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 Uczelni 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

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 subdetektorów [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 dektektora 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 wchodzą zarówno urządzenia takie jak multiplekser 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 utworzenie 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 ogólnoprzyjęty 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 projektu w formie krótkich instrukcji czy 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. Sprawiło 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 umożliwiających 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 istniejących 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 starym 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 starym standardzie
// 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_opers.c
    `-- complex_opers.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

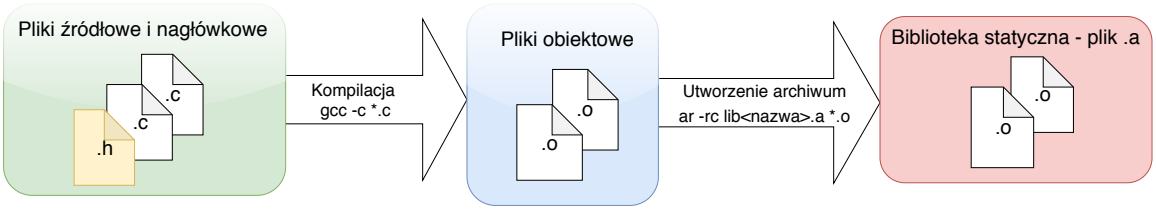
```
user@host:~/complex$ gcc -c *.c
user@host:~/complex$ ls
complex_number.h  complex_opers.c  complex_opers.h  complex_opers.o
```

– 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_opers.c  complex_opers.h  complex_opers.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 wszyskie pliki obiektowe wchodzące w skład danej biblioteki. Istnieje również możliwość wylistowania 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).

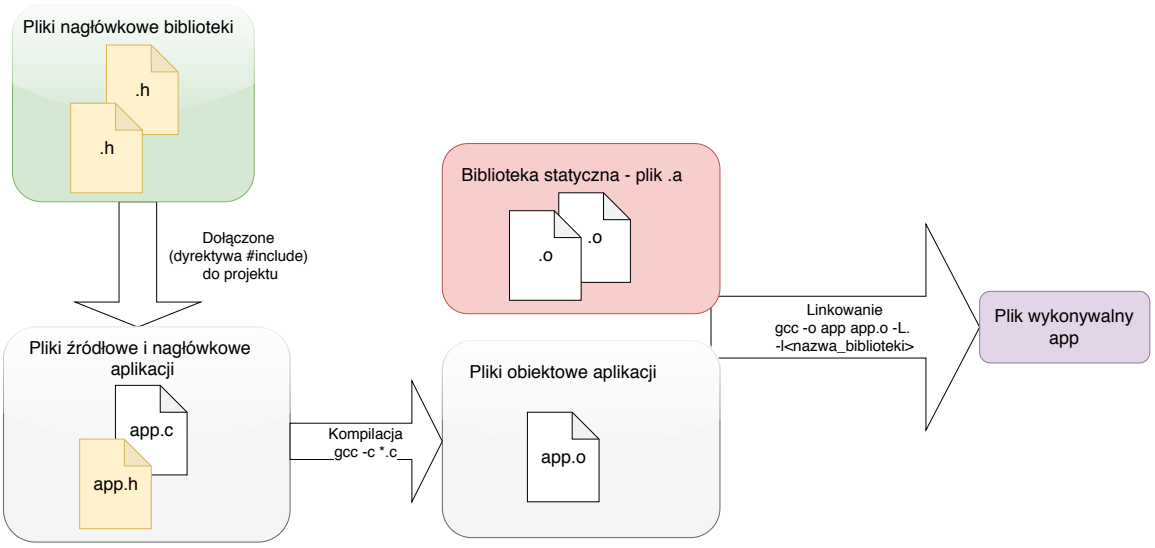
Listing 2.6. Użycie poleceń *ar -t* oraz *nm* na bibliotece statycznej.

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

complex_opers.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_opers.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 źródlowych programu do postaci plików obiektowych za pomoca *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 bibliteki (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 -lcoml
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

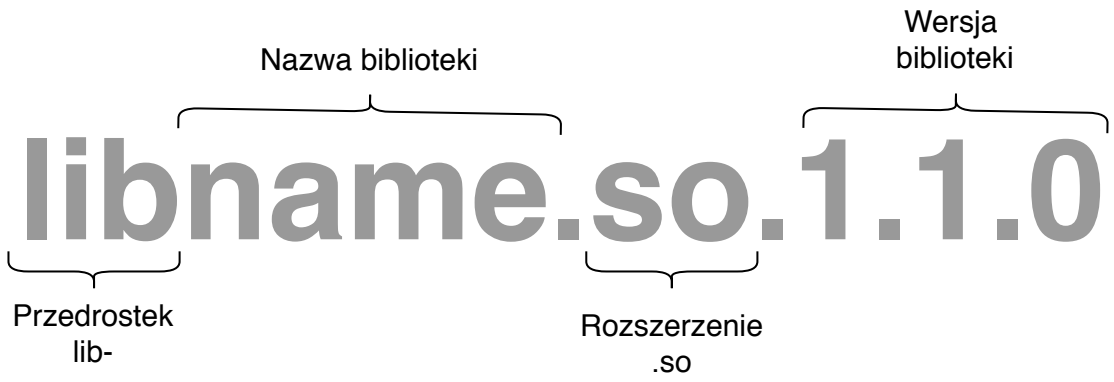
Działanie bibliotek współdzielonych jest bardziej skomplikowane. Z uwagi na fakt, iż w projekcie GGSS biblioteki współdzielone są używane jedynie w postaci zewnętrznych zależności, przedstawiony opis dotyczył będzie jedynie podstaw tworzenia i działania tych bibliotek. Główna różnica w stosunku do bibliotek 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 rekompilacji 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 poszcze-

gólne 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 miedzy 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 pozbiór nazwy przedstawionej na Rys. 2.3, bez dwóch ostatnich numerów wersji (przykład: *libname.so.1*). Nazwa *soname* stanowi zwyklę 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_opers.c  complex_opers.h  complex_opers.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 (soname)*. 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

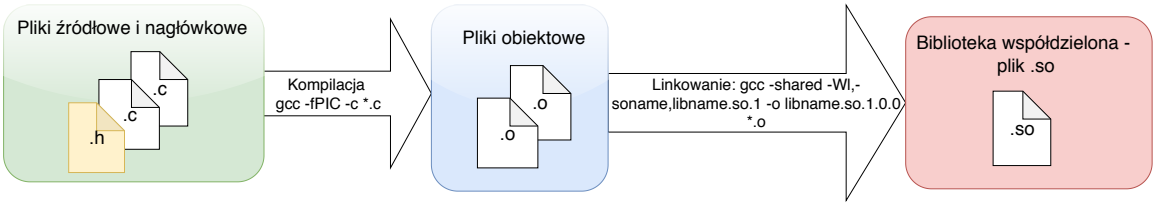
```
user@host:~/complex$ gcc -shared -Wl,-soname,libcoml.so.1 -o ↩
    libcoml.so.1.0.0 *.o
user@host:~/complex$ ls
complex_number.h  complex_opers.c  complex_opers.h  complex_opers.o  ↩
    libcoml.so.1.0.0
user@host:~/complex$ objdump -p libcoml.so.1.0.0 | grep SONAME
  SONAME               libcoml.so.1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

– 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 dowiązania symbolicznego wskazującego na bibliotekę za pomocą programu *ldcongfig*

```
user@host:~/complex$ ldconfig -n .
user@host:~/complex$ ls
complex_number.h  complex_opers.c  complex_opers.h  complex_opers.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 możliwośc 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 porażką.

