawie obrazu, czyli wzorcowego środowiska, które zostało zamrożone w celu późniejszego odtworzenia. W przypadku **Dockera** obrazy te są tworzone na podstawie tzw. **Dockerfile**. Wewnątrz takiego pliku zapisywane są informacje o krokach podejmowanych w celu utworzenia obrazu, np.:

- informacje o bazowym systemie operacyjnym
- informacje o zmiennych środowiskowych
- komendy menadżera pakietów w celu instalacji zależności

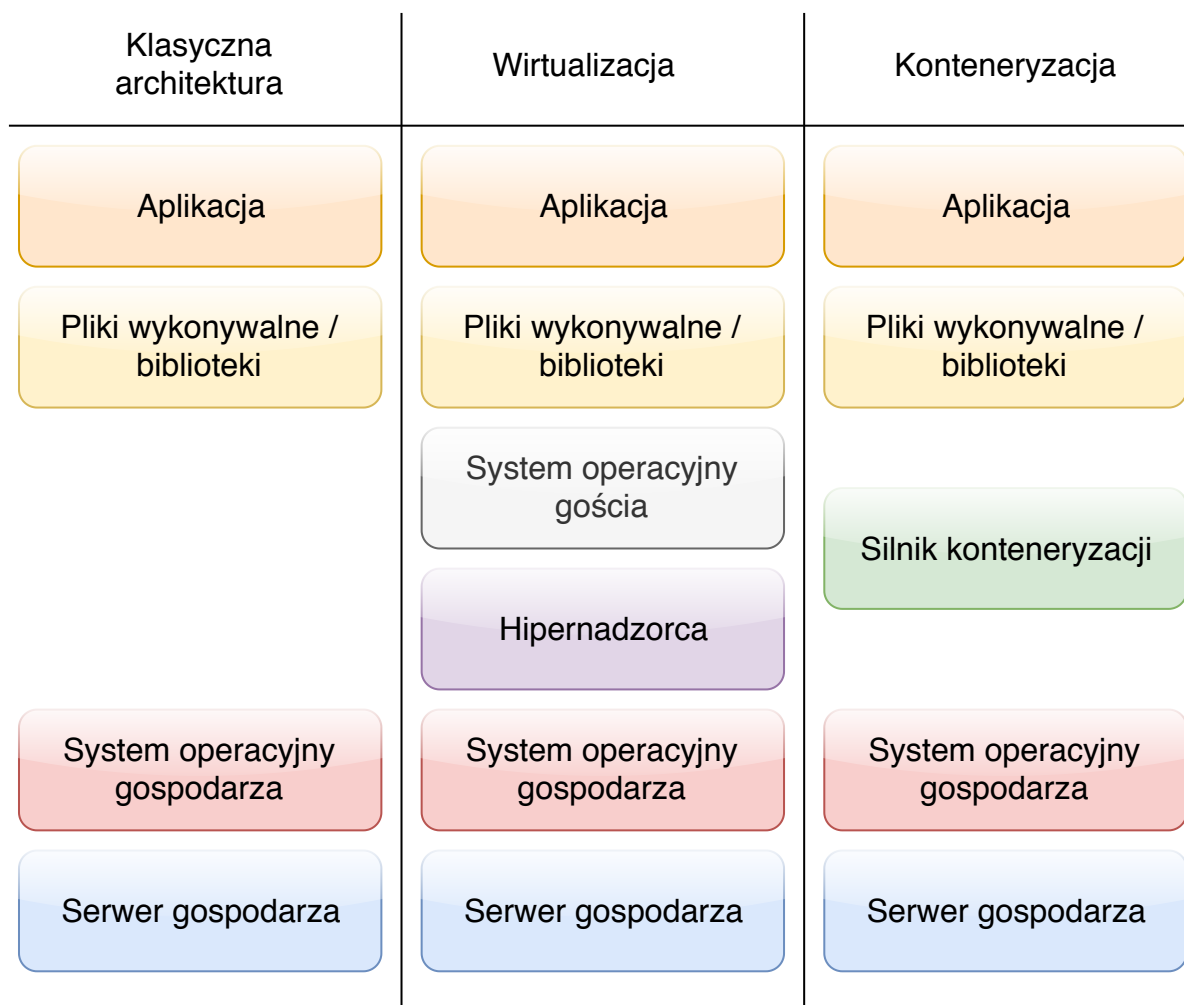
Informacje te są poprzedzone odpowiednimi słowami kluczowymi, np.: **ENV**, czy **RUN**. Listing 2.28 ukazuje przykładowy Dockerfile, którego użycie, za pomocą odpowiedniej komendy Dockera, spowoduje utworzenie obrazu bazującego na dystrybucji centos7 z zainstalowanym kompilatorem języka c++.

Listing 2.28. Przykładowy Dockerfile

```
FROM cern/cc7-base:latest  
  
RUN yum -y install gcc-c++
```

Zasadniczą różnicą między konteneryzacją, a wirtualizacją, jest używanie jądra systemu operacyjnego gospodarza w celu zapewnienia wymaganych funkcjonalności. Pomimo tego, że wykorzystywane jest to samo jądro systemu, to kontenery są w pełni odseparowane od siebie. W przypadku Linuksa osiągnięte jest to za pomocą **cgroups**, które pozwalają na limitowanie zasobów przypadających na grupę, oraz **przestrzeni nazw**, które służą do separacji procesów. Procesy wykonywane w ramach przestrzeni nazw „nie widzą” procesów z poza tej przestrzeni. Ze względu na brak warstwy pośredniej, jaką w przypadku wirtualizacji jest system operacyjny gościa oraz warstwa integracji tego systemu z systemem gospodarza, kontenery są znacznie szybsze zarówno

w kwestii tworzenia, uruchamiania, jak i działania. Minusem natomiast jest przywiązanie do jądra gospodarza, ze względu na to nie jesteśmy w stanie podmienić wersji jądra w obrazie, czy też nie jesteśmy w stanie w prosty sposób uruchomić systemu operacyjnego z niezgodnym jądrem (np.: kontener Windows na systemie operacyjnym Linuks). W przypadku niezgodności jąder wymagana jest dodatkowa warstwa w postaci maszyny wirtualnej, która zapewni nam odpowiedni rodzaj jądra systemu operacyjnego. Obecnie takie rozwiązanie jest stosowane w aplikacjach wykorzystujących technologię **Docker** na Windows, np.: *Docker-for-Windows*. [26] [27]



Rys. 2.8. Porównanie klasycznej architektury z technologiami wirtualizacji oraz konteneryzacji [28]

3. Stan początkowy projektu

Niniejszy rozdział został przygotowany w oparciu o wiedzę przekazaną przez opiekuna pracy dra hab. inż. Bartosza Mindura, pracę magisterską Pana Przemysława Pluteckiego pt. *Aktualizacja oprogramowania oraz sprzętu elektronicznego dla Systemu Stabilizacji Wzmocnienia Gazowego* [29] oraz pracę magisterską Pana Pawła Zadroźniaka pt. *Aktualizacja sprzętu elektronicznego dla Systemu Stabilizacji Wzmocnienia Gazowego* [30].

3.1. Architektura

Główna logika projektu została wykonana z użyciem języka C++, dzięki czemu aplikacja jest szybka i wydajna. Część kodu źródłowego została napisana z użyciem standardu 11. Wykorzystano również zewnętrzne biblioteki takie jak biblioteka Boost, czy też GSL. Początkowa architektura projektu oparta była o płaską strukturę folderów. Konwencja jaka została przyjęta, to **nazwaLib** dla folderów zawierających pliki bibliotek oraz **__nazwa** dla folderów zawierających aplikacje użytkowe.

Biblioteki, które były zawarte w ramach projektu, to:

- ggssLib - biblioteka zawierająca logikę aplikacji **ggssrunner**, czyli głównego programu wykonywalnego projektu
- fifoLib - biblioteka implementująca kolejkę FIFO (first in first out)
- FitLib - biblioteka wspierająca obliczenia numeryczne z wykorzystaniem biblioteki zewnętrznej **GSL**
- utilsLib - biblioteka zawierająca pliki pomocnicze projektu
- xmlLib - biblioteka dokonywująca parsowanie pliku konfiguracyjnego aplikacji, który jest napisany w formacie **xml**
- handleLib - biblioteka realizująca obsługę mechanizmu sygnałów i slotów
- logLib - biblioteka odpowiedzialna za mechanizm logowania

- ThreadLib - biblioteka odpowiadająca za wykorzystanie wielowątkowości w aplikacji
- CaenHVLib - biblioteka okalająca (ang. wrapper) bibliotekę **CaenN1470Lib**
- CaenN1470Lib - biblioteka implementująca komunikację z urządzeniem sprzętowym (**CAEN N1470**)
- OrtecMcbLib - biblioteka okalająca (ang. wrapper) bibliotekę **mcaLib**
- mcaLib - biblioteka odpowiedzialna za komunikację z urządzeniem sprzętowym (**CAEN N957**)
- usbrmLib - biblioteka, której zadaniem jest komunikacja z multiplekserami
- daemonLib - biblioteka umożliwiająca uruchamianie aplikacji **ggss-runner** w postaci **daemon**, czyli „w tle”

Aplikacje, które były zawarte w ramach projektu, to:

- __ggss - aplikacja ggssrunner, która odpowiada za człon projektu, jest to główny program wykonywalny
- __dimCS - aplikacja odpowiedzialna za wysyłanie komend, za pomocą technologii *DIM*
- __ggssspector - aplikacja, której zadaniem zapisywanie parametrów pracy, np.: ustawione napięcie, odczytane napięcie, ciśnienie itp
- caen_n957 - aplikacja odpowiadająca za testowanie poprawności działania analizatora wielokanałowego oraz jego sterowników

Oprócz opisanych wcześniej katalogów występowały również:

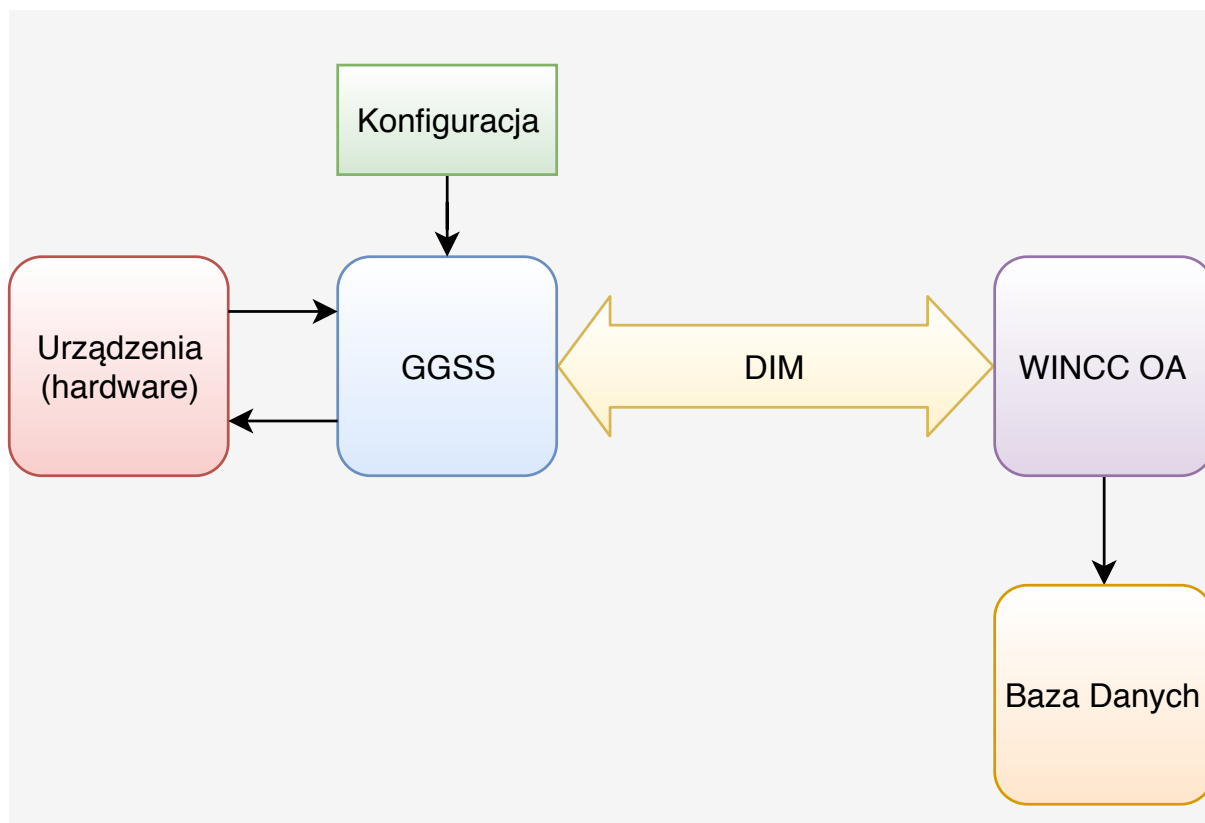
- zawierające pliki nagłówkowe wspólne dla wielu bibliotek (include)
- zawierające biblioteki zewnętrzne (lib)
- zawierające konfigurację aplikacji oraz skrypt pomocniczy w języku Python dla biblioteki serial (misc)
- zawierające skrypty pomocnicze w języku Python i Bash oraz skrypty watchdog (obserwatorzy)

Dodatkowo projekt wykorzystywał następujące biblioteki zewnętrzne:

- Boost - biblioteka C++ ogólnego przeznaczenia
- GSL (GNU Scientific Library) - biblioteka numeryczna dla języka C oraz C++
- DIM - biblioteka wspierająca komunikacja protokołem *DIM*

- QT oraz QWT - biblioteki wykorzystywane w ramach modułu GGSS Spector

W ramach projektu został również przygotowany panel informacyjno-administracyjny w technologii Scada (WinCC OA), który komunikuje się z główną częścią aplikacji za pomocą protokołu *DIM*. Rysunek 3.1 przedstawia wysokopoziomową architekturę wraz z przepływem informacji.



Rys. 3.1. Wysokopoziomowa architektura projektu GGSS

W ramach architektury projektu utworzone zostały również pliki **CMake**, które służyły jako szablon akcji wykorzystywanych wielokrotnie w pozostałych miejscach, np.: wyszukiwanie odpowiedniej biblioteki.

Architektura projektu charakteryzowała się całkowicie płaską strukturą, nie było żadnej gradacji bibliotek, z której nie wynikały żadne zależności między modułami. Plusem takiego rozwiązania była łatwość w budowie całego systemu (np. brak problemów ze skomplikowanymi ścieżkami), aczkolwiek nie wynikała z niego żadna informacja nt. systemu ze względu na co próg wejścia, czy też utrzymanie projektu jest utrudnione. Widoczny był podział na moduły, lecz nie był on w żaden sposób uporządkowany, np. ze względu na przeznaczenie bibliotek (programowe, sprzętowe).

Projekt zawierał również moduł przeznaczony obsłudze sterownika projektu GGSS (ggss-driver). Zawierał on archiwum z sterownikiem dla urządzenia firmy Caen (CAEN N957) dostarczanym przez ww. firmę. W ramach modułu został również zawarty plik *CMake*, którego zada-

niem było przygotowanie pakietu RPM w skład którego wchodził sterownik, biblioteki od firmy Caen (libCAENN957) oraz pre-generowane skrypty pozwalające na instalację oraz dezinstalację pakietu. W celu sprawdzenia poprawności działania pakietu odpowiedzialnego za instalację sterownika został utworzony osobny moduł (mcaN957), którego zadaniem jest zebranie napięć rejestrowanych przez analizator wielokanałowy (MCA) oraz zapisanie ich w pliku tekstowym w postaci zgrupowanej zakresami. Aplikacja (mcaN957) przyjmuje dwa parametry wejściowe jakimi są czas działania oraz wartość progowa rejestrowania napięć.

3.2. Budowanie

Projekt w swojej oryginalnej postaci budowany był za pomocą narzędzia **CMake** w wersji **2.8**. Wyróżnić można było jeden nadrzędny plik *CMakeLists.txt* znajdujący się w katalogu głównym projektu oraz pomniejsze pliki dla każdego z modułów.

Nadrzędny plik *CMakeLists.txt* pozwalał na budowanie aplikacji: *ggssrunner*, *dimCS* oraz opcjonalnie *ggsspector*. Zawierał on zmienną *PROJECTS* przechowującą listę zależności, czyli nazwy wszystkich stworzonych w ramach projektu bibliotek (listing 3.1). Było to jedyne miejsce w całym systemie budowania określające wewnętrzne zależności w projekcie.

Listing 3.1. Fragment oryginalnego pliku *CMakeLists.txt* znajdującego się w katalogu głównym pierwotnej wersji projektu przedstawiający zmienną *PROJECTS* zawierającą listę bibliotek [29]

```
set (PROJECTS
    logLib
    xmlLib
    utilsLib
    handleLib
    ThreadLib
    fifoLib
    FitLib
    OrtecMcbLib
    CaenHVLib
    ggssLib
    usbrmLib
    CaenN1470Lib
    mcaLib
    daemonLib
)
```

Pliki *CMakeLists.txt* poszczególnych komponentów projektu były bardzo proste (przykład na listingu 3.2). Pliki te nie były jednak niezależne od głównego pliku *CMake*: nie znajdowały się w nich informacje na temat wewnętrznych zależności projektu czy ścieżek do użytych plików *.cmake*. Sprawiało to, że niemożliwym było użycie ich do zbudowania pojedynczego komponentu projektu w sposób niezależne od całości.

Listing 3.2. Oryginalny pliku *CMakeLists.txt* służący do budowania biblioteki *ThreadLib* oraz zawartość pliku *LibraryTemplate.cmake* [29]

```
# Plik CMakeLists.txt dla biblioteki ThreadLib
include(LibraryTemplate)
target_link_libraries (ThreadLib ${Boost_LIBRARIES})

# Zawartość pliku LibraryTemplate.cmake
project (${dir_name})
file (GLOB subdirectory_files ${dir_name} *.h *.cpp *.hxx *.cxx *.c)
add_library (${dir_name} STATIC ${subdirectory_files})
set_target_properties (${dir_name} PROPERTIES LINKER_LANGUAGE CXX)
```

Projekt posiadał zatem tylko jeden, z góry określony sposób budowania - budowanie **wszystkiego za pomocą głównego pliku CMakeLists.txt**. Pomimo podziału projektu na komponenty (biblioteki) na poziomie architektury nie było możliwości budowania ich osobno (za pomocą dostarczonego systemu). Zaletą takiego podejścia jest jego prostota - pliki *CMakeLists.txt* są krótkie, odwołują się zwykle do jednego szablonu. Istnieje jednak szereg wad takiego rozwiązania. Jedną z nich jest brak łatwej możliwości identyfikacji wewnętrznych zależności w projekcie - niemożliwe jest np. wywnioskowanie zależności biblioteki **ThreadLib** od bibliotek **handleLib** i **logLib**. Jediną podpowiedzią na ten temat mogłaby być kolejność ich występowania w zmiennej *PROJECTS* w głównym pliku *CMakeLists.txt* - taka informacja nie jest jednak informacją pełną i tego typu rozumowanie nie zawsze byłoby prawdziwe. Innym ograniczeniem tego rozwiązania był brak możliwości łatwego podziału projektu na niezależne repozytoria. Nie było również możliwość zbudowania aplikacji wchodzących w skład projektu niezależnie od siebie - dedykowane im pliki *CMakeLists.txt* korzystają z elementów zdefiniowanych w pliku głównym (np. ze wspomnianej już listy zależności czy ścieżek wskazujących lokalizację plików nagłówkowych).

Projekt zawierał kilka plików o rozszerzeniu *.cmake* definiujących szablony możliwe do wykorzystania w wielu miejscach systemu budującego. Udostępniały one funkcjonalności takie jak: identyfikacja systemu operacyjnego, wyszukanie w systemie potrzebnych bibliotek zewnętrznych czy przeprowadzenie podstawowych operacji koniecznych do stworzenia biblioteki statycznej. Zawartość jednego z nich (*LibraryTemplate.cmake*) widoczna jest na listingu 3.2.

3.3. Dostarczanie i uruchamianie

Projekt nie miał automatycznego systemu dostarczania gotowych do użycia plików binarnych, czy też pakietów RPM. Wszystkie akcje prowadzące do utworzenia odpowiednich plików trzeba było wykonywać ręcznie. Nie dostarczono również żadnego systemu regresji, który automatycznie testowałby wprowadzane zmiany. Uruchamianie projektu odbywało się poprzez ręczne umieszczenie plików w środowisku docelowym, a następnie uruchomienie aplikacji za pomocą skryptów pomocniczych. Aplikacja była również uruchamiana za pomocą zadania cyklicznego

Linux (program crontab). Zadanie to sprawdzało, czy działał proces aplikacji ggssrunner, jeżeli takowy nie istniał oraz nie został utworzony plik .lock, to aplikacja była uruchamiana.

3.4. Kontrola wersji

Technologią kontroli wersji zastosowaną był do tej pory **SVN** (Subversion). Mgr. Plutecki w swojej pracy wspomina o wykorzystaniu technologii **Git**, natomiast była ona prawdopodobnie używana jedynie w trakcie prac. Wniosek taki został wysnuty z powodu braku jakichkolwiek śladów wykorzystania tej technologii w projekcie. Wykorzystane zostały jedynie podstawowe funkcjonalności kontroli wersji, brak jakiegokolwiek mechanizmu pozwalającego na automatyczne wersjonowanie, czy kontrolę tworzonych rewizji.

4. Stan docelowy projektu

Niniejszy rozdział zawiera opis docelowej wersji systemu GGSS, jaka powinna zostać osiągnięta po zakończeniu przez autorów prac. Cele do zrealizowania podzielone zostały na dwie główne części, wynikające z organizacji czasowej prac tzn. wkład autorów w system nie zamyka się wraz z zakończeniem prac nad niniejszym manuskrypcem. Z tego powodu niniejszy rozdział podzielony został na dwie części - pierwsza z nich opisuje finalną wersję projektu, natomiast druga - wersję po zakończeniu prac w ramach projektu inżynierskiego.

4.1. Finalna wersja projektu

Projekt w swojej wersji finalnej ma charakteryzować się modularną architekturą zarówno jeśli chodzi o organizację kodu, jak i sposób jego budowania. Pozwala to na proste i efektywne testowanie każdego komponentu z osobna. Ułatwia to również podmianę komponentów w środowisku produkcyjnym. Większa modularyzacja pozwala skrócić czas poszukiwania źródła ewentualnych błędów w działaniu systemu. Z drugiej natomiast strony podział systemu na dużą liczbę komponentów utrudnia proces budowania, przez co wymagana jest jego znacząca automatyzacja. Konieczne jest przygotowanie więc prostej w użytkowaniu infrastruktury wspomagającej proces produkcyjny. Powinna być ona dobrze udokumentowana, by próg wejścia do projektu był możliwie niski. Powinny więc zostać przygotowane instrukcje w języku angielskim zawierające zestaw najczęściej używanych komend wraz z wariantami ich użycia (np. flagi). Kluczowym celem jest również modernizacja kodu źródłowego - zarówno jeśli chodzi o jego jakość, jak i zastosowane technologie. Projekt charakteryzować się ma więc ustandaryzowanym, ogólnie przyjętym przez społeczność programistów jako tzw. *dobre praktyki*, nazewnictwem, odpowiednim podziałem na poziomie kodu źródłowego (funkcje, klasy itp.). W swojej ostatecznej wersji projekt powinien bazować na najnowszych, dostępnych w ramach infrastruktury produkcyjnej CERN-u, technologiach, np. standard języka C++. Dzięki temu zależności zewnętrzne powinny zostać ograniczone do minimum, na rzecz standardowych rozwiązań (np. biblioteka standardowa), by zagwarantować możliwie dużą przenośność. Zaplanowano również rozszerzenie projektu o nowe komponenty ułatwiające korzystanie z systemu (np. graficzny interfejs użytkownika).

4.2. Stan oczekiwany w ramach projektu inżynierskiego

Z uwagi na brak możliwości realizowania wszystkich powyższych postulatów dotyczących celów pracy w ramach projektu inżynierskiego (co wynika z ograniczonego czasu), wybrany został następujący podzbiór wymagań:

- przygotowanie środowiska umożliwiającego zarządzanie prowadzonym projektem
- modularyzacja projektu (z poziomu architektury i systemu budowania *CMake*)
- przygotowanie infrastruktury automatyzującej proces produkcyjny, zapewniającej spójne środowisko do testowania
- wykonanie dokumentacji zgodnej z wymienionymi założeniami
- wprowadzenie standardu nazewnictwa na poziomie procesu budowania i podziału na repozytoria
- przeprowadzenie testów wynikowego produktu

Rezultatem zakończenia tej części prac powinien być w pełni działający, udoskonalony system.

5. Ograniczenia dostępnej infrastruktury

Z uwagi na silny związek oprogramowania GGSS z infrastrukturą CERN oraz wymóg zapewnienia możliwości budowania projektu na należących do niej maszynach, przed autorami postawiony został szereg ograniczeń związanych z możliwymi do użycia technologiami oraz sposobem wykonywania pewnych operacji. Niniejszy rozdział stanowi opis najważniejszych z tych ograniczeń z uwzględnieniem ich wpływu na obraną przez autorów pracy ścieżkę rozwoju projektu.

5.1. Ograniczone uprawnienia w środowisku docelowym

Ze względu na ograniczone uprawnienia w środowisku docelowym proces wprowadzania zmian w systemie **GGSS** wymagał komunikacji z administratorami środowiska. Każdorazowa instalacja poprawek dla modułu sterownika (**ggss-driver**) wymagała autoryzacji oraz przeprowadzenie procesu instalacji przez osobę upoważnioną. Ze względu na to zostało przygotowane specjalne środowisko pozwalające na testowanie pakietów **RPM** przed ich instalacją w środowisku docelowym. Dodatkowo uprawnienia autorów w środowisku docelowym są ograniczone ze względu na możliwości instalacji dodatkowych pakietów, czy też modułów. Nie jest możliwe również budowanie komponentów w środowisku docelowym, restart maszyny produkcyjnej, czy też ładowanie/usuwanie modułów spoza listy określonej przed administratorów. Wszystkie te czynniki negatywnie wpłynęły na czas, który był wymagany w celu wykonania testów w środowisku produkcyjnym oraz wykonane zmiany w projekcie.

5.2. Wersje kompilatorów i interpreterów

Dostępne wersje kompilatorów i interpreterów stanowią jeden z kluczowych czynników, który należy uwzględnić podczas wprowadzania zmian w istniejącym systemie, ponieważ definiują one możliwy do wykorzystania podzbiór technologii. W kontekście systemu GGSS ograniczenia te dotyczą przede wszystkim kompilatora języka C++ oraz interpretera języka Python.

Dostępna w ramach infrastruktury projektu wersja kompilatora języka C++ to **g++ (GCC) 4.8.5**. Wspiera ona w pełni standard C++11, czyli funkcjonalności takie, jak referencje

do r-wartości, wyrażenia `lambda` czy zakresowa pętla `for` [31]. Wersja ta nie wspiera niestety nowszych wydań języka (C++14/17).

Domyślną wersją Pythona jest **Python 2.7.5**, jednak dostępny jest również Python 3 (w wersji **Python 3.6.8**). Z uwagi na wspomniany wcześniej koniec oficjalnego wsparcia dla Pythona 2, który ma nadejść wraz z początkiem 2020 roku, naturalnym jest więc wybór wersji 3. Infrastruktura projektu posiada jednak znaczące braki jeśli chodzi o dostępne dla wersji 3 biblioteki zewnętrzne - domyślnie nie jest np. dostępna biblioteka *Beautiful Soup*, służąca do przetwarzania dokumentów w formacie HTML. Niektóre popularne biblioteki i frameworki (np. *PyTest* - wykorzystywany do przeprowadzania testów oprogramowania) nie są dostępne dla obu wersji Pythona.

5.3. Wersja narzędzia budującego CMake

Dostępna wersja narzędzia *CMake* stanowiła zdaniem autorów największe ograniczenie w czasie prac nad projektem. Na maszynach docelowych dostępna jest jedynie stara wersja **2.8.12.2**. Nowsza wersja (**3.14.6**) dostępna jest na niektórych z komputerów, jednak z uwagi na konieczność zachowania kompatybilności ze wspomnianymi maszynami docelowymi, nie było możliwe jej użycie. Stosowanie wersji o numerze niższym od **3.0** skutkuje szeregiem ograniczeń - brakuje w niej wielu funkcjonalności pozwalających na stosowanie ogólnie przyjętych dziś praktyk, jak np. określenie zakresu wersji narzędzia *CMake*, w którym powinna mieścić się używana wersja, by projekt można było bez problemu zbudować, czy wsparcie dla instrukcji *target_link_directories* [32].

5.4. Związek projektu z wersją jądra systemu

Ze względu na mocny związek modułu **ggss-driver** z jądrem systemu operacyjnego ważnym aspektem pracy jest zapewnienie środowiska deweloperskiego zgodnego z produkcyjnym pod względem wersji jądra. Okazało się to problematyczne, ponieważ maszyna dostarczona przez administratorów w celu budowania aplikacji okazała się różnić od maszyny produkcyjnej właśnie pod względem wersji jądra systemu operacyjnego. Z powodu tego, że mocnym ograniczeniem okazało się jądro systemu nie wystarczyło zastosowanie ogólnodostępnych maszyn budujących w ramach wewnętrznego portalu opartego o technologię Gitlab. Ze względu na to autorzy postanowili utworzyć własne środowisko produkcyjne. Rozwiązanie problemu wymagało zastosowania maszyny wirtualnej, z pełnym dostępem do konta administratora oraz skonfigurowania środowiska konteneryzacyjnego Docker właśnie na tej maszynie, co opisane jest w rozdziale 6.6.

6. Wykonane prace

Niniejszy rozdział zawiera opis wykonanych przez autorów prac w ramach projektu **GGSS**. Rozdział został podzielony na sekcję według obszarów tematycznych poruszanych przez autorów: zarządzanie projektem, kontrola wersji i nowa architektura projektu, zastosowanie podejścia CI/CD, budowanie aplikacji, budowanie i dystrybucja sterownika, wirtualizacja i konteneryzacja, pomniejsze prace w projekcie oraz zasady i sposób tworzenia dokumentacji projektu.

6.1. Wykorzystanie funkcjonalności portalu GitLab wspierających zarządzanie projektem



Ze względu na zespołowy charakter pracy bardzo ważną częścią było planowanie oraz zarządzanie projektem. W celu usprawnienia tego procesu autorzy wykorzystali funkcjonalności dostarczane w ramach portalu GitLab oraz dobre praktyki programistyczne.

Pierwszym krokiem było utworzenie grupy, o odpowiedniej nazwie (**atlas-trt-dcs-ggss**) definiującej projekt, na portalu GitLab udostępnianego w ramach infrastruktury CERN. Oprócz oczywistej zalety w postaci identyfikacji przynależności oraz odpowiedzialności za repozytoria zawarte w ramach takiej grupy dodatkowym atutem jest możliwość wykorzystania systemu rang, w celu przyznawania odpowiednich uprawnień. Rysunek 6.1 przedstawia panel zarządzania członkami zespołu, widoczne jest pole odpowiedzialne za dodawanie zarejestrowanych w systemie osób do grupy. W trakcie dodawania możemy również wybrać jeden z zdefiniowanych zestawów uprawnień w postaci rangi (Guest, Reporter, Developer, Maintainer, Owner) oraz datę automatycznego wygaśnięcia członkostwa. Pełny opis uprawnień powiązanych z rangą dostępny jest w oficjalnej dokumentacji portalu GitLab [33]. Poniżej formularza odpowiedzialnego za dodawanie nowych osób do grupy widoczny jest również spis obecnych członków, czas który upłynął od ich dołączenia, ranga, czy też pola oraz przyciski odpowiedzialne za modyfikację ich obecnego statusu w zespole.

Add new member to **atlas-trt-dcs-ggss**

<input type="text" value="Search for a user"/>	<div>Guest</div>	<input type="text" value="Expiration date"/>	<input type="button" value="Add to group"/>
Search for members by name, username, or email, or invite new ones using their email address.	Read more about role permissions	On this date, the member(s) will automatically lose access to this group and all of its projects.	

Existing **2**

Members with access to atlas-trt-dcs-ggss		<input type="text" value="Search"/>	<input type="button" value="2FA"/>	<div>Everyone</div>	Sort by	<div>Oldest joined</div>
	user1 @user1 Given access 1 day ago				<div>Owner</div>	<input type="button" value="Leave"/>
	user2 @user2 Given access 1 day ago		<div>Member</div>	<input type="text" value="Expiration date"/>		<input type="button" value="Remove"/>

Rys. 6.1. Panel zarządzania członkami zespołu na portalu GitLab

Podstawowym problemem do rozwiązania w trakcie pracy grupowej jest odpowiedni podział zadań ze względu na ilość oraz trudność. Wykorzystane w tym celu zostały **issues** (problemy, zagadnienia). Są one podstawową funkcjonalnością wykorzystywaną do współpracy, planowania, czy też przedstawiania swoich pomysłów lub problemów.

W ramach **issue** jesteśmy w stanie precyzyjnie opisać problem, czy też zadanie. Zawdzięczamy to mnogością możliwych do wypełnienia pól, dzięki czemu zarządzanie projektem jest znacznie ułatwione. Na Rysunku 6.2 widnieje panel w ramach którego jesteśmy w stanie zdefiniować:

- **Tytuł** (Title) - krótki opis słowny zawartości
- **Opis** (Description) - pełny opis problemu, zadania, czy też pomysłu, który jest poruszany w ramach **issue**
- **Osoba odpowiedzialna** (Assignee) - główna osoba koordynująca lub wykonująca zadanie
- **Termin** (Due date) - data do której należy zamknąć **issue**, czyli wykonać wszystkie, wymagane w ramach zadania, akcje
- **Kamień milowy** (Milestone) - pole wynikające z zastosowania dobrych praktyk programistycznych. Duże cele powinny być podzielone na mniejsze zadania, które można wykonać w stosunkowo krótkim czasie, dzięki czemu możliwość zmierzenia stopnia wykonania zadania jest większa. Zadania realizowane w ramach jednego większego celu są powiązane za pomocą kamienia milowego, który powinien zawierać informacje nt. tegoż celu.
- **Etykieta** (Label) - służy przypisaniu do zadania atrybutu. W przypadku pracy wykonanej przez autorów etykiety mają na celu określanie stanu realizacji zadania (Ongoing, To

Do), powodu niewykonania zadania (Blocked, Duplicate) oraz oznaczenie zadań z wysokim priorytetem (Urgent). Etykiety nie mają żadnych ograniczeń co do nazwy, zatem możemy za ich pomocą przypisywać dowolne atrybuty zadaniom. Pozwalają one również na łatwe wyszukiwanie zadań do nich przypisanych, ze względu na możliwość filtrowania według etykiet. Najważniejszą oraz najbardziej przydatną funkcją etykiet jest możliwość utworzenia tzw. **Kanban Board** (tablica Kanban).

- **Waga** (Weight) - jest to wartość definiująca trudność zadania oraz wymaganą ilość pracy na jego wykonanie. Wartość ta nie jest bezpośrednio przekładana na czas, wymaga ona interpretacji pod względem przypisanej osoby. Dokładniejsze wytłumaczenie zostanie ukazane na przykładzie.

Poniższy przykład pokazuje wykorzystanie wagi **issue** w celu oszacowania czasu potrzebnego na jego wykonanie.

Zakładając, że do dyspozycji jest dwóch pracowników, jeden doświadczony, który w ciągu tygodnia jest w stanie wykonać pracę o równowartości wagowej 40, a drugi niedoświadczony, który w ciągu tygodnia wykonuje pracę o równowartości wagowej 20. Aby przełożyć wagę zadania, która wynosi w przykładzie 8, należy wziąć pod uwagę osobę, która zostanie do tego zadania przydzielona. Zakładając, że ww. pracownicy pracują na pełen etat, czyli 40 godzin tygodniowo, w przypadku przydzielenia zadania osobie doświadczonej waga 8 przełoży się na 8 rzeczywistych godzin, natomiast w przypadku osoby niedoświadczonej waga 8 przełoży się na 16 rzeczywistych godzin.

Określenie wydajności pracownika wymaga zastosowania techniki nazywanej sprzężeniem zwrotnym. W pierwszej kolejności zakładamy jakąś arbitralną wartość wydajności pracownika i na podstawie tej wartości przydzielamy mu odpowiednią ilość zadań z odpowiednimi wagami. Następnie cyklicznie weryfikujemy (np. co tydzień), czy pracownik jest w stanie wykonać zadania o określonej wadze w danym cyklu. Po każdym cyklu odpowiednio modyfikujemy wartość wydajności pracownika, tak, aby była zbliżona do wartości wydajności osiągniętej za poprzedni okres. Aby zwiększyć poprawność wyznaczania wspomnianej wartości należy wziąć pod uwagę również wcześniejsze okresy, a nie jedynie ostatni.

New Issue

Title

Add [description templates](#) to help your contributors communicate effectively!

Description

Write

Preview

B *I* ” </> 🔗 ☰ ☷ ✍️ 📄 ↗️

Write a comment or drag your files here...

Markdown and [quick actions](#) are supported [Attach a file](#)

☐ This issue is confidential and should only be visible to team members with at least Reporter access.

Assignee

[Assign to me](#)

Milestone

Labels

Weight

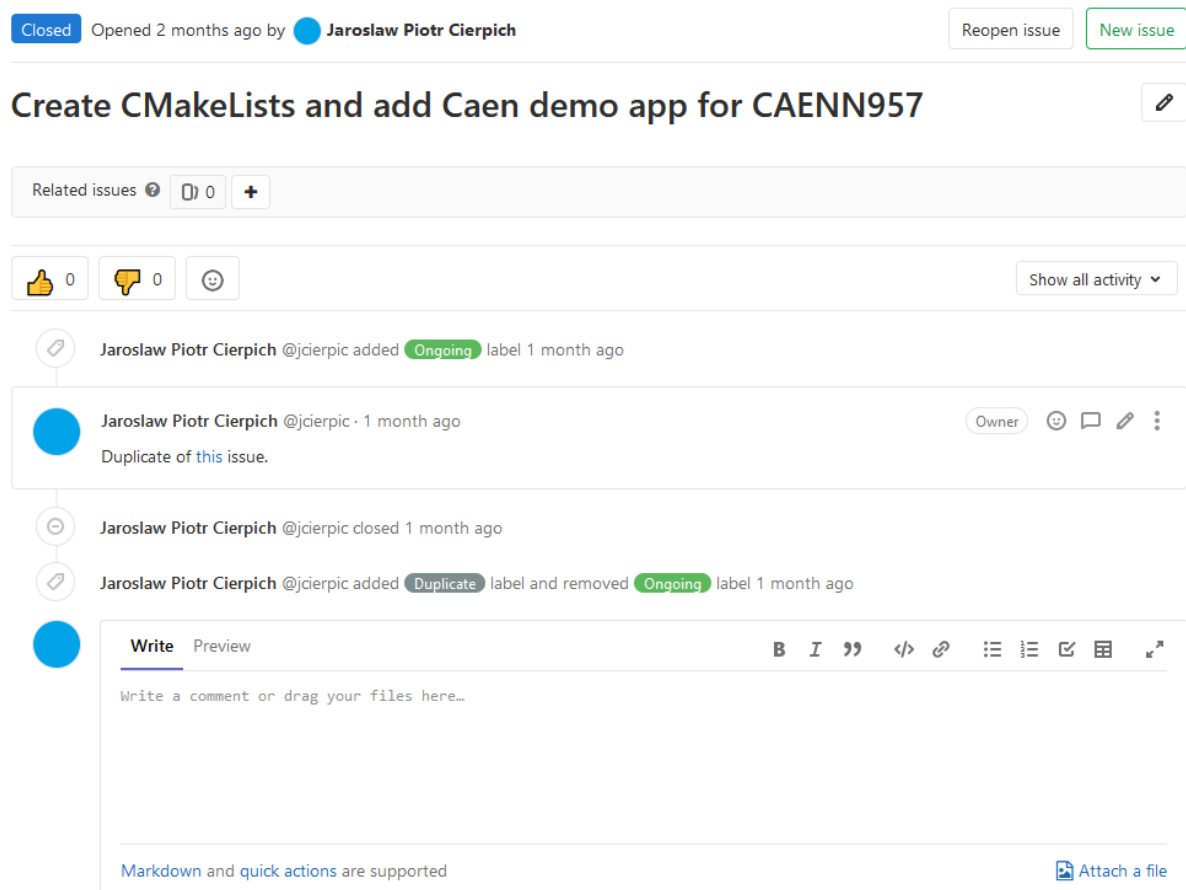
Due date

Rys. 6.2. Formularz tworzenia **issue** na portalu GitLab

W celu usprawnienia tworzenia **issue** możliwe jest zdefiniowanie szablonów w języku **Markdown** i umieszczenie ich w odpowiednim repozytorium, natomiast funkcjonalność ta nie została wykorzystana w trakcie realizacji projektu

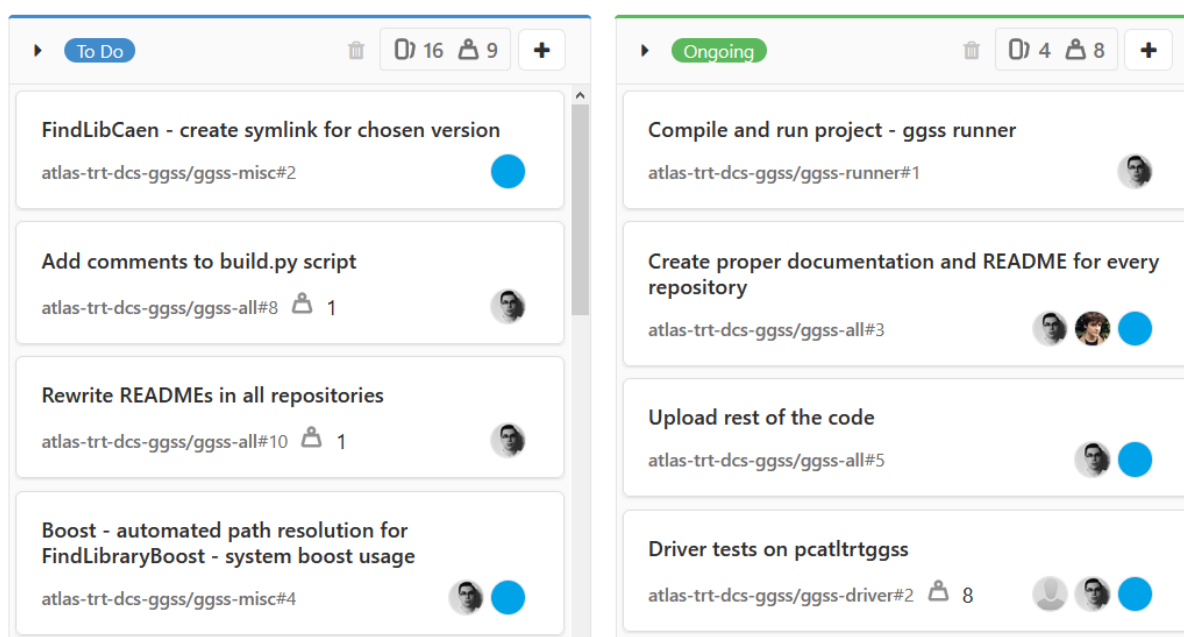
Wszystkie prace wykonywane w ramach projektu były rejestrowane na portalu GitLab w ramach funkcjonalności **issues**. Posiadały one odpowiednio przypisane etykiety. Część zadań miała również przypisane wagi, na podstawie których były one odpowiednio przydzielane członkom zespołu. Zadania realizujące większy cel były grupowane w ramach kamieni milowych.

Ważną kwestią dotyczącą **issues** jest również możliwość śledzenia zmian zachodzących w ramach nich, która była mocno wykorzystywana w przypadku problematycznych zadań. Każde **issue** zapisuje historię zmian dotyczących jego parametrów, jak np. zmiana etykiety. W ramach zadania można również dodawać komentarze opisujące aktualny stan. **Issue** na portalu GitLab są również zintegrowane z wiadomościami definiowanymi w ramach tworzenia nowej rewizji z użyciem technologii **Git**, dzięki czemu tworząc nową wersję jesteśmy w stanie automatycznie zamknąć zadanie, lub dodać referencję do niego co skutkuje odpowiednim wpisem w historii zadania.



Rys. 6.3. Historia zmian **issue** na portalu GitLab





Kolejną ważną funkcjonalnością portalu GitLab wykorzystywaną w ramach projektu był **Kanban Board**. Za jego pomocą można w szybki oraz prosty sposób ocenić etap postępów w projekcie. W jednym widoku wyświetlane jest podsumowanie stanów zadań pogrupowanych według etykiet. Z poziomu widoku tabli możliwa jest również zmiana status zadania za pomocą prostej techniki **drag and drop** (przeciągnij i upuść). Każda wyświetlana etykieta posiada krótkie podsumowanie zadań, czyli ich ilość oraz sumę wag przypisanych do tych zadań. Dodatkowo dla każdego zadania wyświetlana jest nazwa repozytorium, w ramach którego zadanie jest realizowane oraz osoby do niego przypisane, co widoczne jest na Rysunku 6.4



Rys. 6.4. Tablica Kanban dla grupy na portalu GitLab

Portal GitLab zapewnia również funkcjonalność, która wspiera wykonywanie operacji **merge** w ramach technologii **Git**. Udostępniony jest specjalny widok, który pozwala na połączenie dwóch dowolnych gałęzi, które zostały wcześniej przekazane na platformę. Po wybraniu i zatwierdzeniu docelowych gałęzi zostaje utworzony **merge request**, którego zadaniem jest udostępnienie interfejsu zarządzania operacją **merge** oraz, podobnie jak w przypadku **issue**, śledzenie zmian dokonywanych w trakcie trwania **merge request**. W celu wykonania złączenia gałęzi należy taką operację zatwierdzić w panelu **merge request** oraz wykonać ją za pomocą specjalnego przycisku **merge**. Dopiero wtedy zostanie wykonane faktyczne złączenie.

New Merge Request

Source branch	Target branch
<div>atlas-trt-dcs-ggss/ggss-driver</div> <div>gitlab-ci-tests</div>	<div>atlas-trt-dcs-ggss/ggss-driver</div> <div>master</div>
<div> clean gitlabci Jarosław Piotr Cierpich authored 2 weeks ago</div> <div>7cec326a</div> <div></div>	<div> Update submodules, ggss-driver release 10 Jarosław Piotr Cierpich authored 1 week ago</div> <div>c0479c04</div> <div></div>
<div>Compare branches and continue</div>	

Rys. 6.5. Tworzenie nowego merge request

Rysunek 6.6 przedstawia panel tworzenia nowego **merge request**. Możemy w nim zdefiniować takie parametry, jak:

- Tytuł (*Title*)
- Opis (*Description*)
- Osoba przypisana (*Assignee*), czyli osoba, której zadaniem jest wykonanie operacji **merge** oraz rozwiązanie ewentualnych konfliktów
- Kamień milowy (*Milestone*)
- Etykiety (*Labels*)
- Zasady zatwierdzenia (*Approval rules*) - tutaj definiuje się wymagania odnośnie zatwierdzenia przez członków zespołu poprawności wykonanych zmian. Jest to jedna z praktyk dobrego programowania (*code review*).

Title

Gitlab ci tests







Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.

Add [description templates](#) to help your contributors communicate effectively!

Description


Write

Preview

B *I* ” </>      

Write a comment or drag your files here...

Markdown and [quick actions](#) are supported

 [Attach a file](#)

Assignee

Unassigned

Assign to me

Milestone

Milestone

Labels

Labels


Approval rules

Approvers

No. approvals required

All members with Developer role or higher and code owners (if any)

0



Edit

Tip: add a [CODEOWNERS](#) to automatically add approvers based on file paths and file types.

Rys. 6.6. Panel tworzonego merge request

6.2. Migracja projektu do systemu kontroli wersji Git i zmiany w architekturze

Głównymi założeniami prac wykonanych przez autorów było: umieszczenie całego projektu w ramach systemu kontroli wersji **Git** oraz zmiana architektury projektu na bardziej przyjazną dla użytkownika.