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 (0x00007fffc16b6000)
        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 na-

pisanej 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 True)

# 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 narzedziem, 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, bardziej ustrukturyzowanym [16] podejściu, 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 modułem 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

## Prosty skrypt przetwarzający argumenty podane
## przy jego uruchomieniu przez użytkownika

## Definicja funkcji w języku Python
def parse_command_line_arguments():

    ## Obiekt przetwarzający argumenty (parser)
    parser = argparse.ArgumentParser()

    ## Argumenty wzajemnie się wykluczające
    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__":
    ## Wywołanie funkcji
    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* 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 [20]. 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 zawartośc 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.

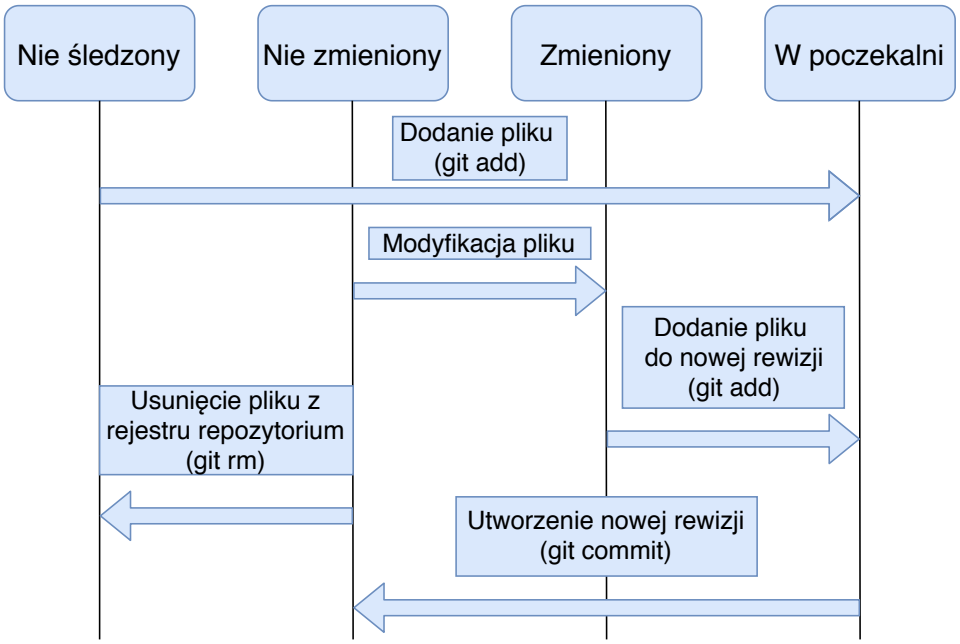
Listing 2.22. Inicjalizacja repozytorium git

```
user@host:/ścieżka/do/projektu$ git init
Initialized empty Git repository in /ściezka/do/projektu
user@host:/ściezka/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:/ściezka/do/projektu# git add plik1 folder1
user@host:/ściezka/do/projektu# 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 [21]

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@host.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@host.localdomain>
Date:   Wed Dec 4 17:30:10 2019 +0100

    Druga rewizja

commit 1d2445e961beb25940dffa9d73963f887ee553ad
Author: user <user@host.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 Gitowych 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 naszej wersji

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

2.7. Manager pakietów - RPM

Menadżer pakietów jest zbiorem oprogramowania, które w sposób automatyczny zarządza instalacją, aktualizacją, konfiguracją oraz usuwaniem programów komputerowych [22]. 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 manadżerom wszystko jest wykonywane automatycznie. Jeżeli w trakcie procedur nie wystąpi żaden problem, to pakiet, którego zleciliśmy instalacje, 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 manadżerem 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 pakiecie, proces ten odbuwa 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/rmmod 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 instacji zachowują się tak, jakby znajdowały się na **odseparowanym komputerze** z własnym, dedykowanym systemem operacyjnym, bibliotekami oraz innym oprogramowaniem. Dużym plusem jest pełna separacja instancji uruchomionych na systemie gospodarza. Jedna instancja **nie jest afektowana** przez procesy innej instancji [23].

Proces wirtualizacji odbywa się za pomocą oprogramowania, które nazywa się hipernadzorcą (ang. hipervisor). Odpowiada on za zapewnienie środowiska, które pozwoli na uruchomienie maszyny wirtualnej. Wyróżnanie 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 utworzone [referencej edureka]. Kontenery są tworzone na podstawie obrazu, czyli wzorcowego środowiska, które zostało zamrożone w celu późniejszego odwtorzenia. 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 **??** ukazuje przykładowy Dockerfile, którego użycie, za pomocą odpowiedniej komendy Dockera, spowoduje utworzenie obrazu bazującego na dystrybucji centos z zainstalowanym kompilatorem języka c++.

Listing 2.28. Przykładowy Dockerfile

3. Stan początkowy projektu

3.1. Architektura

3.2. Budowanie

Niniejsza część pracy zawiera opis pierwotnego sposobu budowania aplikacji, wraz z zastosowanymi rozwiązaniami technologicznymi (struktura i zawartość plików CMake) oraz listą potencjalnych ograniczeń wynikających z dotychczasowego podejścia do budowania.

Struktura plików CMake

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 głownym projektu oraz pomniejsze pliki dla każdego z modułów. Rysunek przedstawia w uproszczeniu pierwotną strukturę projektu, z wyszczególnieniem plików odpowiedzialnych za jego budowanie.

TODO: tutaj dac jakis fajny rysuneczek

Obsługa bibliotek zewnętrznych

Ograniczenia pierwotnego systemu budowania

Pierwotna wersja projektu narzuca daleko idące ograniczenia na sposób jego budowania. Najważniejszym z nich jest brak bezpośredniej możliwości zbudowania pojedynczych komponentów projektu. Listing 3.1 przedstawia fragment oryginalnego pliku *CMakeLists.txt* znajdującego się w katalogu głównym projektu. **TUTAJ REF DO PRACY PLUTECKIEGO** Plik ten pozwala na zbudowanie trzech aplikacji wchodzących w skład oprogramowania projektu GGSS: *ggssrunner*, *dimCS* oraz opcjonalnie *ggsspector*. Jest to jedyny plik w całym projekcie zawierający wszystkie informacje konieczne do zbudowania wymienionych aplikacji - tzn. posiadający listę bibliotek, od których aplikacje te są zależne. Oznacza to, że niemożliwe jest zbudowanie aplikacji *ggssrunner* jedynie za pomocą dedykowanego jej pliku *CMakeLists.txt*. Zatem pomimo, iż struktura projektu jest zmodularyzowana jeśli chodzi o architekturę (oprogramowanie zostało

podzielone na biblioteki), to niemożliwe jest (w prosty sposób, za pomocą dostarczonych plików *CMakeLists.txt*) zbudowanie pojedynczych modułów projektu.

Listing 3.1. Fragment oryginalnego pliku CMakeLists.txt znajdującego się w katalogu głównym pierwotnej wersji projektu