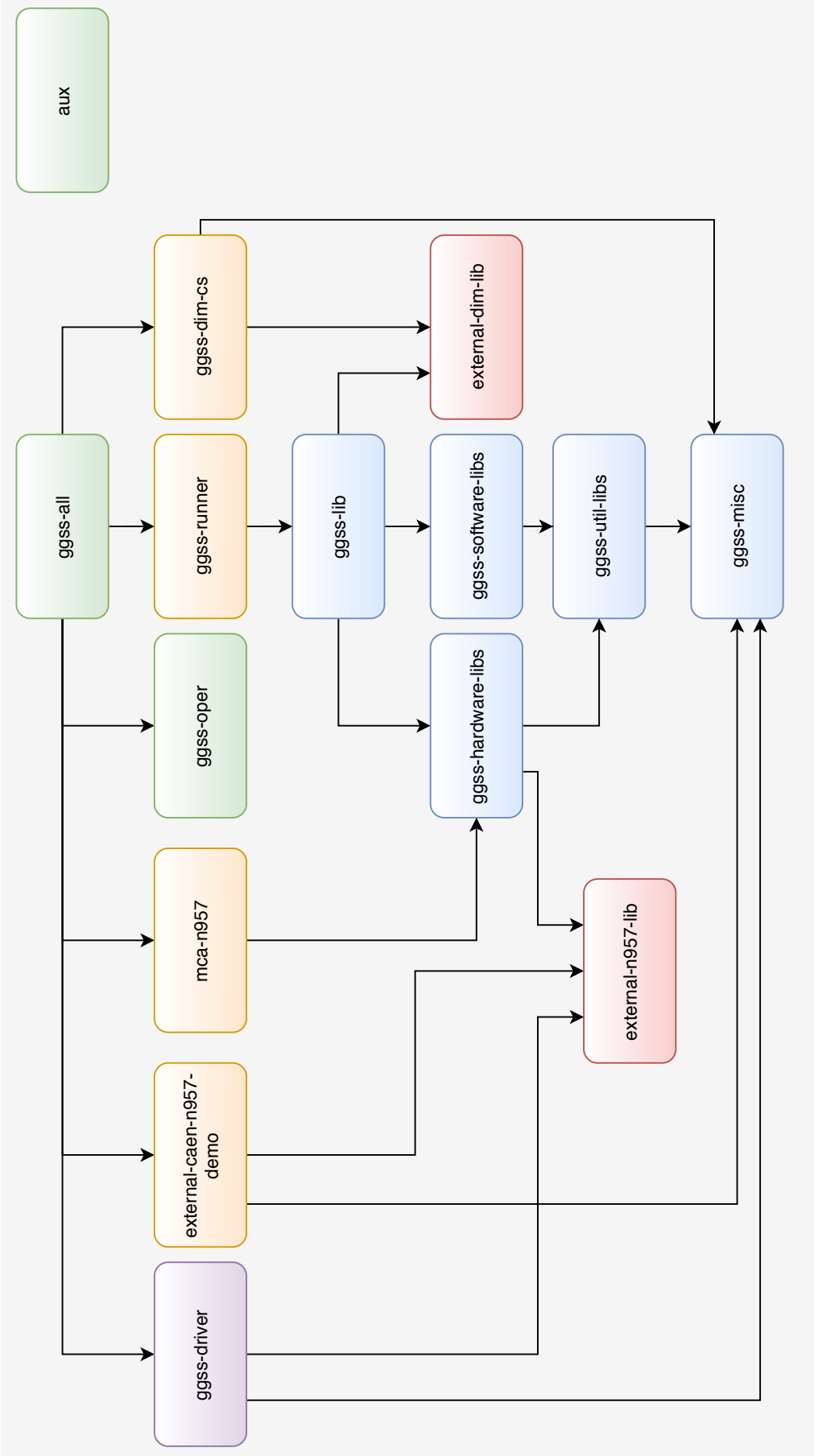
W celu uzyskania jak najbardziej przejrzystej architektury autorzy nałożyli na komponenty projektu następujące wymagania:

- każdy komponent projektu powinien być móc zbudowany w jego minimalnej wersji, czyli proces budowania powinien budować komponent sam w sobie oraz jedynie wymagane zależności
- struktura komponentów powinna zostać zhierarchizowana, czyli każdy komponent powinien zawierać strukturę katalogów oddających jego zależności

Pierwszym krokiem w celu wykonania ww. celów było zdefiniowanie zależności dla każdego komponentu z osobna. Na podstawie zależności została utworzona struktura drzewiasta, co ukazuje Rysunek 6.7. Kolorem czerwonym zostały oznaczone repozytoria zawierające biblioteki zewnętrzne, kolorem niebieskim biblioteki utworzone w ramach projektu **ggss**, kolorem żółtym aplikacje, kolorem fioletowym pakiety **RPM**, a kolorem zielonym repozytoria pomocnicze.

Początkowo planowane było utworzenie osobnego repozytorium dla każdej biblioteki w projekcie **ggss**, natomiast ze względu na dużą ilość bibliotek wewnętrznych zostały one pogrupowane według następujących kryteriów:

- **ggss-util-libs** - repozytorium zawiera wszystkie biblioteki które są jednocześnie wykorzystywane przez komponenty zawarte w ramach repozytoriów *ggss-hardware-libs* i *ggss-software-libs*
- **ggss-software-libs** - biblioteki zawierające logikę powiązaną jedynie z działaniem aplikacji
- **ggss-hardware-libs** - biblioteki posiadające powiązanie z fizycznym sprzętem, np.: wykorzystujące API do jego obsługi



Rys. 6.7. Nowa architektura projektu

Dla nazewnictwa repozytoriów przyjęto następującą konwencję: słowa pisane małymi literami rozdzielone znakiem pauzy. Repozytoria zawarte w ramach projektu oraz ich odpowiedniki z początkowej architektury przedstawione w nawiasach, to (kolorem pomarańczowym oznaczono nowe komponenty w architekturze):

- **ggss-misc** - zawiera szablony programu *CMake* oraz pliki nagłówkowe szeroko używane w projekcie
- **ggss-util-libs** - w ramach repozytorium zawarte zostały następujące biblioteki:
 - **handle-lib** (handleLib)
 - **log-lib** (logLib)
 - **thread-lib** (ThreadLib)
 - **utils-lib** (utilsLib)
- **ggss-software-libs** - w ramach repozytorium zawarte zostały następujące biblioteki:
 - **daemon-lib** (daemonLib)
 - **fifo-lib** (fifoLib)
 - **fit-lib** (FitLib)
 - **xml-lib** (xmlLib)
- **ggss-hardware-libs** - w ramach repozytorium zawarte zostały następujące biblioteki:
 - **caenhv-lib** (CaenHVLib)
 - **caenn1470-lib** (CaenN1470Lib)
 - **mca-lib** (mcaLib)
 - **ortecmcb-lib** (OrtecMcbLib)
 - **usbrm-lib** (usbrmLib)
- **ggss-lib** (ggssLib)
- **ggss-runner** (__ggss)
- **ggss-dim-cs** (__dimCs)
- **external-dim-lib** - repozytorium odpowiedzialne za przechowywanie archiwum biblioteki **DIM**, dodatkowo zawiera skrypty obsługujące proces jej budowania
- **external-n957-lib** - repozytorium zawiera pliki biblioteki dostarczanej przez firmę **CAEN** w celu obsługi analizatora wielokanałowego N957

- **ggss-oper** - zawiera skrypty pomocnicze projektu **GGSS** oraz skrypt odpowiednio je rozmieszczający w środowisku docelowym
- **mca-n957** (caen_n957)
- **external-caen-n957-demo** - repozytorium zawierające aplikację dostarczaną przez firmę **CAEN** w celu testowania poprawności działania analizatora wielokanałowego
- **ggss-driver** (ggss-driver)
- **ggss-all** - repozytorium zawierające wszystkie aplikacje zawarte w ramach projektu **GGSS** oraz skrypty obsługujące budowę całości
- **aux** - repozytorium zawierające elementy związane z projektem **GGSS**, ale nie związane z kodem źródłowym, np.: szablon *.gitlab-ci.yml*, szablon *Dockerfile*.

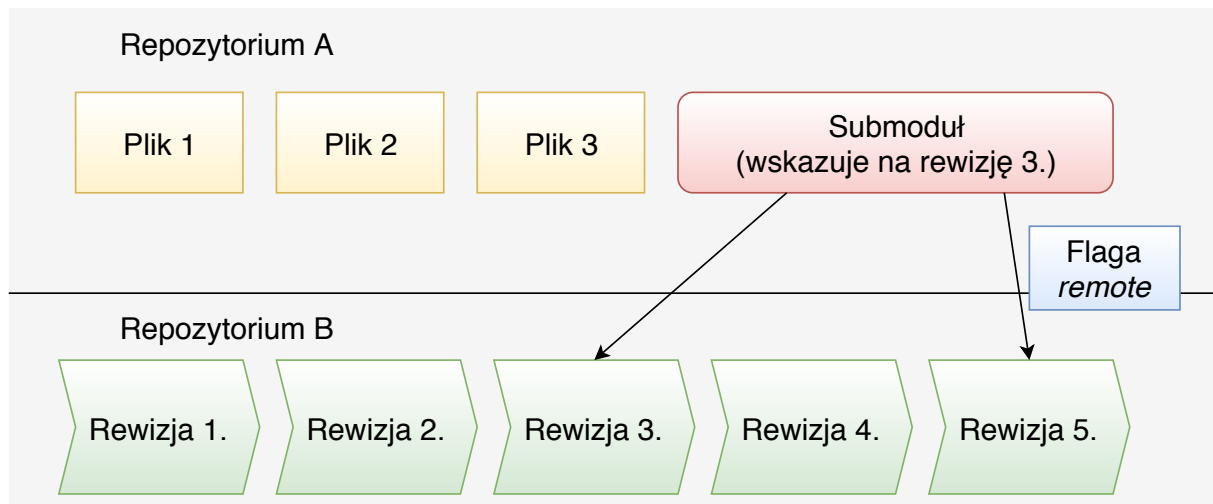
W celu utworzenia wyżej opisanej struktury zależności została wykorzystana funkcjonalność **submodułów** będąca częścią technologii **Git**. Jest ona narzędziem służącym tworzeniu kompozycji repozytoriów w dużych projektach. Sposób działania tej funkcjonalności jest podobny do działania dowiązania symbolicznego. **Submoduł** tak na prawdę jest dowiązaniem wskazującym na konkretną wersję zewnętrznego repozytorium, od którego zależy projekt. Ze względu na precyzję jaką oferuje to rozwiązanie wymagane było przeanalizowanie możliwości wykorzystania w projekcie. Plusem takiego rozwiązania jest możliwość kontrolowania wersji zewnętrznych zależności, co uodparnia nas na zmianę API. Minusem natomiast jest konieczność ciągłego monitorowania zmian i aktualizowania dowiązań.

Pierwszym podejściem jakie autorzy chcieli zastosować jest wykorzystanie funkcjonalności flagi *remote*, która pomija sprawdzanie wersji zależności przypisanej za pomocą **submodułu**, a zamiast tego pobierana jest zawsze najnowsza wersja. Pozwalałoby to na porzucenie konieczności czasochłonnego aktualizowania dowiązań.

Drugim, klasycznym podejściem jest pozostawanie przy wersjach „dowiązanych”. Wymaga to dodatkowego nakładu pracy w utrzymaniu aktualnych zależności, natomiast pozwala na „zamrożenie” stabilnej wersji produktu.

Ze względu na potrzebę ciągłej sprawności systemu zostało wybrane drugie, stabilne rozwiązanie. Pozwala ono na zachowanie informacji o wersjach zależności, z którymi komponent działał poprawnie. Oba podejścia zostały w prosty sposób ukazane na Rysunku 6.8.

Ze względu na wielokrotne zagnieżdżenia **submodułów** w projekcie zastosowana została również flaga *recursive* pozwalająca na pobranie pełnego drzewa zależności na raz. Komendą powszechnie stosowaną w projekcie w celu inicjalizacji oraz aktualizacji **submodułów** jest *git submodule update --init --recursive*.



Rys. 6.8. Sposób działania **submodułów**: klasyczny oraz z flagą *remote*

6.3. Zmiana sposobu budowania aplikacji

Zmiany wprowadzone w systemie budowania projektu GGSS stanowią dużą część wykonanych przez autorów prac. Dotyczą one zarówno przystosowania projektu do nowej architektury, jak i wprowadzenia mechanizmów ułatwiających przeprowadzenie procesu budowania (np. automatyczne pobieranie zależności zewnętrznych). Niniejsza część pracy podzielona została tematycznie na kilka segmentów dotyczących: koncepcji działania nowego systemu, jego wysokopoziomowej implementacji za pomocą narzędzia *CMake*, rozwiązywania zależności zewnętrznych oraz implementacji systemu na poziomie pojedynczych komponentów. Nie został tutaj opisany sposób budowania modułu *ggss-driver* - informacje na jego temat znaleźć można w sekcji 6.5.

6.3.1. Zarys rozwiązania

Wykonane przez autorów rozwiązanie oparte jest, podobnie jak oryginalny system budowania, na narzędziu **CMake** w wersji **2.8**. Główne założenie wprowadzonych zmian to umożliwienie łatwego zbudowania zarówno każdego z dostępnych komponentów z osobna, jak i projektu jako całość. Każdy moduł projektu posiada więc niezależny od modułów poziomu wyższego plik *CMakeLists.txt* pozwalający na zbudowanie go bez konieczności kompilacji całego systemu. Przygotowany system pozwala na łatwe dodawanie nowych modułów do projektu. Aby to umożliwić przygotowane zostały specjalne szablony (pliki *.cmake*), odpowiedzialne m.in. za tworzenie bibliotek statycznych czy znajdowanie zewnętrznych zależności. Projekt wspiera tzw. **budowanie out-of-source** - czyli poza katalogiem zawierającym kod źródłowy projektu. Umożliwia to utrzymanie porządku w strukturze repozytoriów (proces budowania nie generuje w nich żadnych nowych, potencjalnie niepożądanych plików). Dodatkowo jeśli w projekcie dana zależność pojawia się wielokrotnie, przygotowany system zbuduje ją tylko raz, co skraca znacznie czas kompilacji oraz sprawia, że wynikowa struktura katalogów jest czytelna i łatwo się po niej poruszać (przedstawia ją listing 6.1).

Listing 6.1. Fragment struktury katalogów wygenerowanej przez nowy system budujący projekt (użycie dla repozytorium *ggss-runner*). Dla uproszczenia pominięta została zawartość katalogów *build*. Przykład ilustruje płaską strukturę wynikową ułatwiającą szybkie znalezienie odpowiedniej biblioteki lub pliku wykonywalnego (tutaj *ggss-runner*)

```
.
|-- caenhv-lib
|   |-- build
|   '-- lib
|       '-- libcaenhv.a
|-- caenn1470-lib
|   |-- build
|   '-- lib
|       '-- libcaenn1470.a
```



```
|-- daemon-lib
|   |-- build
|   '-- lib
|       '-- libdaemon.a
|-- fifo-lib
|   |-- build
|   '-- lib
|       '-- libfifo.a
|-- fit-lib
|   |-- build
|   '-- lib
|       '-- libfit.a
|-- ggss-lib
|   |-- build
|   '-- lib
|       '-- libggss.a
|-- ggss-runner

... Dalsza część struktury
```

6.3.2. Szablony CMake

W ramach prac nad projektem napisanych zostało kilka plików *.cmake*, stanowiących szablony ułatwiające dodawanie nowych komponentów do aplikacji. Zostały one umieszczone w repozytorium *ggss-misc*. Lista przygotowanych plików wraz z ich zastosowaniem:

- **FindLibraryGSL.cmake** - plik ułatwiający dołączanie biblioteki *GSL* do projektu.
- **FindLibraryCaen.cmake** - plik ułatwiający dołączanie biblioteki *CAENN957* do projektu.
- **FindLibraryBoost.cmake** - plik ułatwiający dołączanie biblioteki *Boost* do projektu. Pozwala on użytkownikowi na dostosowanie sposobu wykonania tej operacji za pomocą flag. Możliwy jest więc wybór między linkowaniem statycznym i dynamicznym oraz wybór lokalizacji, w której biblioteka powinna zostać znaleziona (umożliwia to zastosowanie wersji biblioteki innej niż domyślnie zainstalowana).
- **CheckPlatform.cmake** - wykonanie sprawdzenia platformy, na której budowany jest projekt.
- **BuildLibrary.cmake** - budowanie biblioteki statycznej będącej częścią projektu (listing 6.2).
- **BuildDependencies.cmake** - obsługa zależności aktualnie budowanej aplikacji lub biblioteki (listing 6.3).

Omówione zostaną dwa ostatnie z wymienionych powyżej szablonów. Korzystanie z nich polega na dołączeniu ich do pliku *CMakeLists.txt* za pomocą polecenia *include*, po wcześniejszym zdefiniowaniu wymaganych przez nie zmiennych (np. *target_name*). Instrukcja użycia tych plików zawarta została w pliku *README.md* w repozytorium *ggss-misc*.

Plik **BuildLibrary.cmake** wykorzystywany jest w projekcie GGSS do budowania bibliotek statycznych (pliki z rozszerzeniem *.a*). Listing 6.2 przedstawia zawartość tego pliku. W pierwszej jego części definiowana jest lokalizacja, w której przeprowadzony zostanie proces budowania (zmienna *BUILD_OUTPUT_DIRECTORY*). To właśnie ten fragment umożliwia budowanie *out-of-source*. Następnie określany jest katalog w którym umieszczona zostanie zbudowana biblioteka (*CMAKE_ARCHIVE_OUTPUT_DIRECTORY*). Komenda *file* odpowiada za wyszukiwanie wszystkich plików źródłowych i nagłówkowych będących częścią biblioteki. Samo archiwum zostaje dodane do projektu za pomocą polecenia *add_library*. Ostatnia linia pliku odpowiada za dodanie stworzonej biblioteki do listy zależności aktualnie tworzonego projektu - dzięki temu biblioteka ta zostanie poprawnie dołączona do potencjalnego projektu nadrzędnego (np. aplikacji).

Listing 6.2. Plik *BuildLibrary.cmake* będący częścią systemu budującego projekt GGSS. Poszczególne linie odpowiadają m.in. za dodanie nowego projektu, określenie katalogu, w którym odbędzie się budowanie i katalogu docelowego, zebranie plików biblioteki oraz dodanie jej do projektu.

```
project(${target_name})

if(NOT DEFINED BUILD_OUTPUT_DIRECTORY)
    set(BUILD_OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}")
endif()

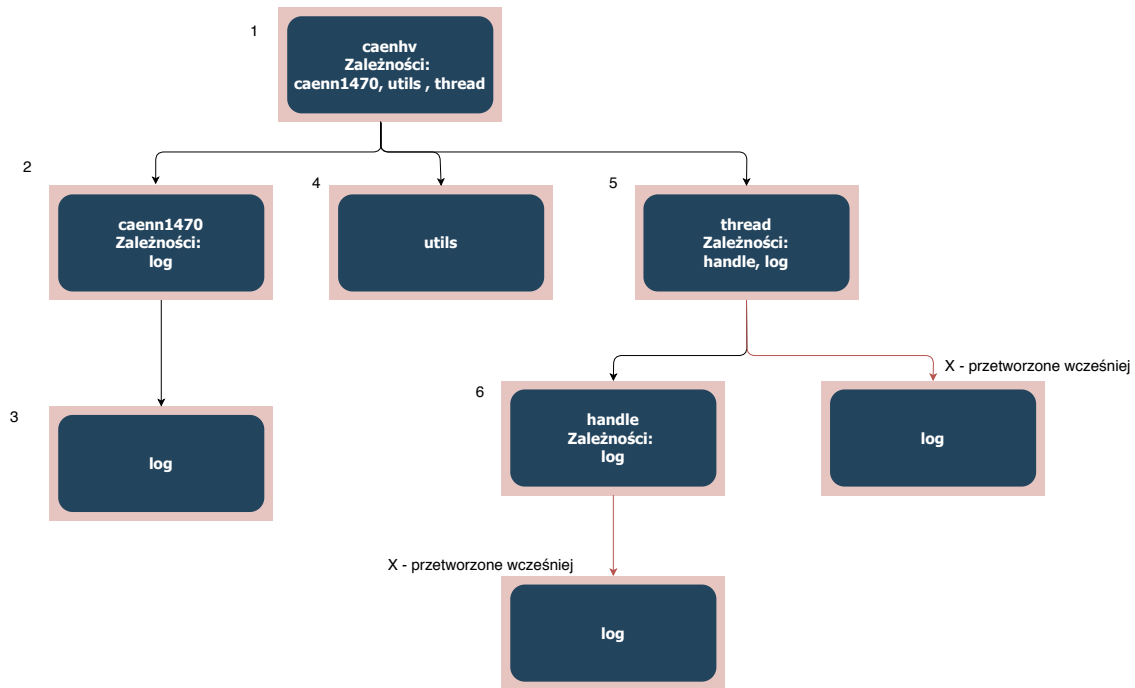
set(DIR_NAME "${target_name}-lib")
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${BUILD_OUTPUT_DIRECTORY}/${DIR_NAME}/lib")

file(GLOB SOURCE_FILES "${CMAKE_CURRENT_SOURCE_DIR}/../${DIR_NAME}/src/*.cpp"
    "${CMAKE_CURRENT_SOURCE_DIR}/../${DIR_NAME}/src/*.cxx"
    "${CMAKE_CURRENT_SOURCE_DIR}/../${DIR_NAME}/src/*.c")

if(UNIX)
    set(CMAKE_CXX_FLAGS "-std=c++11 -Wall -Wno-reorder")
endif()

add_library(${target_name} STATIC ${SOURCE_FILES})
set_target_properties(${target_name} PROPERTIES LINKER_LANGUAGE CXX)
target_include_directories(${target_name} PUBLIC ↵
    "${CMAKE_CURRENT_SOURCE_DIR}/include")

list(APPEND dependency_list ${target_name})
```



Rys. 6.9. Działanie przygotowanego systemu budującego (przykład dla biblioteki *caenhv*). Przetwarzanie zależności w projekcie przypomina przetwarzanie struktury drzewa. Dany moduł dołącza do procesu budowania swoje pod-moduły, które następnie powtarzają tę czynność itd. Widoczna numeracja obrazuje przykładową kolejność przetwarzania zależności przez narzędzie *CMake* (nie należy jej mylić z kolejnością budowania). Widoczny jest również fakt nieprzetwarzania ponownie raz dołączonych do projektu modułów.

Plik **BuildDependencies.cmake** (listing 6.3) odpowiada za przeprowadzenie procesu budowania komponentów, od których jest zależny aktualnie przetwarzany projekt. Jego działanie opiera się na komendzie *add_subdirectory*, która pozwala na przetworzenie pliku *CMakeLists.txt* budującego daną zależność. Działanie to ilustruje rysunek 6.9. Dzięki takiemu podejściu budowanie projektu przypomina proces przetwarzania drzewa - moduł nadrzędny dołącza do procesu budowania pliki *CMakeLists.txt* swoich zależności, które następnie wykonują są samą czynność dla swoich zależności itd. Lista modułów podrzędnych względem aktualnie budowanego przechowywana jest w zmiennej *dependencies*, po której iteruje pętla *foreach*.

Listing 6.3. Plik *BuildDependencies.cmake* będący częścią systemu budującego projekt GGSS. Zawiera pętlę iterującą po zależnościach aktualnie budowanego modułu. Zależności te dodawane są do systemu budującego za pomocą polecenia *add_subdirectory*. Dołączone do listingu komentarze stanowią wytłumaczenie działania poszczególnych elementów pliku.

```
foreach(dependency ${dependencies})

    # pobranie nazwy zależności bez poprzedzającej ją ścieżki
    get_filename_component(dependency_name_without_path ${dependency} NAME)
```

```
# sprawdzenie, czy zależność nie została dodana do projektu już wcześniej -
# zapobiega wielokrotnemu budowaniu tych samych bibliotek
if (NOT TARGET ${dependency_name_without_path})

    # powoduje dodanie pliku CMake modułu do systemu budującego
    add_subdirectory("${dependency_prefix}/${dependency}-lib" ←
        "${BUILD_OUTPUT_DIRECTORY}/${dependency_name_without_path}-lib/build")

    # pozyskanie zależności modułu podrzędnego
    get_directory_property(child_dependencies
        DIRECTORY "${dependency_prefix}/${dependency}-lib"
        DEFINITION dependency_list
    )

    # dodanie zależności do listy
    foreach(child_dependency ${child_dependencies})
        list(FIND dependency_list ${child_dependency} dependency_index)
        if(NOT ${dependency_index} GREATER -1)
            list(APPEND dependency_list ${child_dependency})
        endif()
    endforeach(child_dependency)

endif()

# linkowanie zależności do modułu nadrzędnego
target_link_libraries(${target_name} ${dependency_name_without_path})

endforeach(dependency ${dependencies})
```

6.3.3. Budowanie poszczególnych bibliotek

Budowanie bibliotek w projekcie GGSS odbywa się za pomocą omówionego wcześniej szablonu *BuildLibrary.cmake*. Na przykładzie jednej z bibliotek wchodzących w skład projektu GGSS ilustruje to listing 6.4. W tym przykładzie zaobserwować można m.in. użycie zmiennej *dependencies* definiującej zależności biblioteki. Zmienna ta jest wykorzystywana przez szablon *BuildDependencies.cmake*. Przykład ilustruje również użycie innych szablonów, takich jak *FindLibraryBoost*. W podobny sposób skonstruowane są wszystkie pliki *CMakeLists.txt* konfigurujące proces budowania bibliotek statycznych będących częścią projektu. Bardzo podobnie wyglądają również pliki budujące aplikacje (np. *ggss-runner*). Z uwagi na to podobieństwo autorzy zdecydowali się nie prezentować ich w niniejszej pracy.

Listing 6.4. Plik *CMakeLists.txt* służący do budowania biblioteki *caenhv*. Widoczne zastosowanie m.in. plików *BuildLibrary.cmake* oraz *BuildDependencies.cmake*.

```
cmake_minimum_required(VERSION 2.8)

set(target_name "caenhv")

if(NOT TARGET ${target_name})

    set(ggss_misc_path "${CMAKE_CURRENT_SOURCE_DIR}/../ggss-util-libs/ggss-misc")
    set(CMAKE_MODULE_PATH "${ggss_misc_path}/cmake_templates")

    include(BuildLibrary)
    include(FindLibraryBoost)

    target_include_directories(${target_name} PUBLIC ↔
        "${CMAKE_CURRENT_SOURCE_DIR}/include/CaenHVLib")
    target_include_directories(${target_name} PUBLIC "${ggss_misc_path}/include")
    target_include_directories(${target_name} PUBLIC "${ggss_misc_path}")

    set(dependency_prefix "${CMAKE_CURRENT_SOURCE_DIR}/..")
    set(dependencies "caenn1470" "ggss-util-libs/utils" "ggss-util-libs/thread")
    include(BuildDependencies)
    include(CheckPlatform)

endif()
```

6.3.4. Budowanie aplikacji i całego projektu

W celu usprawnienia procesu budowania całego projektu, w repozytorium *ggss-all* umieszczony został specjalny plik *CMakeLists.txt* odpowiedzialny za przeprowadzenie budowania w ściśle określonej przez użytkownika konfiguracji. Użytkownik może zmieniać m.in. sposób dołączania do projektu biblioteki Boost (statycznie lub dynamicznie) i wersję budowanego produktu (deweloperska lub produkcyjna).

Narzędzie *CMake* definiuje kilka możliwych wersji produktu. Z perspektywy projektu GGSS najważniejsze z nich to wyżej wspomniana wersja deweloperska (*debug*) oraz produkcyjna (*release*). Tabela 6.1 przedstawia listę flag kompilacji, jaką narzędzie aplikuje w procesie budowania danej wersji. Konfiguracja *release* została jednak przez autorów zmodyfikowana - flaga *-O3* odpowiada za przeprowadzenie bardzo agresywnej optymalizacji, która sprawia że produkt przestaje zachowywać się stabilnie. Została więc zastąpiona flagą *-O2* stanowiącą bezpieczniejszą alternatywę.

W celu uproszczenia (z punktu widzenia użytkownika) procesu budowania produktu w wybranej konfiguracji przygotowany został skrypt *build.py* (fragment widoczny na listingu 6.5) pozwalający na jej łatwe określenie. Skrypt napisany został w języku Python 3 i jego dzia-

Tabela 6.1. Flagi użyte podczas procesu budowania za pomocą narzędzia *CMake* w zależności od wybranej konfiguracji. Tabela omawia tylko konfiguracje stosowane w projekcie (poza nimi dostępne są jeszcze trzy inne). [34]

Konfiguracja (debug/release)	Flagi (gcc)
Debug	-g
Release	-O3 -DNDEBUG

lanie opiera się na module *argparse* służącym do przetwarzania argumentów linii poleceń. Po przetworzeniu podanych przez użytkownika informacji skrypt uruchamia proces przetwarzający wspomniany wyżej plik *CMakeLists.txt* w sposób zgodny z konfiguracją wybraną przez użytkownika.

Listing 6.5. Fragment pliku *build.py* stanowiącego interfejs pozwalający na łatwe budowanie projektu GGSS w różnych konfiguracjach.

```
# Definicje funkcji ...

if __name__ == "__main__":
    arguments = parse_command_line_arguments()
    cmake_command_line_options = prepare_cmake_options(arguments)
    virtualenv_enabled = check_if_venv_should_be_used(arguments)
    build_project_with_options(cmake_command_line_options, virtualenv_enabled)
```

Udostępnione narzędzia pozwalają użytkownikowi wybrać, która z dostępnych w projekcie GGSS aplikacji powinna zostać zbudowana. Aktualnie wspierane są: *ggssrunner*, *ggss-dim-cs* oraz *mca-n957*. Wybór odbywa się za pomocą argumentów przekazanych do skryptu *build.py* i działa na zasadzie wykluczania - użytkownik specyfikuje, które z tych trzech aplikacji mają zostać pominięte w procesie budowania. Listing 6.6 przedstawia wywołanie skryptu z opcją **-h** powodującą wyświetlenie możliwych do użycia w skrypcie flag.

Listing 6.6. Użycie skryptu *build.py* z opcją **-h** powodujące wyświetlenie możliwych do wykorzystania parametrów. Widoczne m.in. parametry *nomcan957*, *nodimcs* oraz *norunner* pozwalające użytkownikowi na wykluczenie niektórych aplikacji wchodzących w skład projektu GGSS z procesu budowania.

```
user@host:~/ggss-all$ python3 build.py -h
usage: build.py [-h] [-d] [-s] [-v] [--debug | --release]
               [--boostvar | --boostdef] [--norunner] [--nodimcs]
               [--nomcan957]

optional arguments:
  -h, --help            show this help message and exit
  -d, --dimonline        build project using latest DIM version downloaded from
                        website. Disabled by default.
```

```
-s, --staticboost  force Boost the use the static libraries. Disabled by
                    default.
-v, --venv         build project using virtualenv. All required dependencies
                    will be installed. Disabled by default.
--debug           enable debug mode
--release         enable release mode
--boostvar        use Boost that can be accessed using BOOST environment
                    variable. Enabled by default.
--boostdef        use default Boost version from /usr/local.
--norunner        exclude ggss-runner from build. Disabled by default.
--nodimcs         exclude dim-cs from build. Disabled by default.
--nomcan957       exclude mca-n957 from build. Disabled by default.
```

6.3.5. Budowanie biblioteki DIM

Projekt GGSS korzysta z pakietu **DIM (Distributed Information Management System)** implementującego komunikację klient-serwer [35]. Ponieważ jest to zależność zewnętrzna i jej rozwój nie jest kontrolowany przez osoby pracujące nad systemem GGSS, jej system budowania działa w odmienny od pozostałej części projektu sposób. Pozwala on na zautomatyzowane pobieranie najnowszej wersji biblioteki.

Repozytorium *external-dim-lib* przechowuje wszystkie pliki konieczne do zbudowania biblioteki DIM. Znajduje się tam również archiwum z samą biblioteką. Poszczególne elementy tego repozytorium to:

- katalog **dim_offline** - zawiera archiwum *.zip* z biblioteką DIM w wersji 20.26.
- plik **CMakeLists.txt** - plik odpowiedzialny za budowanie biblioteki (zarówno w wersji statycznej jak i współdzielonej). Za pomocą tego pliku możliwe jest zbudowanie DIM'a bez konieczności dodawania go jako submodułu do innego projektu.
- plik **dim_build.cmake** - plik, który należy dołączyć do projektu *CMake* (za pomocą komendy *include*) zależnego od biblioteki DIM. Plik ten odpowiedzialny jest za zbudowanie oraz dodanie jej do zależności projektu nadrzędnego.
- plik **dim_download.py** - skrypt napisany w języku Python 3 odpowiedzialny za pobieranie najnowszej wersji biblioteki DIM z jej strony internetowej.
- plik **test_dim_download.py** - testy jednostkowe do skryptu *dim_download.py*.
- pliki **requirements.txt** oraz **venvsetup.sh** - zarządzanie zależnościami wymaganymi przez skrypt pobierający DIM'a.

Proces budowania biblioteki DIM jako części nadrzędnego projektu (czyli za pomocą szablonu *dim_build.cmake*) może przebiegać na jeden z dwóch sposobów: wykorzystana może zostać

jej wersja znajdująca się w katalogu *dim_offline* w repozytorium, lub może zostać podjęta próba pobrania ze strony internetowej i zbudowania jej nowszego wydania. Rys. 6.10 przedstawia dokładniej sposób działania algorytmu. Wybór sposobu budowania podjęty zostaje na podstawie przekazanej do szablonu flagi **ONLINE_DIM**. Jeśli próba użycia nowszej wersji zakończy się niepowodzeniem, zbudowana zostanie ta znajdująca się w repozytorium.

Listing 6.7 przedstawia fragment pliku *dim_build.cmake* odpowiedzialny za podjęcie próby pobrania i zbudowania nowej wersji biblioteki DIM. Poszczególne kroki tego procesu zaimplementowane zostały jako funkcje (listing 6.8 - narzędzie *CMake* pozwala na definicję funkcji posiadających własny zakres zmiennych). Zaprezentowany fragment wykorzystuje komendy **execute_process**, które pozwalają na stworzenie nowego procesu za pomocą interfejsu programistycznego systemu operacyjnego. Za pomocą tych komend tworzony jest katalog w którym przeprowadzony zostanie proces budowania, wykonywany jest skrypt *dim_download.py* pobierający wymaganą bibliotekę, a następnie przeprowadzany jest proces jej budowania (za pomocą pliku *CMakeLists.txt* będącego częścią repozytorium *external-dim-lib* oraz wygenerowanego pliku *Makefile*). Niepowodzenie wykonania dowolnego z etapów powoduje przerwanie procesu i przejście do budowania za pomocą wersji biblioteki dostępnej lokalnie.

Listing 6.7. Fragment pliku *dim_build.cmake* przedstawiający proces budowania nowej, pobranej ze strony internetowej wersji biblioteki DIM. Widoczne wywołania zdefiniowanych przez autorów funkcji.

```
if (USE_ONLINE_DIM)

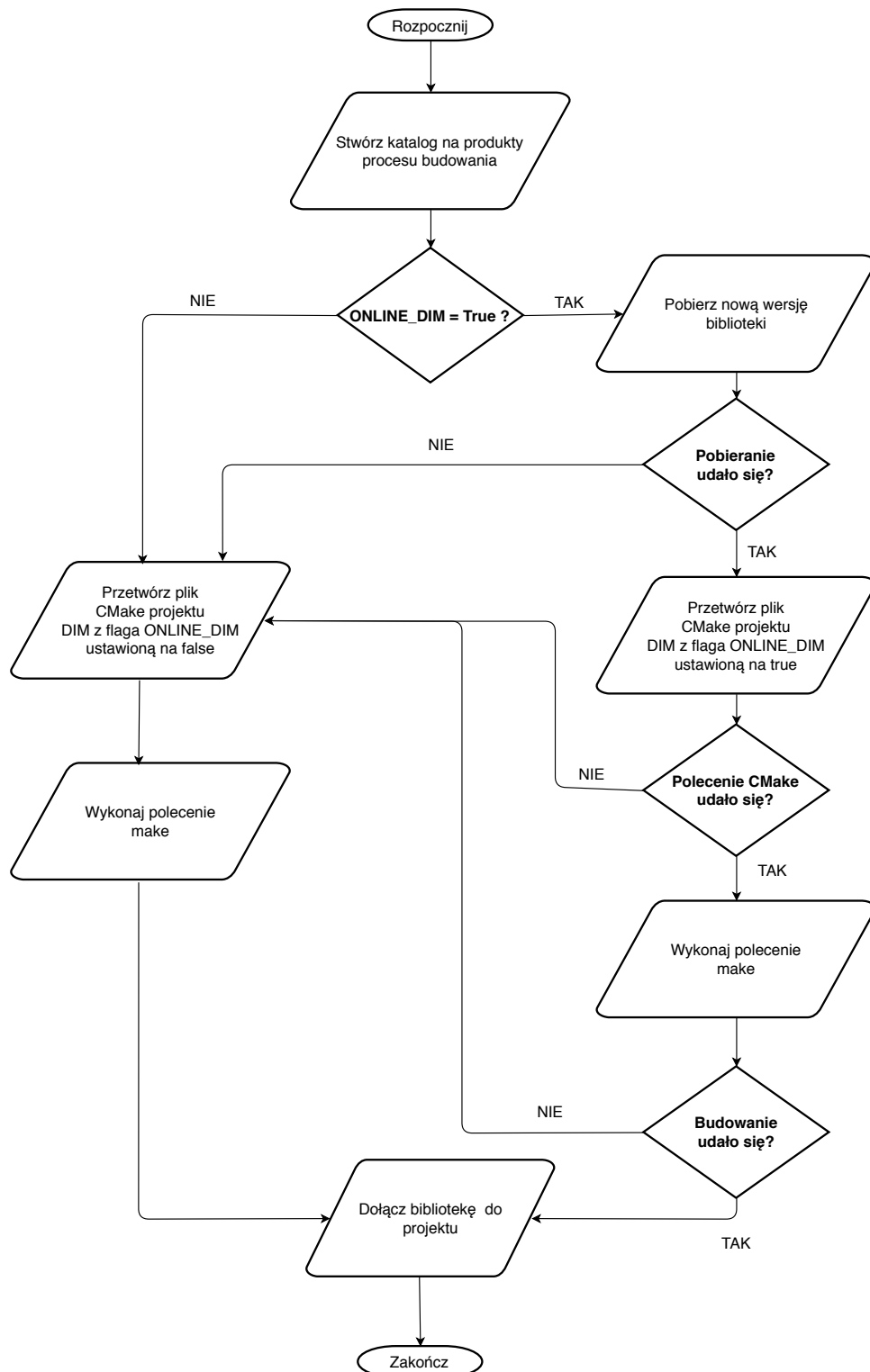
    message (STATUS "Downloading and building online DIM version")

    ## First, try to download DIM from website
    DOWNLOAD_DIM_FROM_WEBSITE ()

    if (NOT result) # if there was no error
        USE_DIM_CMAKE (true)
    endif ()

    if (NOT result)
        message (STATUS "DIM download and build successful.")
    else ()
        message (STATUS "DIM online build problem. Using offline DIM version.")
    endif ()

endif ()
```

Rys. 6.10. Algorytm budowania biblioteki DIM jako części nadrzędnego projektu

Listing 6.8. Przykład funkcji zdefiniowanej w skrypcie *CMake*. Funkcja wykorzystuje komendę *execute_process* do wykonania skryptu pobierającego bibliotekę DIM z jej strony internetowej.

```
function(DOWNLOAD_DIM_FROM_WEBSITE)

    execute_process(
        COMMAND python3 ${CMAKE_CURRENT_SOURCE_DIR}/external-dim-lib/dim_download.py
        RESULT_VARIABLE result
        ERROR_VARIABLE error
    )

    set(result ${result} PARENT_SCOPE)
    set(error ${error} PARENT_SCOPE)

endfunction()
```

Do archiwum zawierającego bibliotekę DIM dołączony jest plik *Makefile* pozwalający na jej zbudowanie. Autorzy zdecydowali się wykorzystać go w przygotowanym przez siebie pliku *CMakeLists.txt*. Rozwiązanie to zapewnia pewien poziom odporności przygotowanej infrastruktury na potencjalne zmiany w sposobie budowania biblioteki. Listing 6.9 przedstawia fragment przygotowanego przez autorów pliku. Wykorzystuje on funkcjonalność **ExternalProject** do budowania biblioteki. Funkcjonalność ta pozwala programiście na zdefiniowane szeregu kroków koniecznych do otrzymania działającego projektu zewnętrznego. Metoda ta została wybrana przez autorów z uwagi na duże możliwości konfiguracji, jakie ona zapewnia. Standardowo proces budowania zewnętrznych zależności za pomocą funkcjonalności *ExternalProject* składa się z kroków takich, jak: pobieranie zależności, budowanie, instalacja czy testowanie [36]. Na potrzeby zastosowania w projekcie GGSS niektóre z tych etapów zostały pominięte.

Listing 6.9. Fragment pliku *CMakeLists.txt* przeznaczonego do konfiguracji procesu budowania biblioteki DIM. Widoczne zastosowanie funkcjonalności *ExternalProject* pozwalającej na wykorzystanie plików *Makefile* (*makefile_dim*) dostarczonych przez autorów biblioteki.

```
ExternalProject_Add(dim_library
    DOWNLOAD_COMMAND
        COMMAND cmake -E echo Extracting dim from archive
        COMMAND unzip -aq ${DIM_ARCHIVE_DIR}/${DIM_ARCHIVE_NAME} -d ↵
            ${CMAKE_CURRENT_SOURCE_DIR}
        COMMAND cmake -E rename ↵
            ${CMAKE_CURRENT_SOURCE_DIR}/${DIM_UNZIP_OUTPUT_DIRECTORY} ↵
            ${CMAKE_CURRENT_SOURCE_DIR}/dim-org
    CONFIGURE_COMMAND ""
    BUILD_COMMAND
        COMMAND cmake -E echo Setting environment variables and building
        COMMAND
            export OS=${OS} &&
            export DIMDIR=${DIMDIR} &&
```

```

export ODIR=${ODIR} &&
echo ${CMAKE_CURRENT_SOURCE_DIR} &&
bash -c "source ${CMAKE_CURRENT_SOURCE_DIR}/dim-org/setup.sh" &&
cd ${CMAKE_CURRENT_SOURCE_DIR}/dim-org &&
make -f ${CMAKE_CURRENT_SOURCE_DIR}/dim-org/makefile_dim
UPDATE_COMMAND ""
INSTALL_COMMAND
    COMMAND cmake -E echo Moving dim to given directory
    COMMAND cmake -E copy ${ODIR}/libdim.a ${LIB_ODIR}/libdim.a
    COMMAND cmake -E copy ${ODIR}/libdim.so ${LIB_ODIR}/libdim.so
    COMMAND cmake -E copy_directory ${CMAKE_CURRENT_SOURCE_DIR}/dim-org/dim ↵
        ${INCLUDE_ODIR}/dim
    COMMAND cmake -E echo Cleaning temporary files
    COMMAND cmake -E remove_directory ${CMAKE_CURRENT_SOURCE_DIR}/dim-org
)

```

Proces pobierania nowej wersji biblioteki DIM przeprowadzany jest za pomocą skryptu *dim_download.py*. Wykorzystuje on m.in. moduł **BeautifulSoup4** do przetwarzania kodu HTML oraz **requests** do pobierania tegoż kodu ze strony internetowej biblioteki. Algorytm działania skryptu jest następujący:

1. Pobranie kodu HTML ze strony biblioteki DIM za pomocą modułu *requests*.
2. Przetworzenie kodu HTML za pomocą modułu *BeautifulSoup4* (listing 6.10). Operacja ta ma na celu uzyskanie adresu URL pliku z biblioteką. Nie jest możliwe podanie tego adresu bezpośrednio w kodzie skryptu, gdyż zawiera on wersję biblioteki.
3. Utworzenie katalogu w którym znajdzie się pobrana biblioteka.
4. Pobranie biblioteki za pomocą modułu *urllib*.

Listing 6.10. Fragment skryptu *dim_download.py* odpowiedzialny za przetworzenie dokumentu HTML. Widoczne użycie elementów modułu *BeautifulSoup4*.

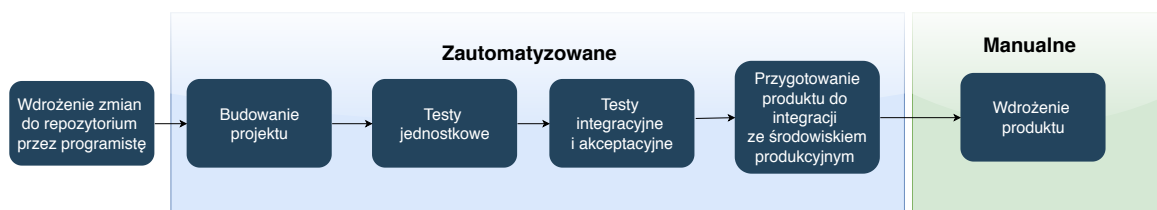
```

def get_dim_archive_url_from_html(dim_website_html):
    dim_website_soup = bs4.BeautifulSoup(dim_website_html, 'html.parser')
    links = dim_website_soup.find_all('a')
    return list(filter(
        lambda link: (any(".zip" in attr for attr in link) and ↵
            link.getText().strip() == "dim.zip"),
        links))[0].attrs['href']

```

6.4. Zastosowanie podejścia CI/CD

Ważną częścią wykonanych prac było przygotowanie środowiska pozwalającego na zautomatyzowane budowanie i dystrybucję aplikacji. Standardowym dziś sposobem na rozwiązanie tego problemu jest podejście **CI/CD**. Skrót CI oznacza tzw. **ciągłą integrację** (*Continuous Integration*) - praktykę polegającą na stosowaniu współdzielonego repozytorium kodu źródłowego, za pomocą którego programiści pracujący nad projektem regularnie integrują swoje zmiany. Nowa wersja kodu jest automatycznie sprawdzana - serwer ciągłej integracji samodzielnie buduje projekt i uruchamia przygotowane dla niego testy. Skrót CD oznacza natomiast **ciągłe dostarczanie** (*Continuous Delivery*). Polega ono na przygotowaniu produktu do stanu, w którym jest on możliwy do wdrożenia w środowisku produkcyjnym. Może to być np. przeprowadzenie różnego rodzaju testów czy przygotowanie odpowiedniej paczki z aplikacją. Ważne jest, że w podejściu tym nie następuje automatyczne wdrożenie aplikacji do środowiska produkcyjnego (jest to domena **ciągłego wdrażania** [37], które jednak nie znalazło zastosowania w projekcie GGSS). Podejście **CI/CD** porównuje się czasem do działania linii produkcyjnej. Rysunek 6.11 obrazuje schematycznie jego działanie.



Rys. 6.11. Przykładowy schemat działania podejścia *Continuous Integration / Continuous Delivery*. Należy zwrócić uwagę, że w zastosowaniu praktycznym kolejność oraz liczba etapów może być różna od widocznej

W kontekście systemu GGSS podejście to jest pożądane, z uwagi na konieczność zachowania poprawności działania systemu pomimo zmian (które aktualnie mają miejsce) w strukturze jego oprogramowania. Głównym celem jego zastosowania było ułatwienie autorom pracy działania jako zespół. Miało również znacząco przyspieszyć proces testowania aplikacji w środowisku produkcyjnym poprzez automatyczne tworzenie paczki z odpowiednią wersją oprogramowania.

6.4.1. Możliwe sposoby implementacji podejścia CI/CD w projekcie GGSS

Istnieje wiele narzędzi pozwalających na implementację ciągłej integracji oraz ciągłego dostarczania w projekcie. Prawdopodobnie najpopularniejszym z nich jest **Jenkins**. Jest to darmowe [38] oprogramowanie do automatyzacji, charakteryzujące się dużą możliwością konfiguracji. Przez lata stał się standardem dla wielu firm wytwarzających oprogramowanie. Jednak jego główną wadą, wykluczającą jego użycie w projekcie GGSS, jest zbyt duża ilość pracy związa-

nej z jego konfiguracją i utrzymaniem. Narzędzie to sprawdza się dobrze przy dużej wielkości projektach. W przypadku systemu GGSS użytkowanie go przyniosłoby więcej pracy niż dałoby realnych korzyści.

Autorzy zdecydowali się więc wykorzystać narzędzie CI/CD udostępniane przez portal **GitLab**. GitLab CI/CD udostępnia wystarczająco dużo możliwości, by możliwe było wprowadzenie automatyzacji budowania i testowania w systemie GGSS. Narzędzie to można wykorzystać na dwa sposoby:

- implementując mechanizm budowania i testowania manualnie, co daje większą kontrolę nad całym procesem
- używając narzędzia **Auto DevOps**, oferującego predefiniowane konfiguracje ciągłej integracji i ciągłego dostarczania

Początkowo podjęta została decyzja o zastosowaniu drugiego z wyżej wymienionych rozwiązań. Przemawiającym za tym argumentem było potencjalne uproszczenie procedury wdrażania podejścia CI/CD do projektu - w portalu GitLab włączenie narzędzia Auto DevOps sprowadza się do zaznaczenia jednej opcji w ustawieniach na poziomie grupy lub repozytorium. Oferuje ono funkcjonalności takie, jak automatyzacja budowania i testowania projektu czy testy jakości kodu [39]. Podczas prób integracji tego rozwiązania do projektu pojawiło się jednak wiele problemów natury technicznej, związanych m.in. z niestandardowym sposobem działania projektu GGSS czy infrastrukturą tzw. *runner-ów* używaną w CERN. Ostatecznie pomysł ten został więc porzucony na rzecz manualnej konfiguracji środowiska CI/CD.