```
# array with used libraries
set(PROJECTS
        logLib
        xmlLib
        utilsLib
        handleLib
        ThreadLib
        fifoLib
        FitLib
        OrtecMcbLib
        CaenHVLib
        ggssLib
        usbrmLib
        CaenN1470Lib
        mcaLib
        daemonLib
        )

foreach (singleproject ${PROJECTS})
        parse_directory(${singleproject})
endforeach(singleproject)

# executables
add_subdirectory (_ggss) # ggssrunner binary
add_subdirectory (_dimCS) #dimCS binary
if(BUILD_GGSSPECTOR)
    add_subdirectory (_ggsspector) #ggsspector binary
endif()
```

Budowanie projektu za pomocą pliku, którego fragment przedstawia listing 3.1 opiera się na liście zależności przechowywanej w zmiennej *PROJECTS*. Umożliwia to stosunkowo łatwe rozszerzanie projektu o nowe biblioteki - wystarczy dopisać nazwę katalogu z biblioteką do listy zależności. Wadą tego rozwiązania jest natomiast brak możliwości wywnioskowania zależności zachodzących w projekcie. Listing 3.2 przedstawia plik *CMakeLists.txt* służący do budowania aplikacji *ggssrunner*. Na podstawie tych dwóch plików można jedynie wywnioskować, że aplikacja *ggssrunner* jest zależna od wszystkich bibliotek, których nazwy znaleźć można w zmiennej *PROJECTS*. Nie ma natomiast możliwości identyfikacji zależności między samymi bibliotekami. Takie podejście utrudnia zrozumienie struktury projektu, co bezpośrednio prowadzi do problemów z jego rozwojem.

Listing 3.2. Oryginalny plik CMakeLists.txt służacy budowania aplikacji ggssrunner.