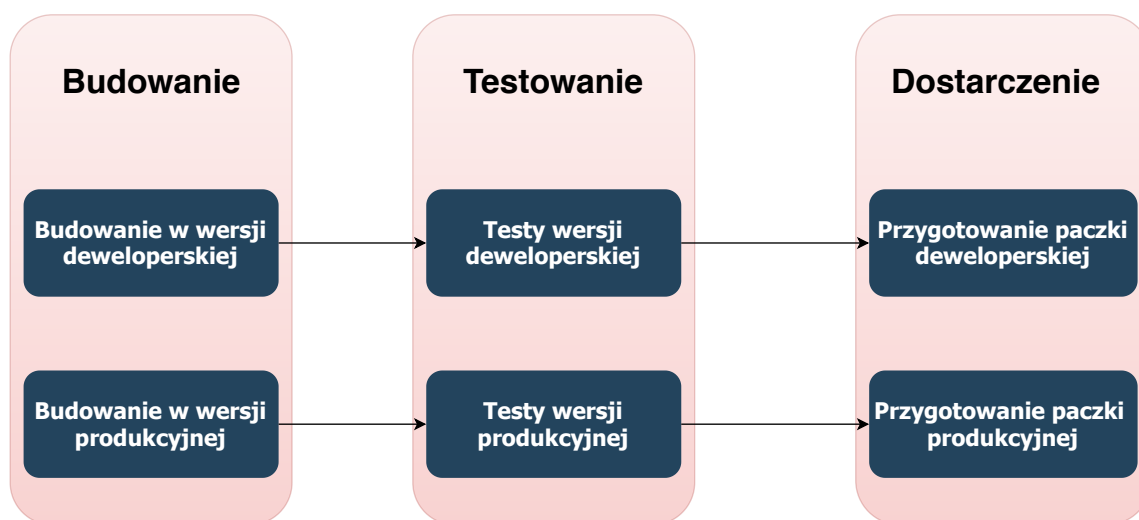
6.4.2. Opis działania GitLab CI/CD

Przed przystąpieniem do opisu sposobu zastosowania narzędzia GitLab CI/CD w projekcie GGSS przedstawiony zostanie sposób jego działania oraz najważniejsze pojęcia z nim związane. Manualna konfiguracja polega na umieszczeniu w repozytorium pliku *.gitlab-ci.yml* zawierającego szczegółowy opis przebiegu całego procesu ciągłej integracji i dostarczania (w tym konieczne do wykonania komendy). Przebieg ten określa się słowem **pipeline**. Składa się on z etapów (*stages*), przy czym każdy etap może zawierać w sobie kilka równoległych zadań (*jobs*). Idea ta została zilustrowana na Rys. 6.12.

Wynikiem każdego z zadań może być tzw. **artefakt**, czyli możliwe do pobrania archiwum zawierające np. plik wykonywalny z produktem. Artefakty mogą być również przekazywane między poszczególnymi etapami procesu.

Zadania składające się na pipeline uruchamiane są przez specjalne narzędzie **GitLab Runner**. Może on znajdować się na serwerach GitLab lub na skonfigurowanej przez klienta maszynie [40]. Przebieg całego procesu koordynowany jest przez **GitLab Server** [41].

Jak zostało wspomniane przebieg procesu CI/CD konfigurowany jest za pomocą specjalnego pliku *.gitlab-ci.yml*. Zastosowany format *YAML* (skrót od *YAML Ain't Markup Language* [42]),



Rys. 6.12. Idea działania *pipeline*. Kolorem czerwonym oznaczone zostały poszczególne etapy (*stages*), natomiast zadania (*jobs*) przedstawione zostały za pomocą ciemnoniebieskich prostokątów. Rysunek przedstawia przykładową strukturę, liczba etapów i zadań różni się zwykle od pokazanej.

ze względu na swoją czytelność, pozwala na stosunkowo szybkie przygotowanie funkcjonalnego systemu. Listing 6.11 przedstawia przykład prostego pliku w tym formacie konfiguracyjnego GitLab CI/CD. Na początku zostaje w nim określony użyty obraz *Docker'a*, następnie wymienione są poszczególne etapy (klucz *stages* - w tym przypadku jeden etap: *software_test*) procesu CI/CD, a na końcu pojawia się opis zadania (*dim_software_test*) należącego do zdefiniowanego etapu. Podczas definicji zadań możliwe jest wyspecyfikowanie mających się wykonać komend (klucz *script*).

Listing 6.11. Przykład prostego pliku *.gitlab-ci.yml* generującego jeden etap procesu CI/CD oraz jedno zadanie w ramach tego etapu

```
image: gitlab-registry.cern.ch/atlas-trt-dcs-ggss/ggss-misc/centos7

stages:
  - software_test

dim_software_test:
  stage: software_test
  script:
    - echo "Test"
```

6.4.3. Opis automatyzacji za pomocą GitLab CI/CD w projekcie GGSS

Na potrzeby projektu GGSS przygotowano zostało środowisko zapewniające ciągłą integrację i dostarczanie. Pozwala ono na automatyzację procesu budowania i testowania poszczególnych komponentów projektu oraz, dzięki mechanizmowi artefaktów, przygotowuje gotowe paczki z najważniejszymi aplikacjami w projekcie (m.in. *ggssrunner*). Każde repozytorium zawierające komponent projektu poddawany kompilacji zostało wyposażone w plik *.gitlab-ci.yml* konfigurujący zautomatyzowany proces budowania i testowania. W niniejszej części pracy zostaną przedstawione konfiguracje zrealizowane w ramach dwóch repozytoriów: *ggss-all* oraz *external-dim-lib*.

W ramach repozytorium *ggss-all* przygotowana została konfiguracja budująca aplikacje: *ggss-runner*, *ggss-dim-cs* oraz *mca-n957*. Listing 6.12 przedstawia fragment przygotowanego pliku *.yml*. Plik ten zawiera siedem zadań zdefiniowanych w ramach etapu *build* (nie ma konieczności bezpośredniej jego definicji, jest to jeden z trzech, obok *test* i *deploy*, etapów które mogą zostać, w razie pojawienia się przypisanego do nich zadania, wygenerowane automatycznie). Zadania te odpowiadają za:

- **build_all_debug_static_boost** - zbudowanie wszystkich trzech wymienionych wyżej aplikacji w wersji deweloperskiej (debug) ze statycznie dołączaną biblioteką Boost oraz przygotowanie artefaktu zawierającego wynikowe pliki wykonywalne.
- **build_all_debug_dynamic_boost** - zbudowanie wszystkich trzech aplikacji w wersji deweloperskiej z dynamicznie dołączaną biblioteką Boost oraz przygotowanie artefaktu zawierającego wynikowe pliki wykonywalne.
- **build_all_release_static_boost** oraz **build_all_release_dynamic_boost** - analogicznie do dwóch powyższych ale w wersji produkcyjnej (release).
- **build_only_ggss_runner**, **build_only_ggss_dim_cs** i **build_only_mca_n957** - zbudowanie każdej aplikacji z osobna, bez produkowania artefaktu, w wersji debug z dynamicznie linkowaną biblioteką Boost (konfiguracja domyślna).

Listing 6.12. Fragment pliku *.gitlab-ci.yml* konfigurującego *pipeline* CI/CD dla repozytorium *ggss-all*

```
image: gitlab-registry.cern.ch/atlas-trt-dcs-ggss/ggss-misc/centos7

before_script:
  - git submodule update --init --remote --recursive

# Debug builds

build_all_debug_static_boost:
  stage: build
  script:
```

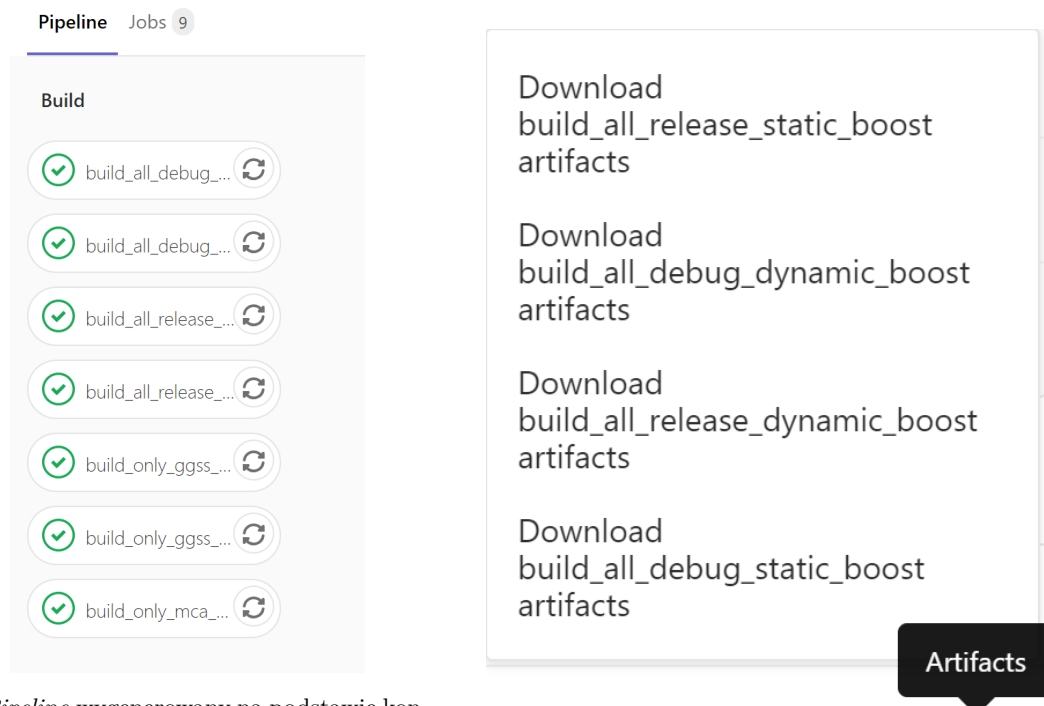
```

- mkdir build
- cd build
- python ../build.py -s --debug
artifacts:
  name: all_debug_static_boost
  paths:
    - build/ggss-dim-cs-build/ggss-dim-cs
    - build/ggss-runner-build/ggss-runner
    - build/mca-n957-build/mca-n957

```

Dalsza część pliku ...

Generowane w repozytorium *ggss-all* artefakty są tymi trafiającymi ostatecznie do środowiska produkcyjnego. Pliki mające trafić do artefaktu specyfikowane są za pomocą klucza *paths* (widoczne na listingu 6.12). Do przeprowadzenia budowania używany jest specjalnie w tym celu przygotowany przez autorów obraz, oparty na oficjalnych obrazach dostarczonych przez CERN. Rys. 6.13 przedstawia wynikowy *pipeline* oraz dostępne do pobrania artefakty.

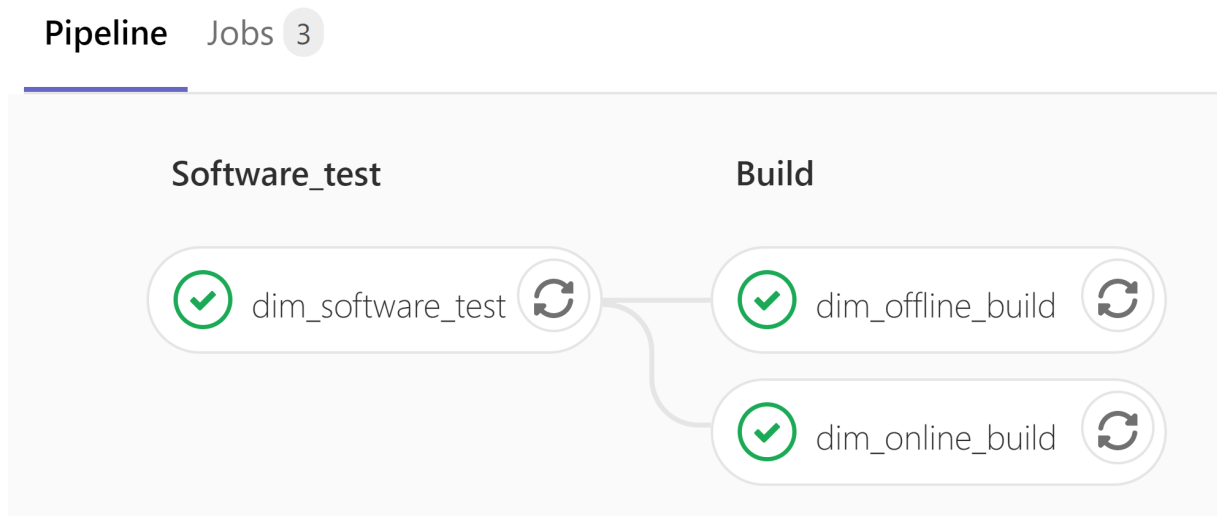


(a) *Pipeline* wygenerowany na podstawie konfiguracji zamieszczonej w pliku *.yml* (listing 6.12)

(b) Możliwe do pobrania artefakty

Rys. 6.13. Zrzuty ekranu wykonane w serwisie GitLab przedstawiające elementy działania ciągłej integracji i dostarczania dla repozytorium *ggss-all*

Konfiguracja stworzona na potrzeby repozytorium *external-dim-lib* różni się nieznacznie od opisanej do tej pory jeśli chodzi o techniczną stronę jej realizacji. Przygotowany *pipeline* (Rys. 6.14) składa się z innych etapów i zadań, ale zostały one skonfigurowane z użyciem zaprezentowanych już technik. Na potrzeby przygotowania biblioteki obsługującej protokół *DIM* zostały więc stworzone dwa etapy: odpowiadający za przeprowadzenie testów skryptu pobierającego bibliotekę ze strony internetowej (*Software_test*), oraz odpowiadający na zbudowanie jej w dwóch wersjach (*Build*). Obie wersje udostępniane są do pobrania za pomocą mechanizmu artefaktów.



Rys. 6.14. Pipeline wygenerowany na potrzeby repozytorium *external-dim-lib*

Podobnego typu prace zostały wykonane dla większości pozostałych repozytoriów. Z uwagi na ich powtarzalny charakter nie zostaną one jednak omówione w niniejszym manuskrypcie.

W przedstawionym przykładzie dotyczącym aplikacji *ggssrunner* (listing 6.12) znajduje się klucz *before_script* z instrukcją pobierającą submoduły projektu. Istnieją dwa wykluczające się sposoby na wykonanie tej czynności. Submoduł może zostać pobrany w swojej najnowszej wersji znajdującej się na **zdalnej rewizji** (tak jak w przytoczonym przykładzie, służy temu opcja **remote**) lub w wersji **aktualnie powiązanej z repozytorium nadrzędnym** (realizację przedstawia listing 6.13). Oba wymienione podejścia mają swoje wady i zalety. Rys. 6.15 stanowi uproszczoną (ograniczoną tylko do jednej gałęzi na repozytorium) ilustrację opisywanego problemu. Pierwszy z wymienionych sposobów jest wygodniejszy, ponieważ pozwala na dostarczenie i przetestowanie najnowszej możliwej wersji produktu. Jest to jednak ryzykowne - najnowsza wersja submodułu może nie być kompatybilna z modułem nadrzędnym (np. mógł zmienić się jej interfejs). Spowoduje to błąd, którego źródło może być trudne do odnalezienia (jeśli programista owej najnowszej wersji nie używał pojawi się sprzeczność w wynikach otrzymywanych przez niego lokalnie, a generowanych przez mechanizm CI/CD). Drugie rozwiązanie jest zatem bezpieczniejsze i gwarantuje stabilność. Autorzy skłaniają się w większości przypad-

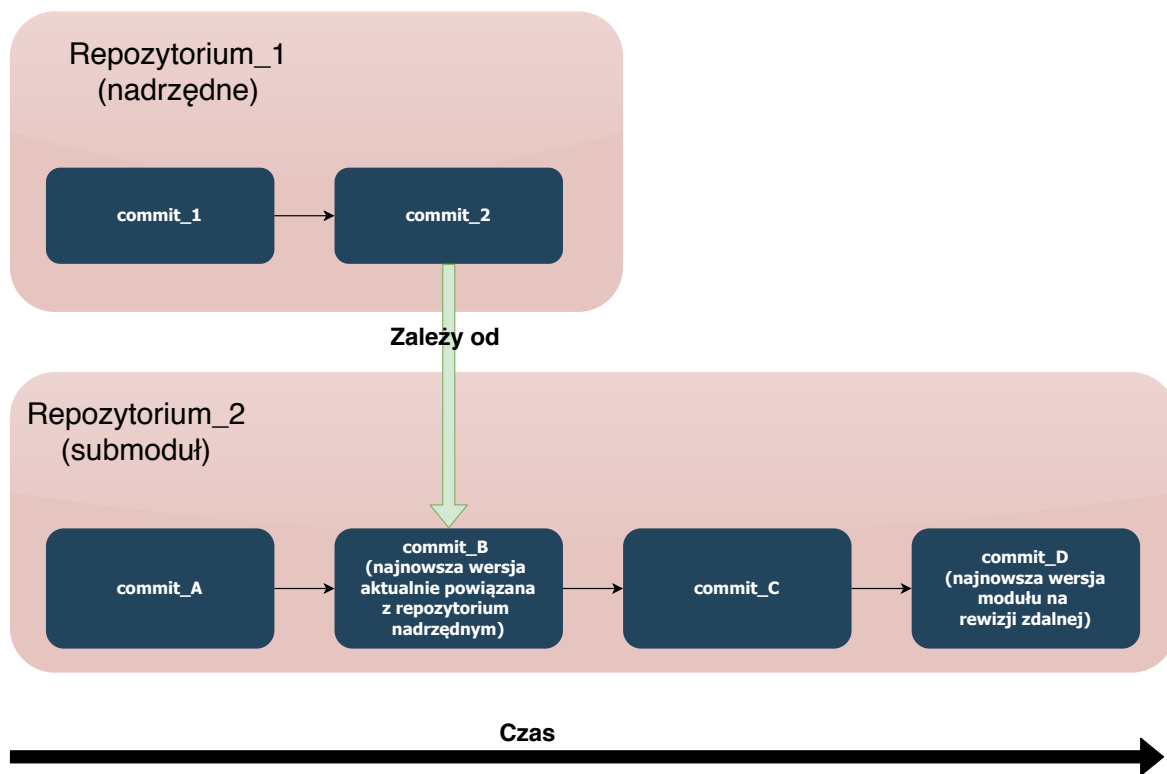
ków do tego właśnie rozwiązania, jednak z uwagi na zalety pierwszej z wymienionych opcji, jest ona wciąż używana w niektórych repozytoriach (np. *ggss-all*).

Listing 6.13. Fragment pliku *.gitlab-ci.yml* znajdującego się w repozytorium *ggss-software-libs* pobierający submodule projektu w wersji aktualnie powiązanej z repozytorium nadrzędnym

```
image: gitlab-registry.cern.ch/atlas-trt-dcs-ggss/ggss-misc/centos7

variables:
  GIT_SUBMODULE_STRATEGY: recursive

# Dalsza część pliku
```



Rys. 6.15. Uproszczona (ograniczona do jednej gałęzi na repozytorium) ilustracja problemu dotyczącego pobieranej przez mechanizm CI/CD wersji submodułu. Problem polega na podjęciu decyzji, czy powinna zostać pobrana wersja najnowsza (tutaj oznaczona jako *commit_D*) czy ta aktualnie powiązana z repozytorium nadrzędnym (*commit_B*)

W ramach prac nad mechanizmem CI/CD w projekcie GGSS został również przygotowany specjalny **szablon** ułatwiający tworzenie *pipeline'ów*. Został on umieszczony w repozytorium *aux*, a jego fragment przedstawia listing 6.14. Zawartość tego szablonu, zmodyfikowana zgodnie ze specyfiką danego repozytorium, posłużyła autorom do napisania większości plików *.yml*, które znalazły się w projekcie.

Listing 6.14. Fragment szablonu ułatwiającego pisanie plików *.gitlab-ci.yml* znajdującego się w repozytorium *aux*

```
# This is gitlab-ci.yml for TRT GGSS project.

# actual image being used by ggss project
image: gitlab-registry.cern.ch/atlas-trt-dcs-ggss/ggss-misc/centos7

# variable needed for submodule clone - can be moved to job locally
# to use submodules on docker without adding ssh key please use relative ↔
  submodules path
variables:
  GIT_SUBMODULE_STRATEGY: recursive

build:
  stage: build
# input your building under scripts
  script:
#   e.g:
#   - cmake
#   - make

#   artifacts are being used to store and share files, for,
#   insert proper path that should be shared, e.g.:
  artifacts:
    paths:
#     - build/ggss-driver-cc7*

#   tags allow to choose on which runner jobs should be executed
  tags:
    - ggss-builder # default ggss runner

# Dalsza część pliku
```

6.5. Budowanie i dystrybucja sterownika oraz aplikacji testującej

Ważną częścią pracy wykonanej przez autorów były zmiany w systemie budowania i dystrybucji pakietu **RPM** zawierającego pliki modułu **ggss-driver**, czyli:

- **CAENUSBdrvB.ko** - sterownik do analizatora wielokanałowego w postaci modułu jądra systemu operacyjnego
- **10-CAEN-USB.rules** - zasady działania dla programu udev, ma na celu utworzenie dowiązania symbolicznego o odpowiednich prawach dostępu w momencie utworzenia urządzenia odpowiedzialnego za obsługę sterownika
- **99-GGSS.rules** - zasady działania dla programu udev, odpowiadają za utworzenie dowiązań symbolicznych w momencie podłączenia urządzeń USB o zadanych atrybutach.
- **CAEN-USB.modules** - plik odpowiedzialny za konfigurację aplikacji **modules** w taki sposób, aby sterownik był ładowany automatycznie w trakcie uruchamiania systemu operacyjnego
- **ggss_conf.py** - skrypt napisany w języku **Python 2** pozwalający na uproszczone generowanie pliku 99-GGSS.rules dla programu udev
- **ggss_usb.conf** - aktualnie używana konfiguracja dla skryptu **ggss_conf.py**

Pierwszym krokiem wykonanym w ramach modułu **ggss-driver** była zmiana logiki budowania w celu zapewnienia usuwania plików pomocniczych tworzonych podczas tego procesu, aby zapewnić uporządkowane środowisko deweloperskie. Zostało to osiągnięte za pomocą dodania odpowiedniej właściwości folderowi, który zawierał niepotrzebne dłużej zasoby, w ramach pliku *CMakeLists.txt*.

Kolejną ważną kwestią było uporządkowanie i poprawienie niedociągnięć znajdujących się w pierwotnej wersji pliku *CMakeLists.txt*. W ramach poprawionej logiki została dodana informacja o wersji jądra systemu operacyjnego użytego w celu kompilacji sterownika. Informację tą można wyświetlić za pomocą komendy **rpm** po zainstalowaniu pakietu w celu sprawdzenia, czy wersja jądra systemu, za pomocą której został utworzony sterownik, jest zgodna z wersją jądra na systemie docelowym. Jest to bardzo ważna funkcjonalność ze względu na to, że różnice nawet w wersji **minor** potrafią powodować błędy w działaniu lub kompletnie uniemożliwiać działanie sterownika.

Kolejną poprawką wprowadzoną do pliku *CMakeLists.txt* było generowanie skryptów obsługujących pakiet **RPM** bezpośrednio do pakietu, a nie jak to miało miejsce wcześniej, czyli zapisywanych do pliku na dysku twardym, a dopiero następnie wprowadzanych do pakietu. Dodatkowo logika samych skryptów uległa poprawie, pozbyto się nadmiarowego usuwania plików,

które było wykonywane automatycznie przez menadżera pakietów. W ramach pakietu dodano również bibliotekę **libCAENN957**, która jest dostarczana przez firmę **CAEN** i ma na celu udostępnienie **API** do obsługi analizatora wielokanałowego. Została ona dodana jako wymagany komponent całego pakietu, w przypadku braku odnalezienia biblioteki proces budowania nie powiedzie się. Repozytorium ją zawierające zostało dodane jako **submoduł** repozytorium **ggss-driver**. W celu zapewnienia odpowiedniej obsługi biblioteki w systemie docelowym generowany jest plik konfiguracyjny programu **ld** zawierający ścieżkę instalacyjną biblioteki. W ramach logiki instalacyjnej program **ld** jest odświeżany, tak, aby ścieżka z pliku została zarejestrowana. W celu poprawnego linkowania biblioteki tworzone jest dowiązanie symboliczne, które nie zawiera numeru wersji w nazwie (**libCAENN957.so** -> **libCAENN957.so.1.6**).

Bardzo ważnym aspektem było wprowadzenie obsługi systemu aktualizacja pakietów. W celu zaktualizowania obecnie zainstalowanego pakietu do nowszej wersji wykonywana jest logika instalacyjna nowego pakietu oraz logika dezinstalacyjna starego pakietu w następującej kolejności:

- Wykonaj skrypt pre-instalacyjny nowego pakietu
- Zainstaluj pliki dostarczane przez nowy pakiet
- Uruchom sekcję post-instalacyjną nowego pakietu
- Uruchom sekcje pre-dezinstalacyjną starego pakietu
- Usuń pliki dostarczane przez stary pakiet, ale tylko te, które nie zostały nadpisane przez nowy pakiet.
- Uruchom sekcje post-dezinstalacyjną starego pakietu. [43]

Ze względu na to, że w ramach logiki instalacyjnej ładowany jest moduł jądra systemu operacyjnego, a w ramach logiki dezinstalacyjnej jest on usuwany wymagane było przeniesienie logiki dezinstalacyjnej starego pakietu przed logikę instalacyjną nowego pakietu, tak, aby po procesie aktualizacji moduł był nadal załadowany. Niestety ww. przebieg procesu aktualizacji uniemożliwiał wykonanie tego w prosty sposób, ze względu na to, że żadna część logiki nowego pakietu nie jest wykonywana na koniec, dzięki czemu moglibyśmy załadować moduł jądra. W celu rozwiązania tego problemu, została wykorzystana właściwość parametru **\$1** w ramach logiki pakietów **RPM**, czyli:

- W przypadku procesu instalacyjnego, gdy argument **\$1** przyjmuje wartość **1** oznacza to, że przeprowadzana jest czysta instalacja, natomiast, gdy argument przyjmuje wartość **2** przeprowadzana jest aktualizacja
- W przypadku procesu dezinstalacyjnego, gdy argument **\$1** przyjmuje wartość **0** oznacza to, że przeprowadza jest dezinstalacja, natomiast, gdy argument przyjmuje wartość **1** przeprowadzana jest aktualizacja

Wykorzystując ww. właściwości logikę dezinstalacyjną starego pakietu, w przypadku wykonywania aktualizacji, została przeniesiona do skryptu odpowiedzialnego za instalację nowego pakietu, przed wykonaniem jakichkolwiek czynności jego dotyczących, co ukazane jest na Listingu 6.15, natomiast skrypt dezinstalacyjny służy jedynie wyświetlaniu informacji o braku podejmowanych kroków w jego części procesu (Listing 6.16).

Listing 6.15. Przykład wykorzystania właściwości argumentów dla skryptu *post-install*

```
#!/bin/sh
# [ $1 = 2 ] - upgrade
# [ $1 = 1 ] - install
if [ $1 = 2 ]; then
    # Dokonaj czyszczenia starego pakietu
fi
# Dokonaj instalacji nowego pakietu
```

Listing 6.16. Przykład wykorzystania właściwości argumentów dla skryptu *pre-uninstall*

```
#!/bin/sh
# [ $1 = 1 ] - upgrade
# [ $1 = 0 ] - uninstall
if [ $1 = 0 ]; then
    # Dokonaj dezinstalacji starego pakietu
elif [ $1 = 1 ]; then
    # Wyświetl komunikat o braku podejmowanych akcji, ze względu na dezinstalację ←
    wykonaną przez skrypt post-install
fi
```

W ramach zmian w module **ggss-driver** zostały również poprawione prawa dostępu do plików instalowanych przez pakiet, tak, aby w systemie docelowym osoby korzystające miały do nich dostęp. Ze względu na to, że miejscem docelowym plików, instalowanych przez pakiet, są foldery, których ścieżki znajdują się w strukturze folderów systemowych wymagane było wyłączenie części z nich z tworzenia w procesie instalacyjnym pakietu. Są to następujące katalogi: */opt*, */usr/lib/modules*, */etc/ld.so.conf.d*, */etc/modules-load.d*, */etc/udev*, */etc/udev/rules.d*.

Każda zmiana w pakiecie **RPM** projektu **ggss-driver** jest odpowiednio wersjonowana, dla każdej z nich jest budowany nowy pakiet w ramach funkcjonalności GitLab CI/CD, który jest udostępniany jako artefakt do pobrania.

W ramach projektu **GGSS** dystrybuowana jest również aplikacja **mcaN957** służąca do testowania poprawności działania analizatora wielokanałowego. Dostarczana jest ona w postaci pakietu RPM, do instalacji którego wymagana jest wcześniejsza instalacja pakietu **ggss-driver**.

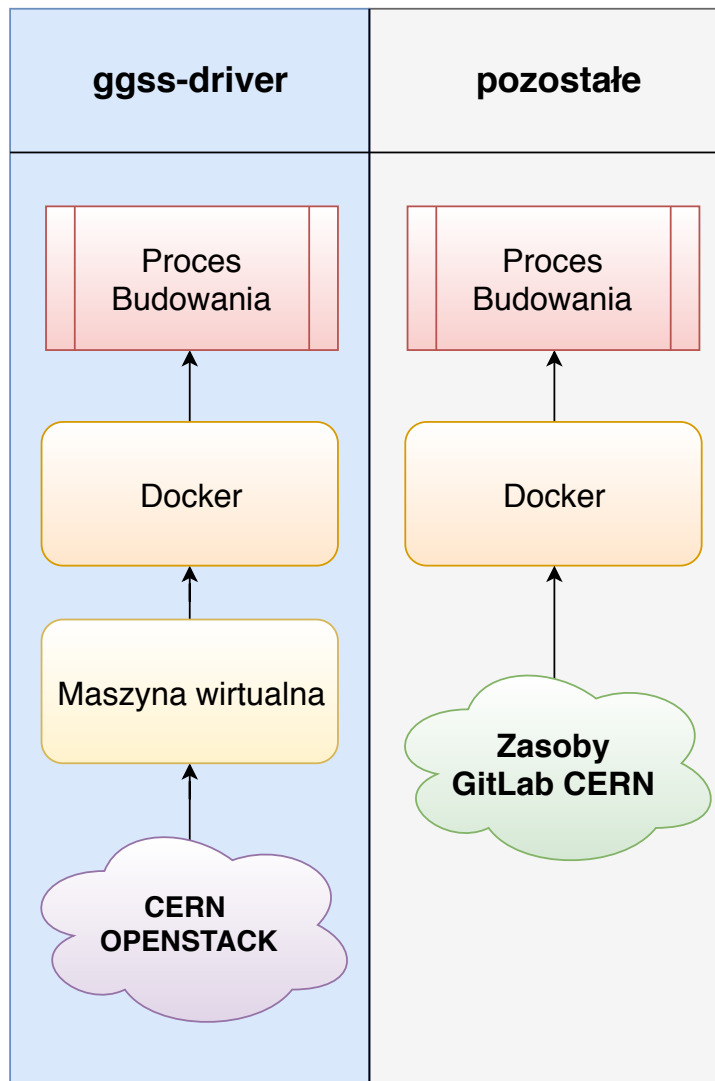
Utworzony został również pakiet zawierający aplikację **CAEN-N957-DEMO**, która również służy do testowania poprawności działania ww. urządzenia, natomiast jest ona tworzona przez firmę **CAEN**. Również ten pakiet wymaga ówczesnej instalacji pakietu **ggss-driver**.

6.6. Maszyna wirtualna oraz konteneryzacja

Ze względu na to, że maszyny dostarczone przed administratorów systemu nie zapewniały odpowiedniej zgodności z maszyną docelową autorzy musieli zapewnić odpowiednie środowisko produkcyjne w inny sposób. Pierwszym wykorzystanym pomysłem było użycie ogólnodostępnych **GitLab CI/CD runner’ów** udostępnianych w ramach portalu **GitLab CERN**. Są to współdzielone maszyny wirtualne, w ramach których uruchamiany jest kontener **Docker**, na którym wykonywane są polecenia zawarte w ramach zadań (*.gitlab-ci.yml*) **GitLab CI/CD**. Niestety rozwiązanie to nie dawało gwarancji zachowania zgodności jąder systemów operacyjnych. Ze względu na wykorzystanie przez **Docker’a** jądra systemu operacyjnego gospodarza nie można z poziomu **kontenera** wykonać zamiany jądra na inną wersję. Biorąc to pod uwagę, jak i brak możliwości administracyjnych na ww. zasobach autorzy nie mogli w pełni wykorzystać tego podejścia do otrzymania odpowiedniego środowiska deweloperskiego.

W celu rozwiązania problemu różnych wersji jądra systemu operacyjnego postanowiono utworzyć maszyny wirtualne w ramach usługi **Cern Openstack Service** mającej na celu udostępnianie zasobów w chmurze. Użytkownicy tego rozwiązania posiadają prawa administracyjne na utworzonych instancjach. Dzięki temu mogą one zostać przystosowane w taki sposób, aby zgadzała się z konfiguracją środowiskową maszyny docelowej, a w szczególności wersji jądra. Aby zwiększyć stabilność oraz powtarzalność środowiska deweloperskiego wykorzystana została technologia **Docker**, dzięki czemu każde zadanie w ramach **GitLab CI/CD** miało zapewnione takie same warunki. Dzięki wykorzystaniu konteneryzacji żadne efekty uboczne zadania nie są w stanie wpłynąć na inne zadanie. Znacząco zwiększa to niezawodność środowiska deweloperskiego.

Ze względu na ograniczone zasoby dostępne w ramach usługi **Cern Openstack Service** są one wykorzystywane jedynie w przypadku repozytorium **ggss-driver**, ze względu na mocne ograniczenie w postaci zgodności wersji jądra operacyjnego. Pozostałe repozytoria wykonują swoje zadania w ramach zasobów współdzielonych na portalu **GitLab CERN**. Rysunek 6.16 przedstawia architekturę od strony maszyn wirtualnych oraz kontenerów zastosowaną w projekcie.



Rys. 6.16. Porównanie architektury dla **ggss-driver** oraz pozostałych repozytoriów

W ramach konteneryzacji został przygotowany obraz **Docker** (atlas-trt-dcs-ggss/ggss-misc/centos7), który umieszczono w rejestrze obrazów **gitlab-registry.cern.ch**. Obraz ten zbudowano na podstawie **Dockerfile** w ramach którego zostały zawarte wszystkie zależności potrzebne do zbudowania projektu **GGSS**, czyli m.in.: Make, CMake, kompilator C++ oraz C, Python, biblioteki Pythonowe etc. **Dockerfile** użyty w tym celu został ukazany na listingu 6.17.

Listing 6.17. Dockerfile dla projektu GGSS

```
FROM cern/cc7-base:latest

COPY PYTHON/requirements.txt requirements.txt
ENV BOOST="/usr/local/boost"

RUN yum -y install cmake \
make \
unzip \
gcc \
gcc-c++ \
python3-pip \
git \
kernel-3.10.0-862.14.4.el7 \
kernel-devel-3.10.0-862.14.4.el7 \
rpm-build \
gsl \
gsl-devel \
libcud

RUN ln -s /usr/bin/pip3 /usr/bin/pip
RUN pip install -r requirements.txt
```

6.7. Dokumentacja projektu

Jednym z wymagań postawionych przed autorami niniejszej pracy było przygotowanie podstawowej dokumentacji opisującej poszczególne komponenty projektu GGSS. Dokumentacja ta miała zostać sporządzona w języku angielskim, co wynika z międzynarodowego charakteru zespołu pracującego nad detektorem ATLAS. Przygotowana dokumentacja powinna oferować możliwie niski próg wejścia - powinna ona umożliwić osobie niezaznajomionej z całym systemem GGSS zbudowanie oraz użycie poszczególnych jego komponentów, jak i całego projektu. Autorzy zrealizowali to zadanie za pomocą plików **README.md** umieszczonych w każdym repozytorium projektu oraz dokumentów opisujących sposób wykonania niektórych czynności związanych z utrzymaniem projektu.

6.7.1. Język Markdown

Pliki *readme* umieszczane w serwisach takich jak *GitLab* czy *GitHub* pisane są przy pomocy specjalnego języka znaczników **Markdown**. Jest to prosty w użyciu język, konwertowany zwykle do odpowiadającego mu kodu *HTML* [44]. Listing 6.18 przedstawia prosty przykład pliku napisanego w języku **Markdown**, natomiast Rys. 6.17 przedstawia wygenerowaną za jego pomocą zawartość w portalu *GitLab*.

Listing 6.18. Przykład prostego pliku napisanego w języku **Markdown**

```
# Przykład nagłówka

Ten przykład został wygenerowany na potrzeby pracy inżynierskiej dotyczącej
systemu GGSS.

## Autorzy
Przykład listy:
* Arkadiusz Kasprzak
* Jarosław Cierpich
```

Przykład nagłówka

Ten przykład został wygenerowany na potrzeby pracy inżynierskiej dotyczącej systemu GGSS.

Autorzy

Przykład listy:

- Arkadiusz Kasprzak
- Jarosław Cierpich

Rys. 6.17. Przykład zawartości wygenerowanej w portalu *GitLab* za pomocą języka **Markdown** na podstawie kodu z listingu 6.18

6.7.2. Opis wykonanej dokumentacji

Dla każdego repozytorium w projekcie wykonany został plik *README.md* zawierający informacje takie jak:

- opis zawartości repozytorium (np. lista bibliotek wraz z krótkim ich opisem)
- wymagania dotyczące środowiska, w którym projekt może zostać zbudowany (np. niezbędne biblioteki zewnętrzne, wersja narzędzia *CMake*)
- szczegółowy opis procesu budowania projektu wraz z gotową komendą umożliwiającą jego zbudowanie

Rys. 6.18 przedstawia fragment zawartości wygenerowanej za pomocą pliku *README.md* znajdującego się w repozytorium *ggss-hardware-libs*, opisującej proces budowania bibliotek wchodzących w skład tego repozytorium. Poza listą kroków opisującą szczegółowo budowanie zestawu bibliotek, instrukcja zawiera również pojedynczą komendę, użycie której wykonuje cały proces bez konieczności dodatkowego wkładu ze strony użytkownika. Takie podejście ma na celu ułatwienie osobom nieznającym struktury projektu szybkie jego zbudowanie. Podobne komendy umieszczone zostały w repozytoriach zawierających aplikacje wchodzące w skład systemu GGSS (takie jak *ggssrunner*).

Oprócz plików *.md* przygotowane zostały również dwa dokumenty opisujące ważne, zdaniem autorów, aspekty utrzymania i rozwoju projektu GGSS. Dokumenty te zostały dołączone do niniejszej pracy w formie dodatków (A.2 oraz A.3). Dotyczą one rozszerzania projektu o nowe moduły (dodatek A.2 - *Adding modules to the project using existing CMake templates*) oraz przygotowywania maszyny wirtualnej do pracy jako *runner* w procesie CI/CD (dodatek A.3 - *Preparing the virtual machine to work as a runner*).

Building

User can build libraries separately or as one project. In both cases, out-of-source building is supported.

Building whole project

To build all libraries at once, use the following commands:

- `git clone ssh://git@gitlab.cern.ch:7999/atlas-trt-dcs-ggss/ggss-hardware-libs.git`
to clone the repository from GitLab
- `mkdir <build_directory>` where *build_directory* should be CMake output directory
- `cd ggss-hardware-libs`
- `git submodule update --init --recursive --remote`
- `cd ../<build_directory>`
- `cmake ../ggss-hardware-libs`
- `make`

You can also use the following *one-liner*:

```
git clone ssh://git@gitlab.cern.ch:7999/atlas-trt-dcs-ggss/ggss-hardware-libs.git &&  
mkdir ggss-hardware-libs-build && cd ggss-hardware-libs && git submodule update --init --  
recursive --remote && cd ../ggss-hardware-libs-build && cmake ../ggss-hardware-libs &&  
make
```

Rys. 6.18. Fragment instrukcji budowania zestawu bibliotek znajdujących się w repozytorium *ggss-hardware-libs*

6.8. Pomniejsze prace

Niniejsza część pracy zawiera opis dwóch problemów niezwiązanych bezpośrednio z omówionymi dotychczas pracami. W obu przypadkach ich cechą charakterystyczną było krótkie, jeśli chodzi o ilość kodu, rozwiązanie, które wymagało jednak od autorów poświęcenia znacznej ilości czasu na zdobycie odpowiedniej wiedzy lub znalezienie źródła powstałego utrudnienia. Ze względu na fakt, iż rozwiązanie tych problemów było niezbędne, by projekt mógł działać poprawnie, autorzy zdecydowali się poświęcić im osobną sekcję niniejszego manuskryptu.

6.8.1. Integracja bibliotek napisanych w języku C z aplikacją w C++

Biblioteka zawarta w ramach repozytorium **external-n957-lib** napisana została w języku C, co utrudnia poprawne wykorzystanie jej w projekcie opartym na języku C++. Wynika to ze sposobu działania symboli w języku C++. Ze względu na istniejące w nim mechanizmy takie jak: przeładowanie funkcji oraz szablony, są one dekorowane w celu przechowywania dodatkowych, koniecznych informacji. Zachowanie to jest niezgodne z zachowaniem w języku C, gdzie taki mechanizm nie występuje. Powoduje to konieczność specjalnego oznaczenia kodu napisanego w języku C, a wykorzystywanego w języku C++ konstrukcją `extern "C"`. W przeciwnym wypadku, podczas linkowania programu, pojawi się błąd dotyczący niezdefiniowanych referencji do symboli.

W początkowej wersji projektu problem ten został rozwiązany poprzez dodanie wyżej wymienionej konstrukcji w plikach nagłówkowych zewnętrznej biblioteki, co po jej aktualizacji spowodowało zaprzestanie poprawnego budowania się projektu **GGSS**. Poprawa tego błędu polegała na wykorzystaniu `extern "C"` w miejscu występowania dyrektywy `include` dołączającej wyżej wspomniane pliki nagłówkowe biblioteki, co ilustruje listing 6.19.

Listing 6.19. Wykorzystanie konstrukcji `extern "C"` w celu integracji biblioteki napisanej w języku C z komponentem **mca-lib**

```
extern "C" {  
    #include <mca/N957Lib.h>  
}
```

6.8.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake

Narzędzie *CMake* dostarcza gotowe API pozwalające dodawać do projektu zewnętrzne biblioteki w prosty sposób, co przedstawia listing 6.20. Przedstawiona konfiguracja uwzględnia fakt, iż biblioteka nie posiada zdefiniowanego pola *SONAME*, co utrudnia poprawne dodanie jej do projektu. Utrudnienie polega na tym, że dodając bibliotekę w sposób standardowy w miejscu *SONAME* pojawiała się pełna ścieżka do pliku, co po przeniesieniu aplikacji na maszynę docelową powodowało błędy. Do pełnego rozwiązania problemu konieczne było również utworzenie dowiązania symbolicznego w miejscu, z którego była importowana biblioteka zgodnie ze standardem opisanym w rozdziale 2.2.

Listing 6.20. Dodanie zewnętrznej biblioteki do projektu za pomocą narzędzia *CMake*

```
add_library(CAENN957 SHARED IMPORTED GLOBAL)
set_target_properties(CAENN957 PROPERTIES
    IMPORTED_NO_SONAME TRUE
    IMPORTED_LOCATION ${CAENLibPath}
)
```

7. Testy nowej wersji oprogramowania

Niniejszy rozdział zawiera opis testów systemu GGSS przeprowadzonych na koniec trwania prac związanych z jego oprogramowaniem. Celem testów była weryfikacja poprawności działania czterech konfiguracji programu *ggssrunner*: wersja deweloperska (*debug*) ze statycznie linkowaną biblioteką *Boost*, wersja deweloperska z dynamicznie linkowaną biblioteką *Boost* oraz analogiczne wersje produkcyjne (*release*). Z uwagi na powtarzalny charakter procesu testowania zaprezentowany zostanie jedynie przebieg testów dla wersji **deweloperskiej ze statycznie linkowaną biblioteką Boost**.

7.1. Przebieg testu

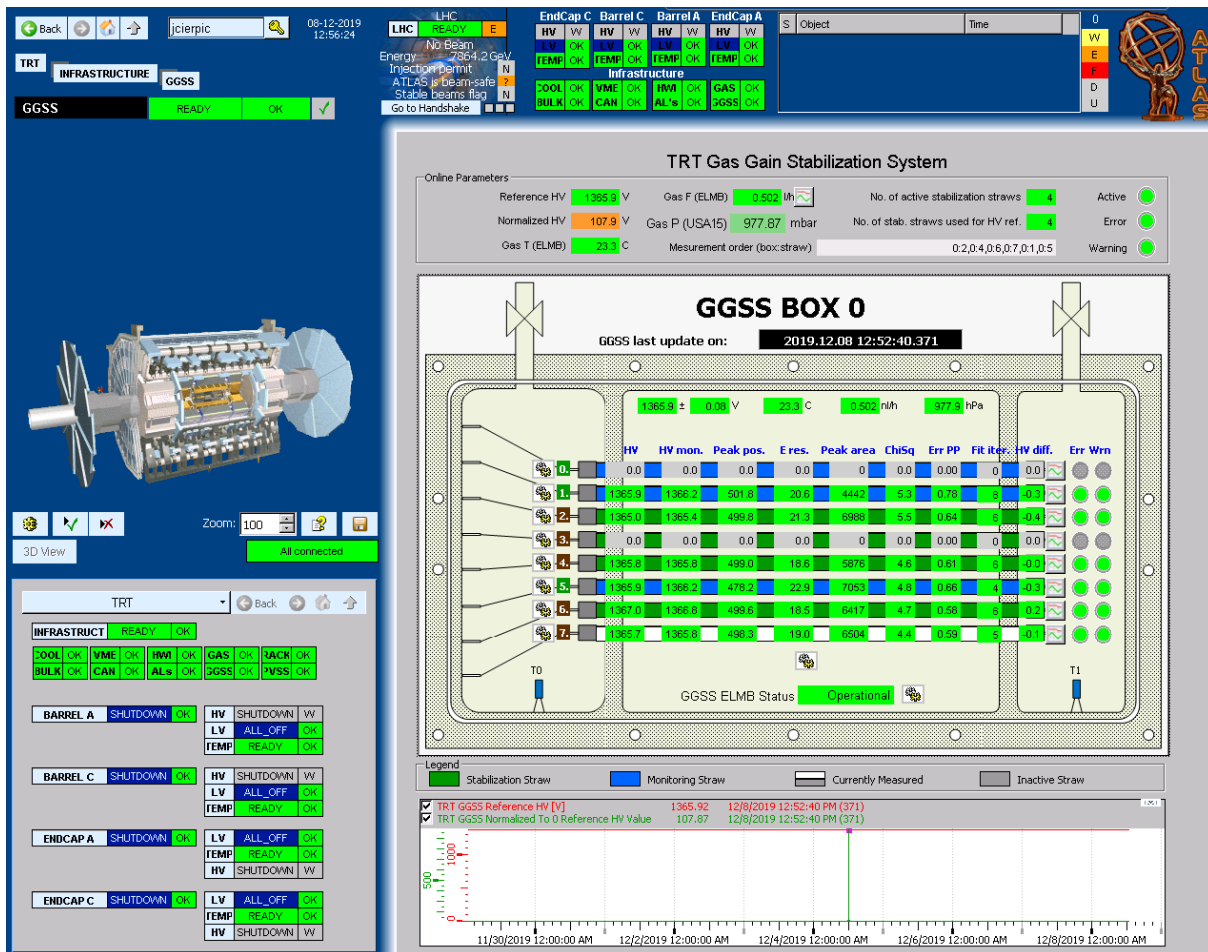
Ze względu na fakt, że środowisko w którym osadzony jest program *ggssrunner* jest ciągle monitorowane, pierwszym krokiem było umieszczenie informacji o przeprowadzaniu testów w dedykowanym do tego celu systemie **ELisA (Electronic Logbook for Information Storage for Atlas)**.

Jarosław Piotr Cierpich: GGSS maintenance			
Jarosław Piotr Cierpich	GGSS	Tests of GGSS project in different deploy configurations.	2019-12-08 12:54
ID:	413410		
Status:	open		
Message Type:	Default Message Type		
System Affected:	DCS, TRT		
Tests of GGSS project in different deploy configurations. Tests are expected to finish on Tuesday (10th of December) morning.			Info Edit Reply

Rys. 7.1. Informacja o przeprowadzaniu testów w systemie ELisA

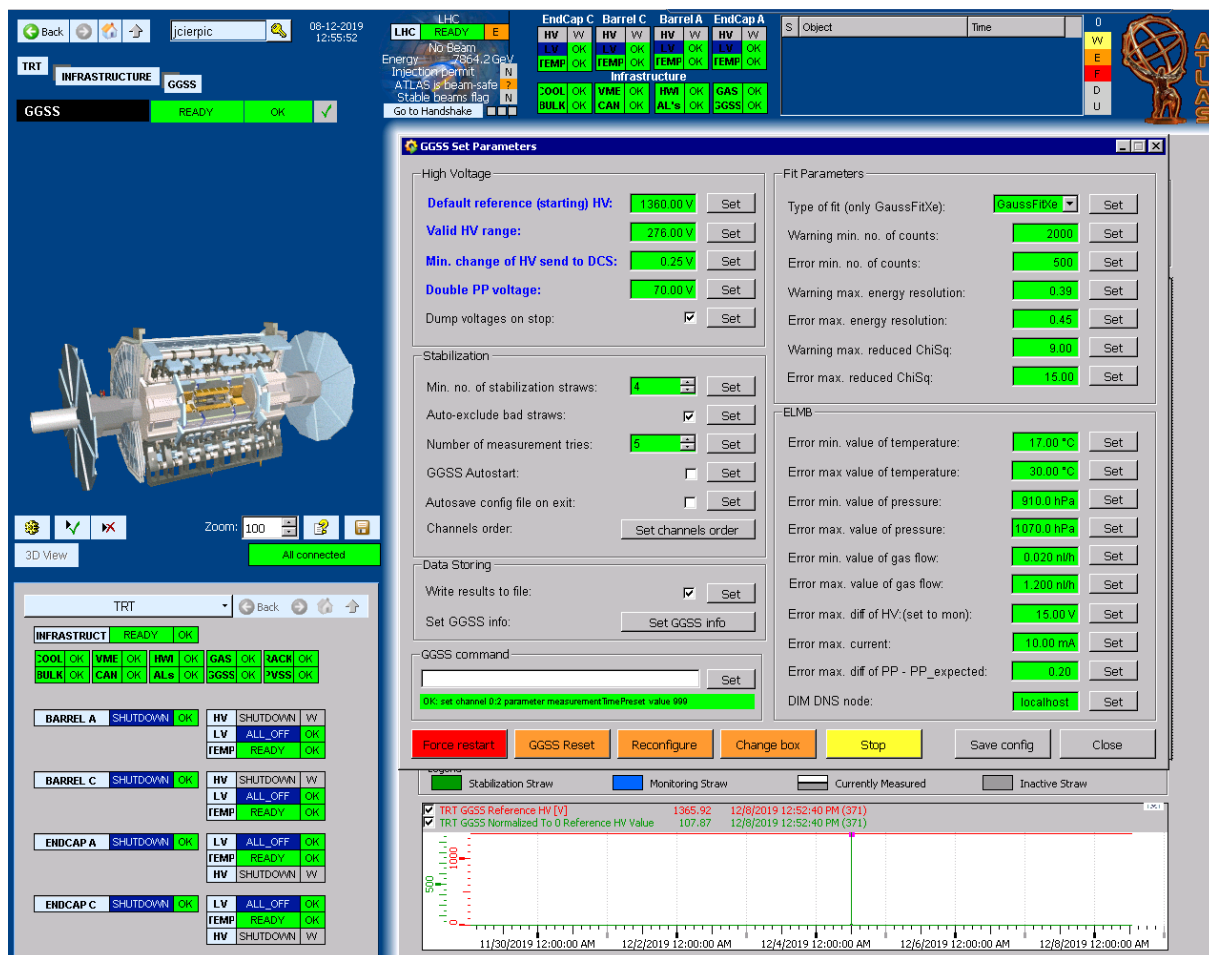
Pliki wykonywalne aplikacji *ggssrunner* wygenerowane zostały za pomocą przygotowanego przez autorów środowiska CI/CD. Zostały one umieszczone na komputerze produkcyjnym. Kolejnym krokiem było zalogowanie się do panelu *WinCC OA* służącego do monitorowania działania detektora ATLAS oraz wybranie panelu odpowiedzialnego za dostarczanie informacji o systemie GGSS.