```
project (_ggss)
add_executable (ggssrunner main)
target_link_libraries (ggssrunner ${PROJECTS})
install(TARGETS ggssrunner RUNTIME DESTINATION bin)
```

3.3. Dostarczanie i uruchamianie

3.4. Kontrola wersji

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 manuskryptem. Z tego powodu niniejszy rodział podzielony został na dwie części - pierwsza z nich opisuje finalną wersję projektu, natomiast druga - wersję po zakończeniu prac w ramach projektu inżynierkiego.

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, ogólnoprzyjętym 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 duża 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 inżynierkiego (co wynika z ograniczonego czasu), wybrany został następujący podzbiór wymagań:

- przygotowanie środowiska umożliwijącego 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

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.

Wersja kompilatora języka C++
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 [24]. Wersja ta nie wspiera niestety nowszych wydań języka (C++14/17).

Wersja interpretera języka Python
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 bibliteki i frameworki (np. *PyTest* - wykorzystywany do przeprowadzania testów oprogramowania) nie są dostępne dla obu wersji

Pythona. Taka sytuacja wymusza więc wykorzystanie narzędzia *virtualenv* w celu ich instalacji w odizolowanym środowisku, nie mającym wpływu na infrastrukturę CERN-u.

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 ogólnoprzyjętych dziś praktyk, jak np. określenie zakresu wersji narzedzia CMake, w którym powinna mieścić się używana wersja, by projekt można było bez problemu zbudować, czy wsparcie dla instrukcji *target_link_directories* [25].

5.4. Związek projektu z wersją jądra systemu

6. Wykonane prace

6.1. Wykorzystanie funkcjonalności portalu Gitlab wspierających zarządzanie projektem

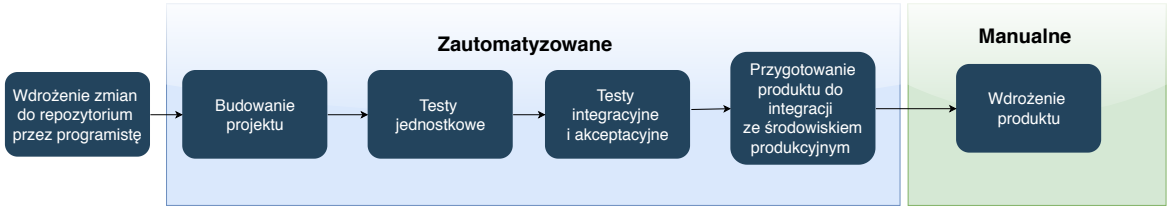
6.2. Migracja projektu do systemu kontroli wersji Git i zmiany w architekturze

6.3. Zmiana sposobu budowania aplikacji

6.4. Zastosowanie podejścia CI/CD

Ważną częścia 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** (*Continous 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** (*Continous 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** [26], które jednak nie znalazło zastosowania w projekcie GGSS). Podejście **CI/CD** porównuje się czasem do działania linii produkcyjnej. Rysunek 6.1 obrazuje schematycznie jego działanie.

Rys. 6.1. Przykładowy schemat działana podejścia *Continous Integration / Continous 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. Prawodopodobnie najpopularniejszym z nich jest **Jenkins**. Jest to darmowe [27] oprogramowanie do automatyzacji, charakteryzujące się dużą konfigurowalnością. 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 związanej 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 zastosowania 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 [28]. 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.

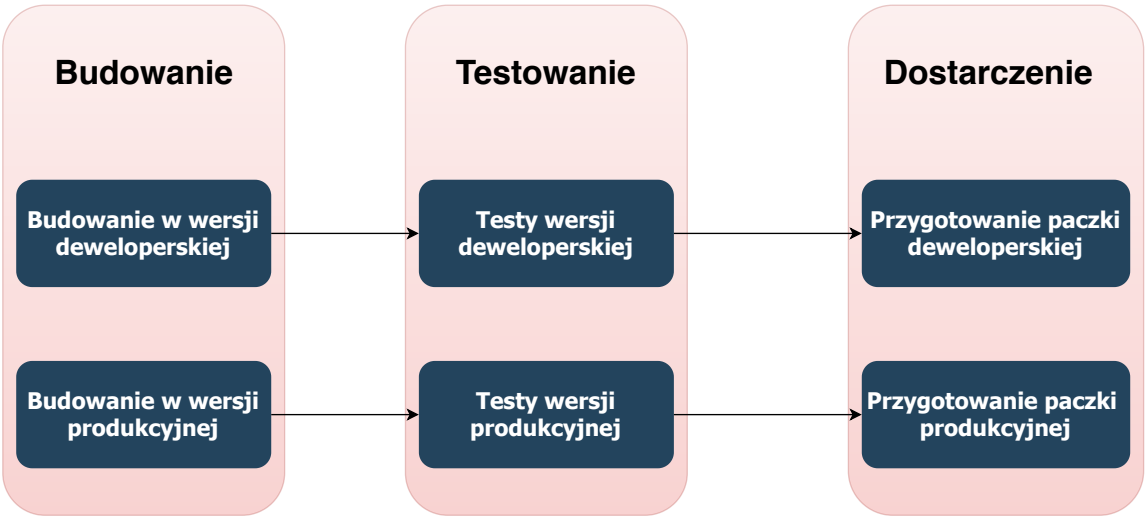
6.4.2. Opis działania GitLab CI/CD

Przed przystąpieniem do opisu sposobu zastosowania narzędzia GitLab CI/CD w projekcie GGSS przedstawiony zostanie sposób jego działania oraz najważniejsze pojęcia z nim zmiązane. 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. 6.2.