7. Testy nowej wersji oprogramowania



Rys. 7.2. Panel WinCC OA monitorujący działanie systemu GGSS

Następnym etapem było przeprowadzenie procesu wyłączania systemu GGSS za pomocą przycisku *Stop* znajdującego się na dedykowanym panelu konfiguracyjnym ukazanym na Rys. 7.3.



Rys. 7.3. Panel konfiguracyjny systemu GGSS podczas działania systemu

Po wyłączeniu systemu należało również przerwać działanie poprzedniej wersji aplikacji *ggssrunner* za pomocą skryptu *ggss_monitor.sh* (listing 7.1). Za pomocą tego skryptu został również potwierdzony stan aplikacji po wyłączeniu.

Listing 7.1. Zatrzymanie działania aplikacji *ggssrunner*

```
user@host:~$ ./ggss_monitor.sh check
ggssrunner is running.

user@host:~$ ./ggss_monitor.sh stop
ggssrunner: no process found
Creating lock /localdisk/ggss/bin/autostartggss.lock

user@host:~$ ./ggss_monitor.sh check
ggssrunner is NOT running. /localdisk/ggss/bin/autostartggss.lock exists. Remove ←
it or start GGSS manually
```

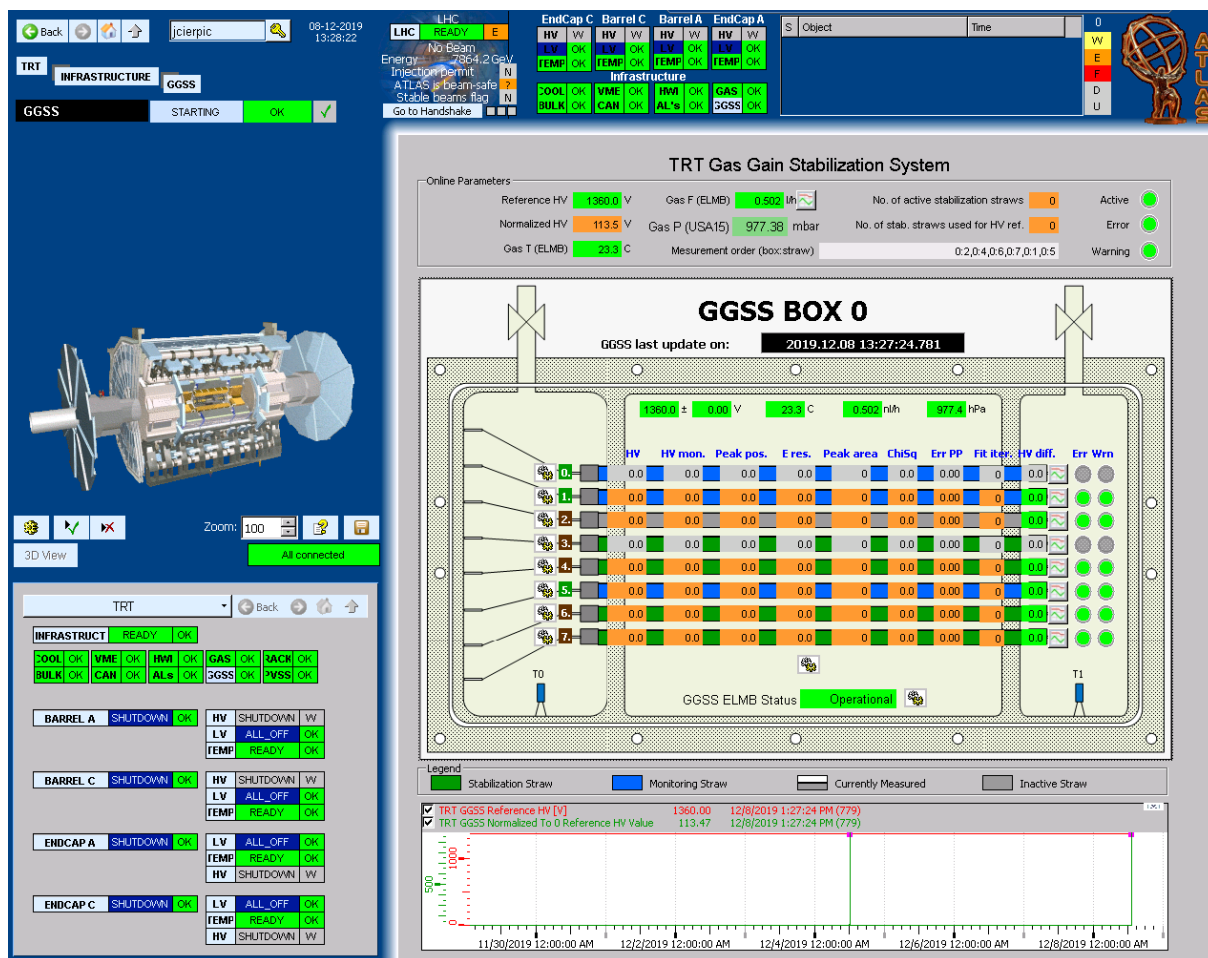
Po wykonaniu wyżej wymienionych czynności podmieniony został plik wykonywalny aplikacji *ggssrunner* na przygotowany przez autorów. Zmiana została wykonana poprzez modyfikację **dowiązania symbolicznego**. Następnie aplikacja uruchomiona została ponownie za pomocą skryptu *ggss_monitor.sh* (listing 7.2) oraz z poziomu panelu *WinCC OA*. Stan panelu monitorującego system GGSS jest widoczny na Rys. 7.4.

Listing 7.2. Ponowne uruchomienie aplikacji *ggssrunner*

```
user@host:~$ ./ggss_monitor.sh remove_lock
Removing lock /localdisk/ggss/bin/autostartggss.lock

user@host:~$ ./ggss_monitor.sh check
ggssrunner is NOT running.

user@host:~$ ./ggss_monitor.sh check_start
```



Rys. 7.4. Panel WinCC OA monitorujący działanie systemu GGSS po ponownym uruchomieniu systemu (widoczny w lewym górnym rogu stan *STARTING*)

Dodatkowo użyty został skrypt pozwalający na monitorowanie zużycia zasobów pamięci przez aplikację (listing 7.3). Sposób użycia oraz fragment generowanego przez ten skrypt wyjścia przedstawia listing 7.4.

Listing 7.3. Skrypt *check_mem_ggssrunner.sh* służący do monitorowania pamięci używanej przez aplikację *ggssrunner*

```
#!/bin/bash
while true
do
    ps afux | egrep " ./ggssrunner" | awk -v date="$(date +"%Y.%m.%d %H:%M:%S")" ←
        '{print date, $5}'
    sleep 1m
done
```

Listing 7.4. Wywołanie oraz fragment wyjścia skryptu *check_mem_ggssrunner.sh* służącego do monitorowania pamięci używanej przez aplikację *ggssrunner*

```
user@host:~$ ./check_mem_ggssrunner.sh
2019.12.08 15:15:26 638800
2019.12.08 15:16:26 638800
2019.12.08 15:17:26 638800
2019.12.08 15:18:26 638800
2019.12.08 15:19:26 638800
2019.12.08 15:20:27 638800
2019.12.08 15:21:27 638800
2019.12.08 15:22:27 638800
```

W takim stanie system pozostawiony został na dłuższy (ponad 6 godzin) czas. Idea testu polegała na sprawdzeniu, czy przez ten czas działanie systemu pozostanie stabilne i nie pojawią się żadne błędy.

7.2. Wyniki testu

Test każdej z przygotowanych konfiguracji trwał **ponad 6 godzin**. Tabela 7.1 przedstawia rezultaty.

Podczas przeprowadzania testów wersji produkcyjnej zostały wykryte błędy w działaniu, co zostało zakomunikowane na panelu WinCC OA. Błędy te pojawiły się zarówno na poziomie całego systemu (Rys. 7.5), jak i pojedynczej słomki (Rys. 7.6). Pojawienie się błędów było skutkiem zastosowania w czasie kompilacji flagi optymalizacji *-O3*. Jest to domyślna flaga stosowana przez narzędzie *CMake* dla wersji produkcyjnej. Podczas kompilacji aplikacji *ggssrunner* w takiej konfiguracji widoczne były dwa ostrzeżenia, zatem błędy nie były dla autorów zaskoczeniem. Flagą *-O3* oznacza agresywną politykę optymalizacji, co często prowadzi do zmiany zachowania aplikacji. Została więc wprowadzona zmiana w sposobie kompilacji aplikacji w wer-

Konfiguracja (debug/release)	Sposób linkowania biblioteki Boost	Wygenerowane błędy	Zaalokowana pamięć
debug	statyczne	brak	stała
release (-O3)	statyczne	błąd zarejestrowany dla całego systemu GGSS	stała
release (-O2)	statyczne	brak	stała
debug	dynamiczne	brak	stała
release (-O3)	dynamiczne	błąd zarejestrowany dla konkretnej słomki	stała

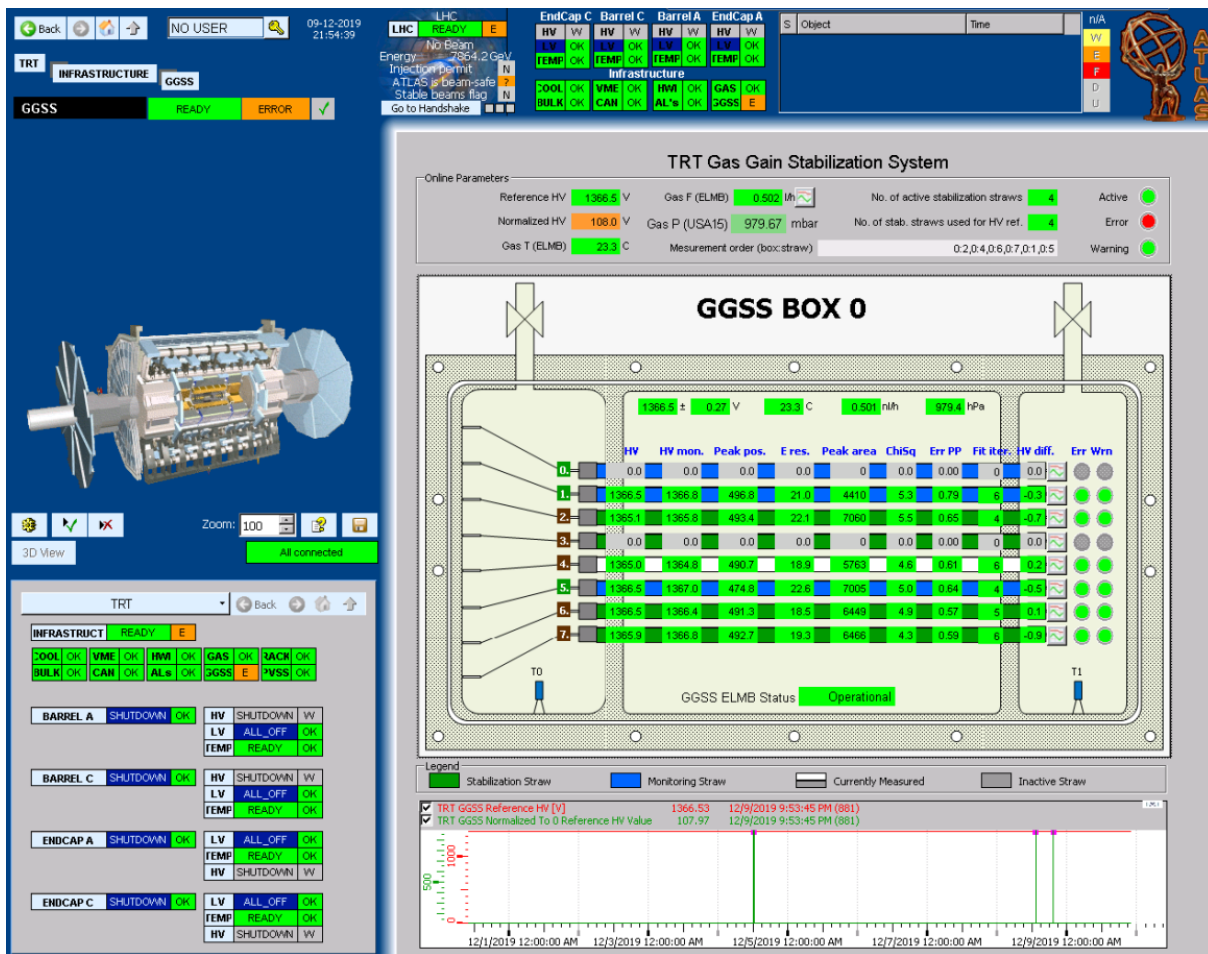
Tabela 7.1. Wyniki testów systemu GGSS po wprowadzonych zmianach

sji produkcyjnej - flaga ta została zastąpiona przez jej łagodniejszy i stabilniejszy odpowiednik *-O2*, powszechnie wykorzystywany do tworzenia wersji produkcyjnych oprogramowania. Testy wykonane z użyciem tej flagi nie wyprodukowały żadnych błędów.

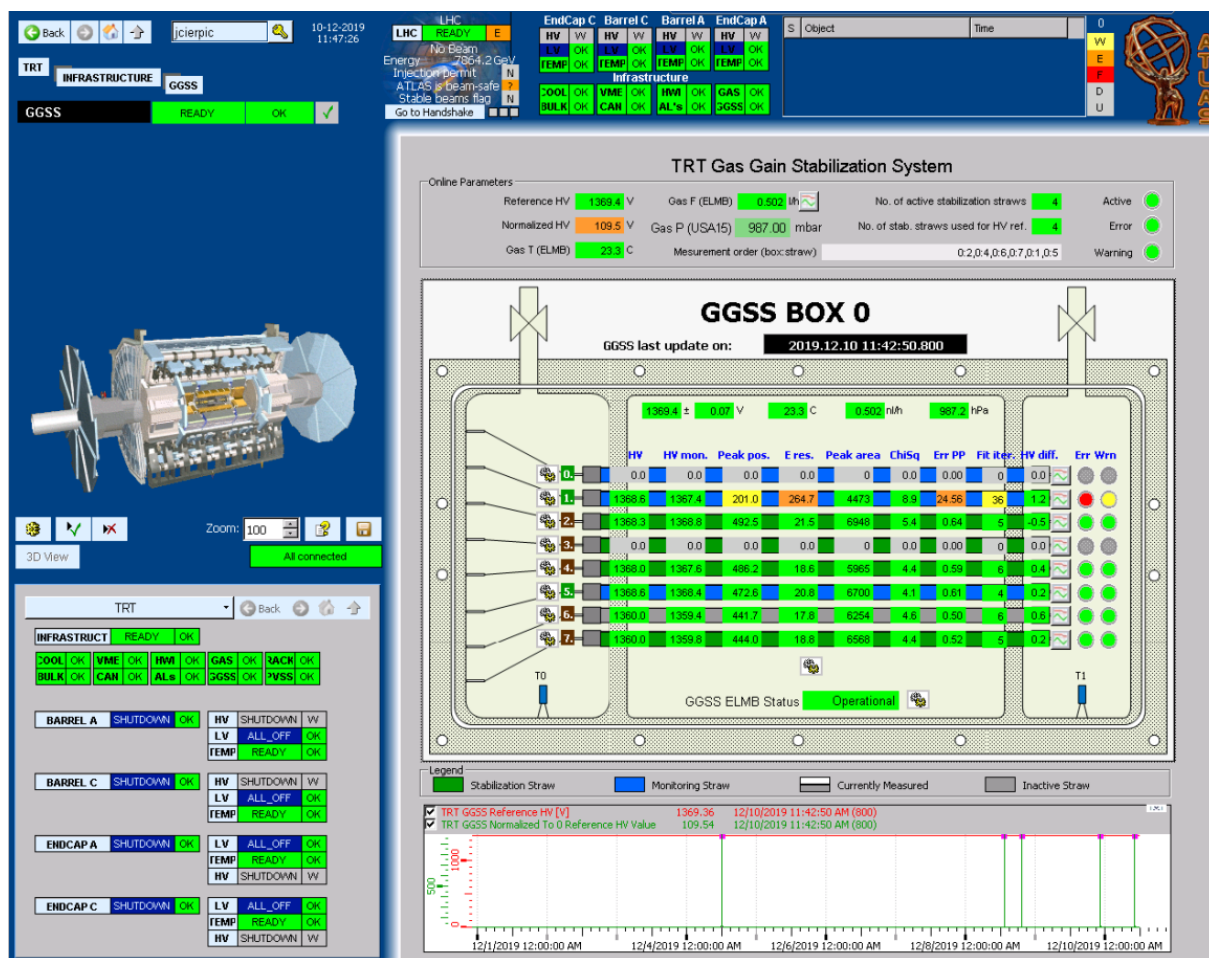
Testy wersji deweloperskiej (w obu konfiguracjach) odbyły się bez problemów - nie zostały wygenerowane żadne błędy ani ostrzeżenia.

W każdym z przypadków ilość zaalokowanej przez program pamięci pozostawała stała przez cały czas trwania testu. Oznacza to brak znaczących problemów z zarządzaniem pamięcią (takich jak wycieki pamięci).

Wyniki przeprowadzonych testów stanowią potwierdzenie poprawności wprowadzonych przez autorów zmian w systemie.



Rys. 7.5. Błąd w działaniu aplikacji *ggssrunner* na poziomie całego systemu widoczny w panelu WinCC OA (czerwone koło w prawym górnym rogu panelu GGSS)



Rys. 7.6. Błąd w działaniu aplikacji *ggssrunner* na poziomie pojedynczej słomki widoczny w panelu WinCC OA (czerwone koło obok drugiej słomki)

8. Dalsza ścieżka rozwoju projektu

Niniejszy rozdział opisuje sugestie autorów oraz już zaplanowane działania dotyczące dalszego rozwoju projektu GGSS, takie jak migracja do nowych standardów języka, zwiększenie jakości kodu czy rozszerzenie projektu o nowe interfejsy graficzne.

8.1. Wprowadzenie zautomatyzowanego systemu testowania projektu

Przygotowana przez autorów infrastruktura CI/CD pozwala na łatwe wprowadzenie do projektu dużej liczby zautomatyzowanych testów. Testy te powinny dotyczyć zarówno potencjalnych nowych komponentów projektu, jak i już istniejących, które podlegać będą modyfikacji. Takie działanie pozwala zwiększyć niezawodność tworzonego oprogramowania. Autorzy planują prowadzić dalsze prace z wykorzystaniem praktyki **Test Driven Development (TDD)** zakładającej tworzenie testów równoległe z oprogramowaniem. Stosującą podejście TDD programista najpierw przygotowuje odpowiedni test, a następnie implementuje funkcjonalność, którą ten test sprawdza. Po wykonaniu tej czynności przechodzi do poprawy jakości już istniejącego kodu, a następnie powtarza cały cykl. Prawdopodobnie do projektu wprowadzone zostanie również narzędzie **Coverity** pozwalające na przeprowadzenie statycznej analizy kodu.

8.2. Poprawa jakości kodu napisanego w języku C++

W swojej obecnej postaci projekt napisany jest w języku C++ z elementami standardu C++11. Planowana jest pełna migracja kodu do standardu C++11 lub, jeśli będzie to możliwe (rozwiązany zostanie problem ograniczeń związanych z infrastrukturą projektu), do standardu C++14 lub C++17. Migracja do nowego standardu oznaczała będzie również wyeliminowanie pewnych zewnętrznych zależności, które dostarczają funkcjonalności będących częścią nowych standardów języka. Dotyczy to przede wszystkim niektórych komponentów biblioteki Boost. Poza zmianami pozwalającymi na modernizację kodu planowane jest również ujednolicenie jego struktury oraz wprowadzenie konwencji dotyczących m.in. nazewnictwa zmiennych czy funkcji. Do tej pory autorom udało się wprowadzić tego typu zmiany w systemie budowania projektu.

8.3. Automatyzacja procesu publikowania produktu

Planowany jest dalszy rozwój przygotowanej przez autorów infrastruktury CI/CD. Poza wspomnianymi już wcześniej zautomatyzowanymi testami oprogramowania, wprowadzone może również zostać automatyczne wersjonowanie wydań projektu. Przydatne może się również okazać przygotowanie skryptu aktualizującego zdalne repozytoria projektu po wprowadzeniu zmian w kilku submodułach. Wymaga to wykonywania operacji *git add*, *git commit* oraz *git push* dla każdego komponentu zależnego od zmodyfikowanych submodułów i tak dalej aż do komponentu stanowiącego korzeń projektu. Wykonywanie tej czynności manualnie jest niewydatne, stąd też potrzeba przygotowania skryptu automatyzującego tą czynność.

8.4. Migracja do języka Python 3

Znaczna część skryptów w projekcie GGSS napisana została kilka lat temu w języku Python 2. Z uwagi na zapowiedziane zakończenie oficjalnego wsparcia dla tej wersji języka sugerowane jest przepisanie skryptów z użyciem Pythona 3. Migracja dotyczyć może również niektórych skryptów napisanych w języku Bash.

8.5. Przygotowanie dodatkowych interfejsów graficznych

Od początku pracy autorów nad projektem GGSS planowane było rozszerzenie jego funkcjonalności o nowe interfejsy graficzne pozwalające na wygodne przeglądanie gromadzonych danych. Istnieje wiele możliwych sposobów implementacji takich interfejsów. Jednym z nich jest zastosowanie popularnego **framework'u Qt**, przeznaczonego do pracy z językiem C++.

9. Podsumowanie oraz wnioski

Niniejszy rozdział stanowi podsumowanie prac przeprowadzonych przez autorów nad projektem GGSS. Zaprezentowane zostaną statystyki związane z projektem oraz wkładem autorów w jego rozwój, jak również zbiór wniosków, wyciągniętych przez autorów podczas prac nad systemem.

Zadaniem autorów było przeprowadzenie rozbudowy i aktualizacji oprogramowania Systemu Stabilizacji Wzmocnienia Gazowego detektora ATLAS TRT. Przeprowadzona została zmiana architektury projektu oraz systemu budującego, wprowadzone zostały nowe rozwiązania, takie jak infrastruktura CI/CD oraz system kontroli wersji **Git**. Pojawiło się wiele udoskonaleń względem poprzedniego rozwiązania, m.in. w logice działania pakietu RPM. Zadaniem autorów przyjęta podczas prac nad projektem strategia polegająca na dogłębnym analizowaniu postawionych przed nimi zadań była opłacalna. Poświęcenie czasu na dokładne przemyślenie rozwiązania pozwoliło autorom na znaczne przyspieszenie prac w fazie realizacji. Pozytywnym aspektem przeprowadzonych prac był również ich zespołowy charakter - pozwoliło to autorom na szybszą eliminację napotkanych problemów oraz dawało możliwość dzielenia się swoją wiedzą. Wkład obu autorów w projekt jest porównywalny, co obrazuje tabela 9.2. Wskazuje to na dobry przebieg współpracy między autorami. Praca nad systemem będącym częścią międzynarodowego projektu pozwoliła autorom również na zdobycie cennego, unikalnego doświadczenia. Zdaniem autorów wszystkie postawione przed nimi zadania zostały realizowane. Prace nad projektem będą kontynuowane w dalszym toku studiów podejmowanych przez autorów.

Tabela 9.1. Ilość linii w plikach projektu

Typ plików	Ilość linii zawarta we wszystkich plikach danego typu
C/C++	18 011
CMake	1 412
Python	602
Markdown	557
.gitlab-ci.yml	430
Shell/Bash	336
Dockerfile	56
RAZEM	21 404

Tabela 9.2. Statystyki wkładu autorów w projekt uzyskane za pomocą portalu GitLab

	Ilość commit'ów	Ilość wykonanych merge request'ów	Ilość zamkniętych issue
Jarosław Cierpich	150	11	14
Arkadiusz Kasprzak	212	6	12

A. Dodatki/Appendixes

Niniejsza część pracy zawiera trzy dodatki dotyczące procesu przygotowywania maszyny wirtualnej do pracy jako tzw. *runner* z narzędziem *GitLab CI/CD*, procesu dodawania nowych modułów do projektu GGSS oraz struktury projektu przed i po wprowadzeniu przez autorów zmian. Dwa z dodatków przygotowane zostały w języku angielskim - powodem tego jest międzynarodowy charakter zespołu pracującego nad detektorem TRT oraz fakt, że dodatki te stanowią część dokumentacji znajdującej się w repozytoriach projektu.

This part of the dissertation contains three appendixes that cover following topics: preparing the virtual machine to work as a GitLab CI/CD runner, adding new modules to the GGSS project using templates created by authors and comparing the project structure before and after changes. Two of the appendixes were written in English - that's due to the international nature of the TRT community as well as the fact that those appendixes are a part of documentation that can be found in project's repositories.

A.1. Porównanie początkowej i obecnej struktury projektu oraz kodu źródłowego

Niniejszy dodatek przedstawia porównanie początkowej oraz końcowej struktury projektu **GGSS**. Listingi A.1 oraz A.2 przedstawiają porównanie katalogów na poziomie całego projektu, gdzie tymi samymi kolorami oznaczone zostały odpowiadające sobie komponenty. Kolorem czarnym oznaczone zostały katalogi nieposiadające odpowiedników.

Listing A.3 przedstawia strukturę biblioteki **ggss-lib** po wprowadzonych zmianach. Na listingu A.4 przedstawiona została poprzednia struktura tejże biblioteki w celu ukazania różnic. Takie zmiany zostały wprowadzone w każdej z bibliotek wchodzących w skład projektu **GGSS**.

Rysunki A.1 i A.2 przedstawiają jedyne zmiany wprowadzone bezpośrednio w kodzie źródłowym aplikacji.

Listing A.1. Aktualna struktura katalogów

```

.
|-- ggss-dim-cs
|   |-- external-dim-lib
|   `-- ggss-misc
|-- ggss-driver
|   |-- external-n957-lib
|   |-- ggss-misc
|   `-- n957
|-- ggss-oper
|   |-- localdisk-scripts
|   `-- opt-scripts
|-- ggss-runner
|   `-- ggss-lib
|       |-- external-dim-lib
|       |-- ggss-hardware-libs
|       |   |-- caenhv-lib
|       |   |-- caenn1470-lib
|       |   |-- external-n957-lib
|       |   |-- ggss-util-libs
|       |   |   |-- ggss-misc
|       |   |   |-- handle-lib
|       |   |   |-- log-lib
|       |   |   |-- thread-lib
|       |   |   `-- utils-lib
|       |   |-- mca-lib
|       |   |-- ortecmcb-lib
|       |   `-- usbrm-lib
|       `-- ggss-software-libs
|           |-- daemon-lib
|           |-- fifo-lib
|           |-- fit-lib
|           |-- ggss-util-libs
|           `-- xml-lib
|-- ggss-spector
`-- mca-n957
    `-- ggss-hardware-libs

```

Listing A.2. Pierwotna struktura katalogów

```

.
|-- caen_n957
|   |-- _mca
|   |-- mcaLib
|   `-- OrtecMcbLib
|-- ggss-bin
|   |-- CaenHVLib
|   |-- CaenN1470Lib
|   |-- daemonLib
|   |-- _dimCS
|   |-- fifoLib
|   |-- FitLib
|   |-- _ggss
|   |-- ggssLib
|   |-- _ggsspector
|   |-- handleLib
|   |-- include
|   |-- lib
|   |-- logLib
|   |-- mcaLib
|   |-- misc
|   |   |-- config
|   |   |-- n957_old
|   |   `-- python
|   |-- OrtecMcbLib
|   |-- scripts
|   |-- ThreadLib
|   |-- usbrmLib
|   |-- utilsLib
|   `-- xmlLib
`-- ggss-driver
    `-- n957

```

Listing A.3. Nowa struktura biblioteki **ggss-lib**

```
.
|-- ggss-software-libs
|-- ggss-hardware-libs
|-- external-dim-lib
|-- CMakeLists.txt
|-- include
|   '-- ggssLib
|       |-- ChannelDataForDim.h
|       |-- ChannelData.h
|       |-- Channel.h
|       |-- ChannelParamsForDim.h
|       |-- DimGgssEventListener.h
|       |-- GgssDataForDim.h
|       |-- GgssEventListener.h
|       |-- GgssEventLoopRunner.h
|       |-- ggssExceptions.h
|       |-- ggss.h
|       |-- GgssParamsForDim.h
|       '-- ggssParams.h
|-- README.md
'-- src
    |-- Channel.cpp
    |-- ChannelData.cpp
    |-- ChannelDataForDim.cpp
    |-- ChannelParamsForDim.cpp
    |-- DimGgssEventListener.cpp
    |-- ggss.cpp
    |-- GgssDataForDim.cpp
    |-- GgssEventLoopRunner.cpp
    |-- ggssParams.cpp
    '-- GgssParamsForDim.cpp
```

Listing A.4. Pierwotna struktura biblioteki **ggss-lib**

```
ggssLib/
|-- Channel.cpp
|-- ChannelData.cpp
|-- ChannelDataForDim.cpp
|-- ChannelDataForDim.h
|-- ChannelData.h
|-- Channel.h
|-- ChannelParamsForDim.cpp
|-- ChannelParamsForDim.h
|-- CMakeLists.txt
|-- DimGgssEventListener.cpp
|-- DimGgssEventListener.h
|-- ggss.cpp
|-- GgssDataForDim.cpp
|-- GgssDataForDim.h
|-- GgssEventListener.h
|-- GgssEventLoopRunner.cpp
|-- GgssEventLoopRunner.h
|-- ggssExceptions.h
|-- ggss.h
|-- ggssParams.cpp
|-- GgssParamsForDim.cpp
|-- GgssParamsForDim.h
'-- ggssParams.h
```

```

▼ mca_lib/include/mcaLib/mca.h
@@ -21,10 +21,12 @@
21 21 #include <cstring>
22 22 #include <string>
23 23 #include <vector>
24 - #include <mca/N957Lib.h>
25 24 #include <boost/thread.hpp>
26 25 #include <boost/date_time.hpp>
27 26 #include "mcaException.h"
27 + extern "C"{
28 +     #include <mca/N957Lib.h>
29 + }
28 30
29 31 //////////////////////////////////////////////////
30 32 /*! \class      MCA
...

```

Rys. A.1. Dodanie konstrukcji `extern "C"` w ramach biblioteki `mca-lib`

```

▼ include/mca/include/N957Lib.h
@@ -37,9 +37,6 @@
37 37 */
38 38 /*@{*/
39 39
40 - #ifndef __cplusplus
41 - extern "C"{
42 - #endif
43 40 //////////////////////////////////////////////////
44 41 /*! \fn      N957_API N957_Init( short BdNum, int32_t *handle)
45 42 * \brief    Initialize a new board
...
@@ -825,9 +822,6 @@ N957_DLL_API N957_API N957_GetConfigROM( int32_t handle, N957_ConfigROM* config_
825 822 //////////////////////////////////////////////////
826 823 N957_DLL_API N957_API N957_FwUpgrade( int32_t handle, const N957_BYTE* data_buff, N957_UINT32 data_size,
N957FlashPageTypes page_type);
827 824
828 - #ifndef __cplusplus
829 - }
830 - #endif
831 825 /*@}*/
832 826
833 827 #endif
...

```

Rys. A.2. Usunięcie konstrukcji `extern "C"` w ramach zewnętrznej biblioteki `CAENN957`

A.2. Adding modules to the project using existing CMake templates

Using BuildDependencies.cmake template

To use this template:

- add path to the template to **CMAKE_MODULE_PATH** variable
- define **target_name** CMake variable - it should contain name of existing CMake project - built dependencies will be linked to it
- define **BUILD_OUTPUT_DIRECTORY** variable - it should contain path to directory where build output will be placed - is using this template with *BuildLibrary.cmake*, this variable is defined there - it is set to directory CMake was called from
- define **dependencies** variable - it should contain list of paths to dependencies that should be built. Omit - *lib* suffix - all GGSS libraries contain it, so it is added by the template itself
- define **dependency_prefix** variable - it should contain common part of all dependencies paths

Example usage:

```
# Set target_name variable
set(target_name "ggss")

# ...

# Add template path to CMAKE_MODULE_PATH variable
set(ggss_misc_path
    "${CMAKE_CURRENT_SOURCE_DIR}/ggss-software-libs/ggss-util-libs/ggss-misc")
list(APPEND CMAKE_MODULE_PATH "${ggss_misc_path}/cmake_templates")

# Include BuildLibrary.cmake template - it sets BUILD_OUTPUT_DIRECTORY variable
include(BuildLibrary)

# ...

# Set dependency_prefix variable - path to every dependency will start with it.
set(dependency_prefix "${CMAKE_CURRENT_SOURCE_DIR}")

# Define list of dependencies - omit '-lib' suffix
set(dependencies "ggss-hardware-libs/ortecmcb"
    "ggss-hardware-libs/caenhv"
    "ggss-hardware-libs/usbrm"
    "ggss-software-libs/fit"
    "ggss-software-libs/xml"
    "ggss-software-libs/fifo"
)

# Include template itself
include(BuildDependencies)
```

Using BuildLibrary.cmake template

To use this template:

- add path to the template to **CMAKE_MODULE_PATH** variable
- define **target_name** CMake variable - it should contain name of library that should be build (without any lib prefix/suffix)

Example usage:

```
# Set target_name variable
set(target_name "log")

# ...

## # Add template path to CMAKE_MODULE_PATH variable
set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/../ggss-misc/cmake_templates")

# Include template itself
include(BuildLibrary)

# ...
```

A.3. Preparing the virtual machine to work as a runner

Description

This file describes steps needed to setup your own Virtual Machine as a GitLab CI/CD runner. The manual assumes that you have already created your own Virtual Machine with Centos7 as a operating system.

Manual

Setting up Docker

This part of manual is based on official Docker documentation.

Remove old docker versions

```
sudo yum remove docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-engine
```

Install new docker engine and CLI using official repository

Install required packages

```
sudo yum install -y yum-utils \
    device-mapper-persistent-data \
    lvm2
```

Set up the repository

```
sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

Install Docker Engine and CLI

```
sudo yum install docker-ce docker-ce-cli containerd.io
```

Start Docker

```
sudo systemctl start docker
```

Verify if Docker works properly

```
sudo docker run hello-world
```

Register virtual machine as a GitLab CI/CD runner

This part of manual is based on official CERN GitLab documentation.

Download appropriate packages

```
curl -LJO \  
https://gitlab-runner-downloads.s3.amazonaws.com/latest/rpm/gitlab-runner_amd64.rpm
```

Install the packages

```
sudo yum install gitlab-runner_amd64.rpm
```

If you want to update runner packages use following command instead:

```
rpm -Uvh gitlab-runner_amd64.rpm
```

Download gitlab-runner binary file

```
sudo curl -L --output /usr/local/bin/gitlab-runner \  
gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64
```

Give proper permissions to the binary file

```
sudo chmod +x /usr/local/bin/gitlab-runner
```

Create a GitLab CI user

```
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
```

Install application and run as a service

```
sudo gitlab-runner install --user=gitlab-runner\  
  --working-directory=/home/gitlab-runner  
sudo gitlab-runner start
```

Register the runner

Register using gitlab-runner

```
sudo gitlab-runner register
```

Enter proper GitLab instance URL

```
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com )
https://gitlab.cern.ch
```

Enter token obtained from your GitLab group page > settings > CI/CD > Runners

```
Please enter the gitlab-ci token for this runner
<your_token_here>
```

Enter a description for the runner (this can be changed later)

```
Please enter the gitlab-ci description for this runner
[hostname] my-runner
```

Enter proper tags (this can be changed later)

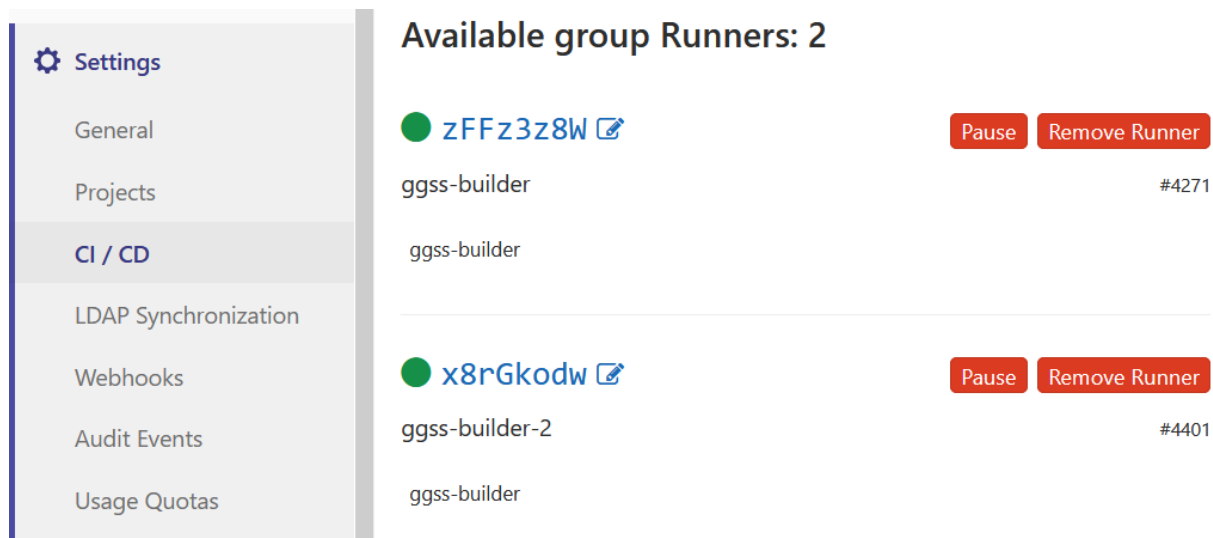
```
Please enter the gitlab-ci tags for this runner (comma separated):
ggss-builder
```

Enter runner executor

```
Please enter the executor: ssh, docker+machine, docker-ssh+machine,\
kubernetes, docker, parallels, virtualbox, docker-ssh, shell:
docker
```

Enter default image (if not defined in .gitlab-ci.yml)

```
Please enter the Docker image (eg. ruby:2.1):
cern/cc7-base:latest
```



Rys. A.3. Panel przedstawiający dodane **GitLab CI/CD Runner** w ramach projektu.

Dodatek A.3 opisuje procedurę dodawania nowej maszyny jako **GitLab CI/CD Runner**, czyli jednostka odpowiedzialna za wykonywanie zadań w ramach funkcjonalności **GitLab CI/CD**. Instrukcja zakłada, że maszyna posiada zainstalowany system operacyjny **Linux** w dystrybucji **Centos7**. Wynikiem wykonanie wszystkich czynności powinno być dodanie maszyny do panelu **CI/CD>Runners** dla wybranej grupy, co ukazuje Rysunek A.3. Instrukcja została napisana w oparciu o dokumentację **Docker**[45] oraz dokumentację **GitLab**[46].

Bibliografia

- [1] Wikipedia The Free Encyclopedia. *ATLAS experiment*. URL: https://en.wikipedia.org/wiki/ATLAS_experiment (term. wiz. 2019-12-07).
- [2] M. Bochenek, T. Bołd, K. Ciba, W. Dąbrowski, M. Deptuch, M. Dwużnik, T. Fiutowski, I. Grabowska-Bołd, M. Idzik, K. Jeleń, D. Kisielewska, S. Koperny, T. Z. Kowalski, S. Kulis, B. Mindur, B. Muryn, A. Obląkowska-Mucha, J. Pieron, K. Półtorak, B. Prochal, L. Suszycki, T. Szumlak, K. Świentek, B. Toczek i T. Tora. „Budowa aparatury detekcyjnej i przygotowanie programu fizycznego przyszłych eksperymentów fizyki cząstek (ATLAS i LHCb na akceleratorze LHC i Super LHC oraz eksperymentu na akceleratorze liniowym ILC).” W: *Akademia Górniczo-Hutnicza im. S. Staszica. Wydział Fizyki i Informatyki Stosowanej. Raport Roczny 2008*. (2008).
- [3] Bjarne Stroustrup. *Język C++, Kompendium Wiedzy, Wydanie IV (The C++ Programming Language, 4th Edition)*. ul. Kościuszki 1c, 44-100 Gliwice: HELION S.A., 2014, s. 35–53.
- [4] Wikipedia The Free Encyclopedia. *C++14*. URL: <https://en.wikipedia.org/wiki/C%2B%2B14> (term. wiz. 2019-12-07).
- [5] Bartłomiej Filipek. *C++ 17 Features*. URL: <https://www.bfilipek.com/2017/01/cpp17features.html> (term. wiz. 2019-12-07).
- [6] *Dokumentacja biblioteki Boost w wersji 1.57.0*. URL: https://www.boost.org/doc/libs/1_57_0/?view=categorized (term. wiz. 2019-12-07).
- [7] Milan Stevanovic. *C and C++ Compiling. An engineering guide to compiling, linking, and libraries using C and C++*. Apress, 2014, s. 53–115.
- [8] Megha Mohan. *All about Static Libraries in C*. URL: <https://medium.com/@meghamohan/all-about-static-libraries-in-c-cea57990c495> (term. wiz. 2019-12-09).
- [9] The Linux Documentation Project. *Shared Libraries*. URL: <http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html> (term. wiz. 2019-12-09).
- [10] Michael Kerrisk. *The Linux Programming Interface*. no starch press, 2010, s. 845–847.

- [11] Jimmy Thong. *Shared (dynamic) libraries in the C programming language*. URL: <https://medium.com/meatandmachines/shared-dynamic-libraries-in-the-c-programming-language-8c2c03311756> (term. wiz. 2019-12-09).
- [12] Peter Seebach. *Dissecting shared libraries*. URL: <https://www.ibm.com/developerworks/library/l-shlibs/> (term. wiz. 2019-12-09).
- [13] Artur Gramacki. *Instrukcja do zajęć laboratoryjnych. Język ANSI C (w systemie LINUX)*. URL: http://staff.uz.zgora.pl/agramack/files/Linux/ANSI_C_Linux_Lab5.pdf (term. wiz. 2019-12-09).
- [14] Kitware. *About CMake*. URL: <https://cmake.org/overview/> (term. wiz. 2019-12-07).
- [15] Sandy McKenzie (KitwareBlog). *CMake Ups Support for Popular Programming Languages in Version 3.8*. URL: <https://blog.kitware.com/cmake-ups-support-for-popular-programming-languages-in-version-3-8/> (term. wiz. 2019-12-07).
- [16] Pablo Arias. *It's Time To Do CMake Right*. URL: <https://pabloariasal.github.io/2018/02/19/its-time-to-do-cmake-right/> (term. wiz. 2019-12-07).
- [17] Mark Lutz. *Python. Wprowadzenie. Wydanie IV (Learning Python, Fourth Edition by Mark Lutz)*. ul. Kościuszki 1c, 44-100 Gliwice: HELION S.A., 2011, s. 49–65.
- [18] Laurence Bradford. *What Should I Learn As A Beginner: Python 2 Or Python 3?* URL: <https://learntocodewith.me/programming/python/python-2-vs-python-3/> (term. wiz. 2019-12-07).
- [19] Team Anaconda. *End of Life (EOL) for Python 2.7 is coming. Are you ready?* URL: <https://www.anaconda.com/end-of-life-eol-for-python-2-7-is-coming-are-you-ready/> (term. wiz. 2019-12-07).
- [20] *Beautiful Soup Documentation*. URL: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/> (term. wiz. 2019-12-22).
- [21] *Bash Reference Manual*. URL: http://www.gnu.org/software/bash/manual/bash.html#What-is-Bash_003f (term. wiz. 2019-12-08).
- [22] Anand Abhishek Singh. *File states in Git*. URL: <https://anandabhisheksingh.me/file-states-git/> (term. wiz. 2019-12-08).
- [23] Wikipedia The Free Encyclopedia. *Package manager*. URL: https://en.wikipedia.org/wiki/Package_manager (term. wiz. 2019-12-08).
- [24] opensource.com. *What is virtualization?* URL: <https://opensource.com/resources/virtualization> (term. wiz. 2019-12-08).
- [25] Vineet Chaturvedi. *What Is Docker And Docker Container ? A Deep Dive Into Docker !* URL: <https://www.edureka.co/blog/what-is-docker-container> (term. wiz. 2019-12-11).

- [26] EuroLinux Sp. z o.o. *Podstawy konteneryzacji – czyli jak w Linuksie działają kontenery*. URL: <https://pl.euro-linux.com/blog/podstawy-konteneryzacji/> (term. wiz. 2019-12-11).
- [27] evgenyl. *Can Windows Containers be hosted on linux?* URL: <https://stackoverflow.com/questions/42158596/can-windows-containers-be-hosted-on-linux> (term. wiz. 2019-12-11).
- [28] Doug Chamberlain. *Containers vs. Virtual Machines (VMs): What's the Difference?* URL: <https://blog.netapp.com/blogs/containers-vs-vms/> (term. wiz. 2019-12-11).
- [29] Przemysław Plutecki. *Aktualizacja oprogramowania oraz sprzętu elektronicznego dla Systemu Stabilizacji Wzmocnienia Gazowego*. 2015.
- [30] Paweł Zadrozniak. *Aktualizacja sprzętu elektronicznego dla Systemu Stabilizacji Wzmocnienia Gazowego*. 2015.
- [31] GCC Team. *Status of Experimental C++11 Support in GCC 4.8*. URL: https://gcc.gnu.org/gcc-4.8/cxx0x_status.html (term. wiz. 2019-12-07).
- [32] *What's new in in CMake*. URL: <https://cliutils.gitlab.io/modern-cmake/chapters/intro/newcmake.html> (term. wiz. 2019-12-07).
- [33] GitLab. *Permissions*. URL: <https://docs.gitlab.com/ee/user/permissions.html> (term. wiz. 2019-12-11).
- [34] Brian Heim. *CMake stuff i wish i knew earlier*. URL: <http://www.brianlheim.com/2018/04/09/cmake-cheat-sheet.html#build-types> (term. wiz. 2019-12-15).
- [35] C. Gaspar. *DIM Distributed Information Management System*. URL: <https://dim.web.cern.ch/> (term. wiz. 2019-12-15).
- [36] *ExternalProject*. URL: <https://cmake.org/cmake/help/latest/module/ExternalProject.html> (term. wiz. 2019-12-15).
- [37] Kamil Porembiński. *Continuous Integration, Continuous Delivery oraz Continuous Deployment*. URL: <https://thecamels.org/pl/continuous-integration-continuous-delivery-oraz-continuous-deployment/> (term. wiz. 2019-12-10).
- [38] Wikipedia The Free Encyclopedia. *Jenkins (software)*. URL: [https://en.wikipedia.org/wiki/Jenkins_\(software\)](https://en.wikipedia.org/wiki/Jenkins_(software)) (term. wiz. 2019-12-10).
- [39] GitLab. *Auto DevOps*. URL: <https://docs.gitlab.com/ee/topics/autodevops/> (term. wiz. 2019-12-10).
- [40] DevKR. *Continuous Integration z GitLab CI*. URL: <https://devkr.pl/2018/01/15/continuous-integration-gitlab-ci/> (term. wiz. 2019-12-10).
- [41] GitLab. *GitLab Continuous Integration (CI) and Continuous Delivery (CD)*. URL: <https://about.gitlab.com/product/continuous-integration/> (term. wiz. 2019-12-10).
- [42] Wikipedia The Free Encyclopedia. *YAML*. URL: <https://en.wikipedia.org/wiki/YAML> (term. wiz. 2019-12-10).

- [43] Martin Streicher. *Upgrading and uninstalling software*. URL: <https://developer.ibm.com/articles/l-rpm2/> (term. wiz. 2019-12-15).
- [44] John Gruber. *Markdown*. URL: <https://daringfireball.net/projects/markdown/> (term. wiz. 2019-12-11).
- [45] Docker Inc. *Get Docker Engine - Community for CentOS*. URL: <https://docs.docker.com/install/linux/docker-ce/centos/> (term. wiz. 2019-12-15).
- [46] GitLab. *Registering Runners*. URL: <https://docs.gitlab.com/runner/register/> (term. wiz. 2019-12-15).