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 [29]. Przebieg całego procesu koordynowany jest przez **GitLab Server** [30].

Rys. 6.2. 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 sie zwykle od pokazanej.



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* [31]), ze względu na swoją czytelność, pozwala na stosunkowo szybkie przygotowanie funkcjonalnego systemu. Listing 6.1 przedstawia przykład prostego pliku w tym formacie konfigurującego 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 sie opis zadania (*dim_software_test*) należącego do zdefiniowanego etapu. Podczas definicji zadań możliwe jest wyspecyfikowanie mających się wykonać komend (klucz *script*).

Listing 6.1. 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 przygotowane zostało środowisko zapewniające ciągłą integrację i dostarczanie. Pozwala ono na automatyzację procesu budowania i testowania poszczególnych komponentów projektu oraz, dzięki mechanizmowi artekfaktów, przygotowuje gotowe paczki z najważniejszymi aplikacjami w projekcie (m.in. *ggssrunner*). Każde repozytorium zawierające kompilowalny komponent projektu 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 6.2 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:

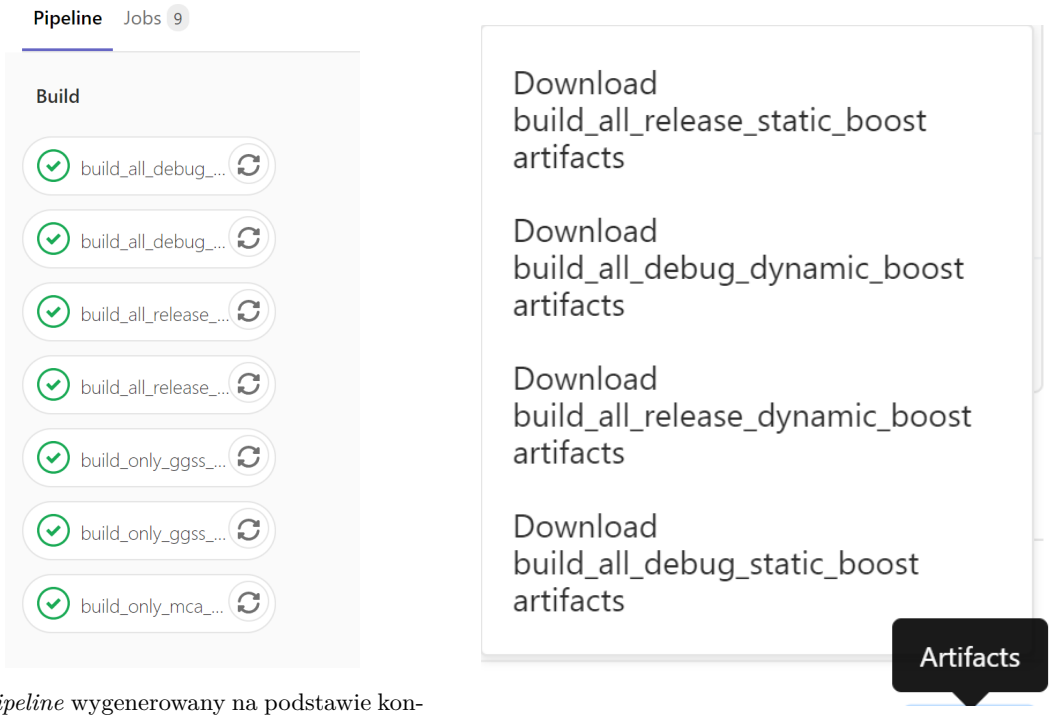
- **build_all_debug_static_boost** - zbudowanie wszystkich trzech wymienionych wyżej aplikacji w wersji deweloperskiej (debug) ze statycznie dołączaną biblioteką Boost oraz przygotowanie artefaktu zawierającego wynikowe pliki wykonywalne.
- **build_all_debug_dynamic_boost** - zbudowanie wszystkich trzech 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 6.2. 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 6.2). Do przeprowadzenia budowania używany jest specjalnie w tym celu przygotowany przez autorów obraz, oparty na oficjalnych obrazach dostarczonych przez CERN. Rys. 6.3 przedstawia wynikowy *pipeline* oraz dostępnie do pobrania artefakty.



(a) *Pipeline* wygenerowany na podstawie konfiguracji zamieszczonej w pliku *.yml* (listing 6.2)

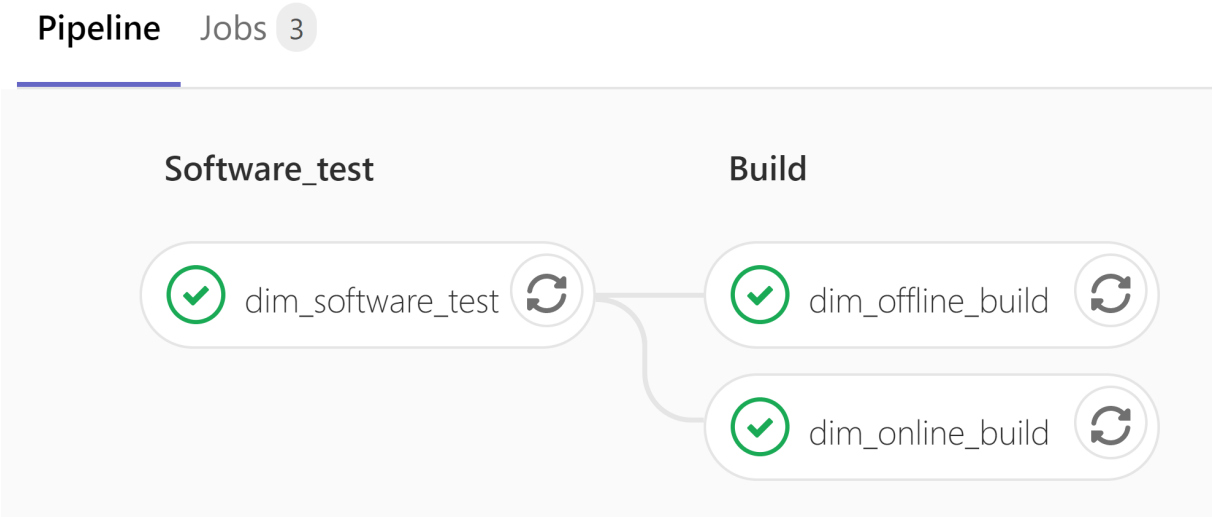
(b) Możliwe do pobrania artefakty

Rys. 6.3. 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.

6.4) 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. 6.4. *Pipeline* wygenerowany na potrzeby repozytorium *external-dim-lib*



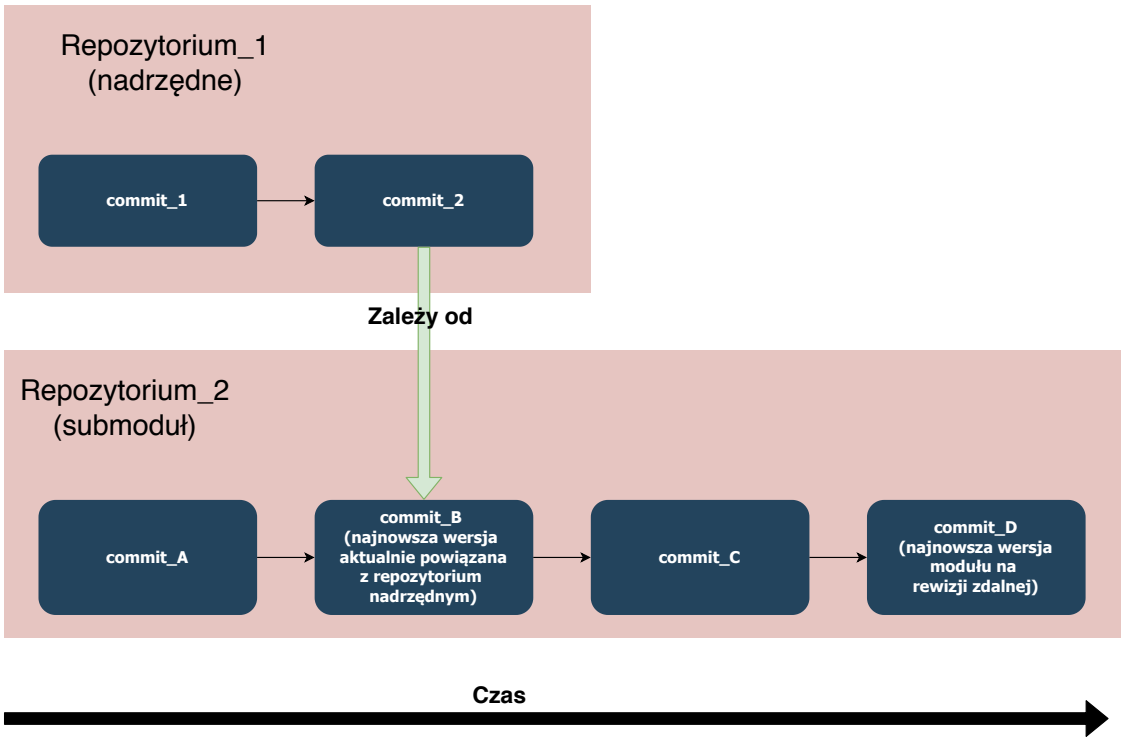
Podobnego typu prace zostały wykonane dla większości pozostałych repozytoriów. Z uwagi na ich powtarzalny charakter nie zostaną one jednak omówione w niniejszym manuskrypcie.

W przedstawionym przykładzie dotyczącym aplikacji *ggssrunner* (listing 6.2) 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 6.3). Oba wymienione podejścia mają swoje wady i zalety. Rys. 6.5 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 przypadków do tego własnie rozwiązania, jednak z uwagi na zalety pierwszej z wymienionych opcji, jest ona wciąż używana w niektórych repozytoriach (np. *ggss-all*).

Listing 6.3. Fragment pliku *.gitlab-ci.yml* znajdującego się w repozytorium *ggss-software-libs* pobierający submoduły projektu w wersji aktualnie powiązanej z repozytorium narzędnym

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

variables:
  GIT_SUBMODULE_STRATEGY: recursive

# Dalsza część pliku
```

Rys. 6.5. Uproszczona (ograniczona do jednej gałęzi na repozytorium) ilustracja problemu dotyczacego 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 6.4. Zawartość tego szablonu, zmodyfikowana zgodnie ze specyfiką danego repozytorium, posłużyła autorom do napisania większości plików *.yml*, które znalazły się w projekcie.

Listing 6.4. 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

6.6. Maszyna wirtualna oraz konteneryzacja - Docker

6.7. Pomniejsze prace

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

6.7.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake

6.8. Dokumentacja projektu

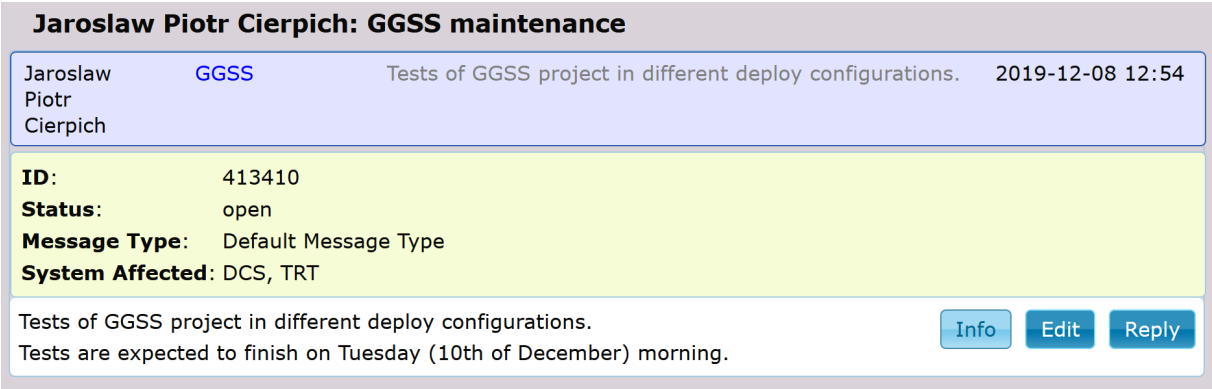
7. Testy nowej wersji oprogramowania

Niniejszy rozdział zawiera opis testów systemu GGSS przeprowadzonych na koniec trwania prac związanych z jego oprogramowaniem. Celem testów była weryfikacja poprawności działania czterach 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

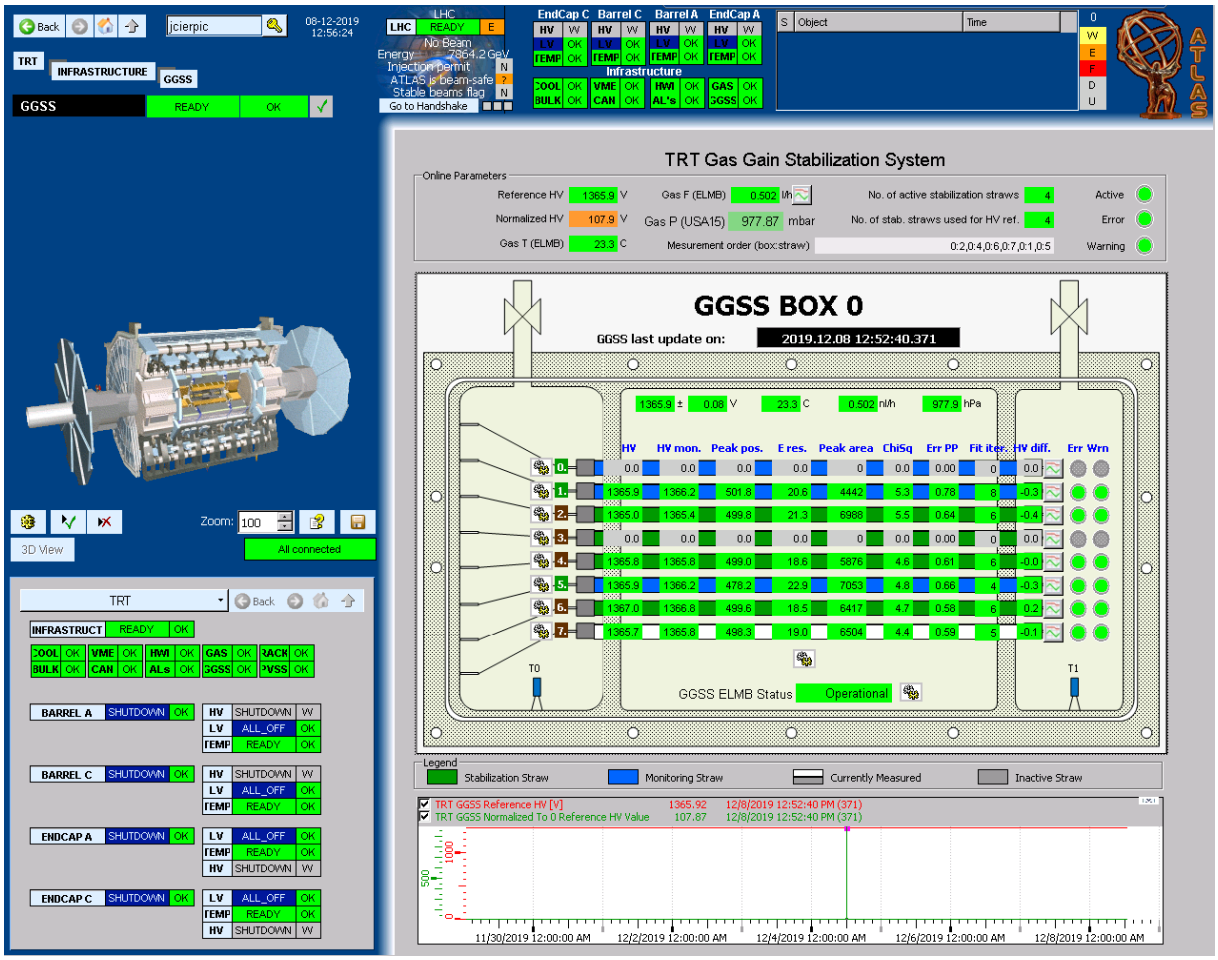
Ze względu na fakt, że środowisko w którym osadzony jest program *ggssrunner* jest ciągle monitorowane, pierwszym krokiem było umieszczenie informacji o przeprowadzaniu testów w dedykowanym do tego celu systemie **ELisA (Electronic Logbook for Information Storage for Atlas)**.

Rys. 7.1. Informacja o przeprowadzaniu testów w systemie ELisA



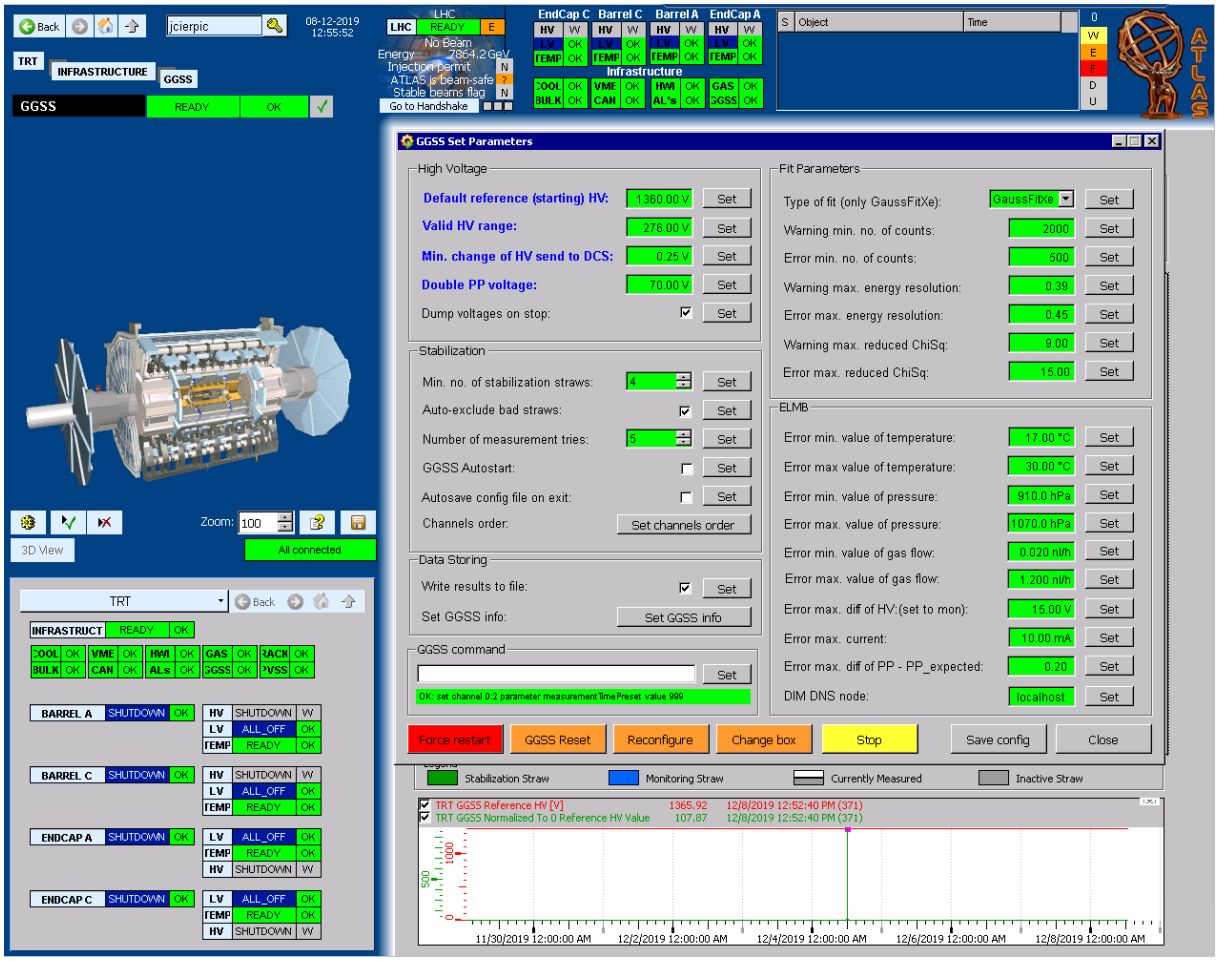
Pliki wykonywalne aplikacji *ggssrunner* wygenerowane zostały za pomocą przygotowanego przez autorów środowiska CI/CD. Zostały one umieszczone na komputerze produkcyjnym. Kolejnym krokiem było zalogowanie się do panelu *WinCC OA* służącego do monitorowania działania detektora ATLAS oraz wybranie panelu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

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*

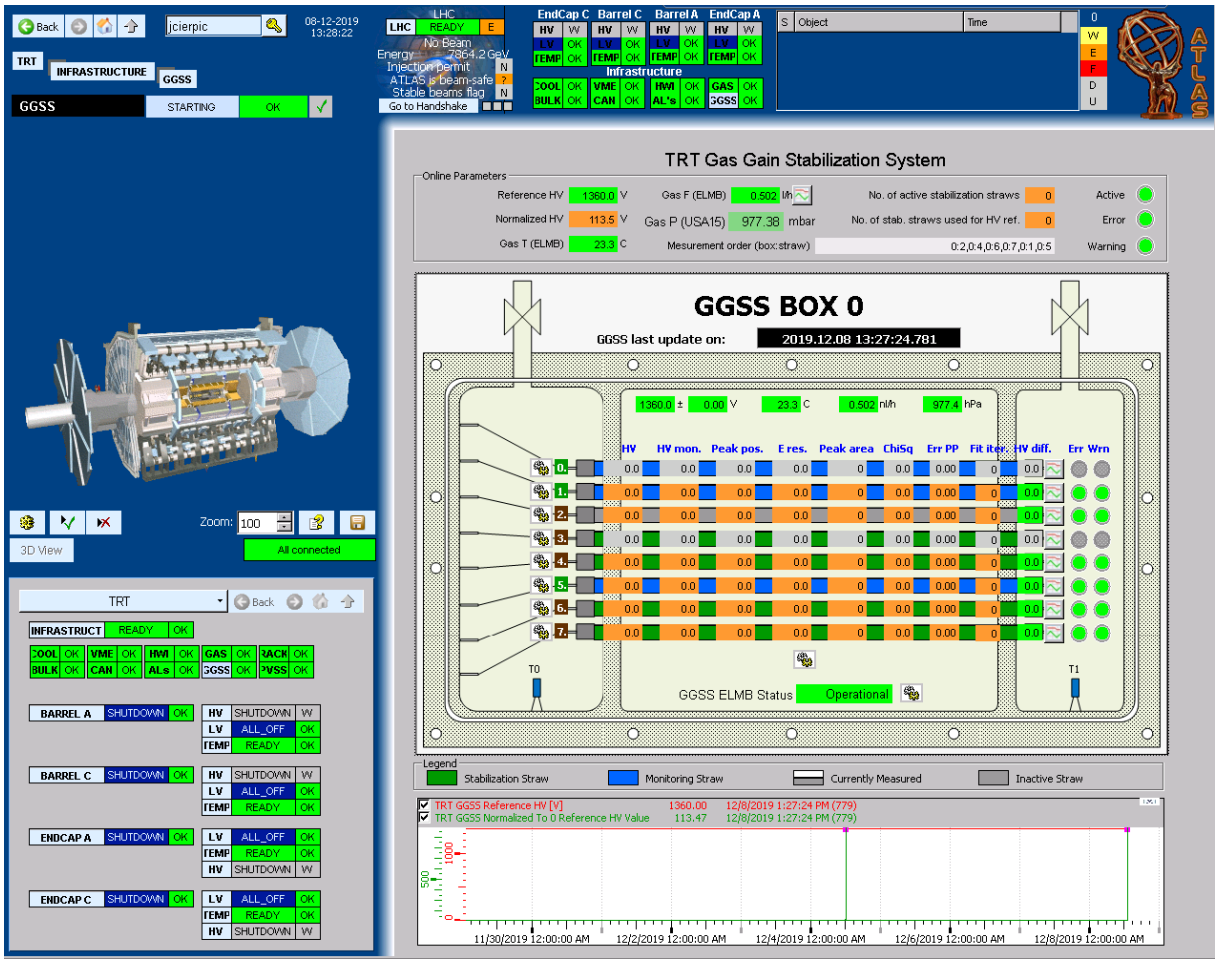
```
user@host:~$ ./ggss_monitor.sh check
ggssrunner is running.

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 modyfikacje **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* służacy 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* służacego 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. Flaga *-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

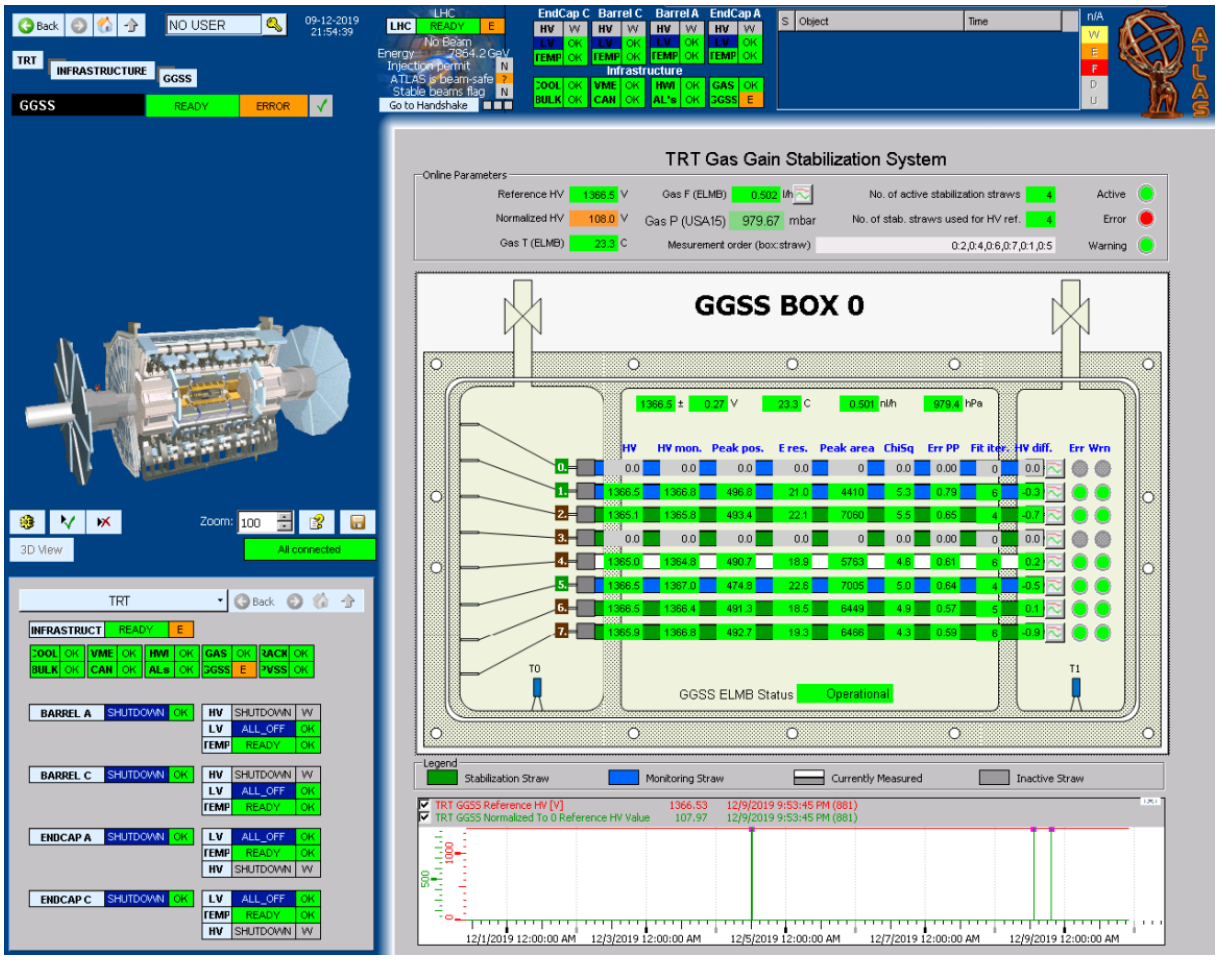
sji produkcyjnej - flaga ta została zastąpiona przez jej łagodniejszy i stabilniejszy odpowiednik *-O2*, powszechnie wykorzystywany do tworzenia wersji produkcyjnych oprogramowania. Testy wykonane z użyciem tej flagi nie wyprodukowały żadnych błędów.

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

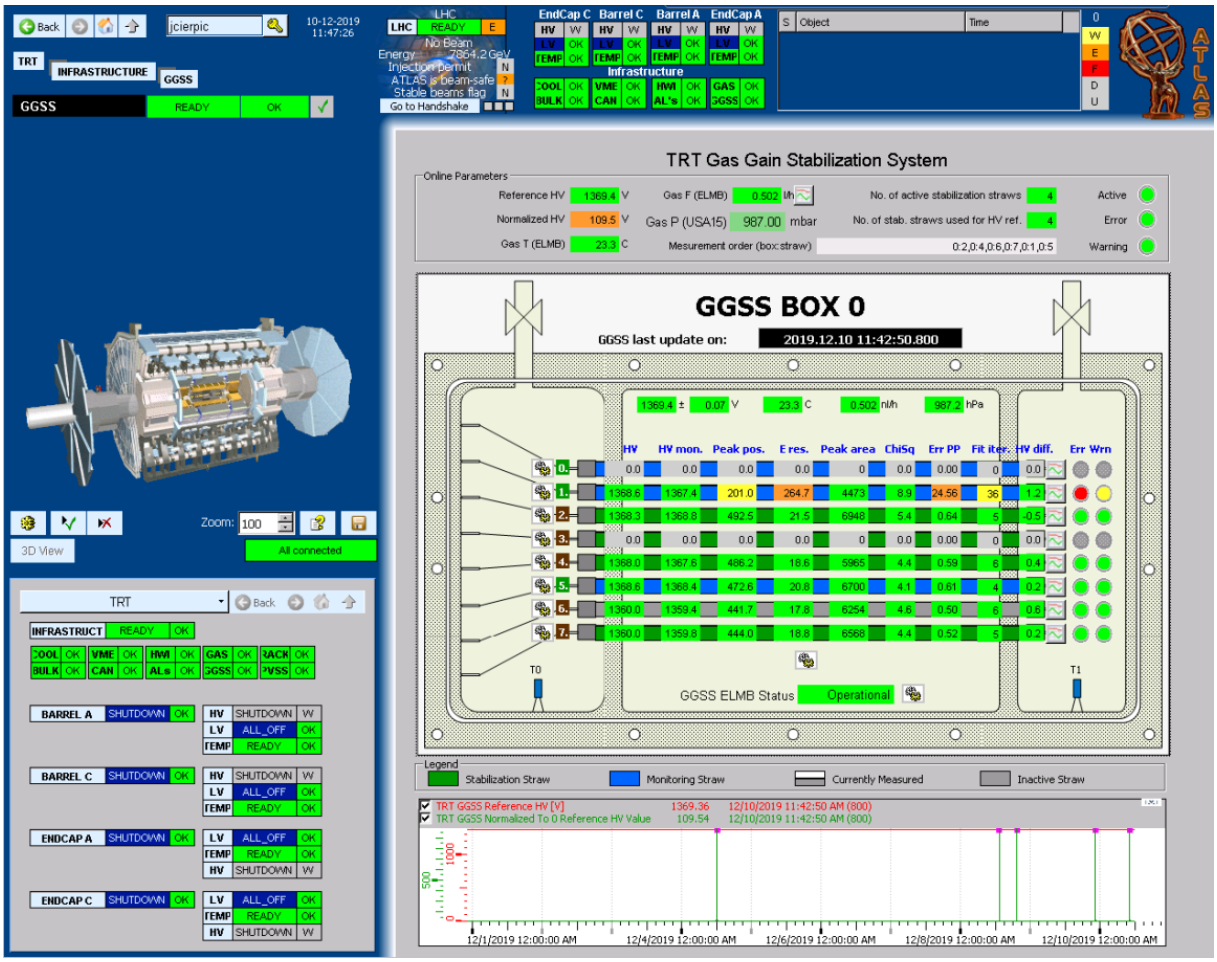
W każdym z przypadków ilość zaalokowanej przez program pamięci pozostawała stała przez cały czas trwania testu. Oznacza to brak znaczących problemów z zarządzaniem pamięcią (takich jak wycieki pamięci).

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

8.1. Wprowadzenie zautomatyzowanego systemu testowania projektu

8.2. Migracja do nowego standardu języka C++

8.3. Automatyzacja procesu publikowania produktu

9. Podsumowanie oraz wnioski

9.1. Statystyki projektu

A. Dodatki/Appendixes

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

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

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

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. Obłąkowska-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] *Bash Reference Manual.* URL: *http://www.gnu.org/software/bash/manual/bash.html#What-is-Bash_003f* (term. wiz. 2019-12-08).

[21] Anand Abhishek Singh. *File states in Git.* URL: *https://anandabhisheksingh.me/file-states-git/* (term. wiz. 2019-12-08).

[22] Wikipedia The Free Encyclopedia. *Package manager.* URL: *https://en.wikipedia.org/wiki/Package_manager* (term. wiz. 2019-12-08).

[23] opensource.com. *What is virtualization?* URL: *https://opensource.com/resources/virtualization* (term. wiz. 2019-12-08).

[24] 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).

[25] *What's new in in CMake.* URL: *https://cliutils.gitlab.io/modern-cmake/chapters/intro/newcmake.html* (term. wiz. 2019-12-07).

[26] 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).

[27] Wikipedia The Free Encyclopedia. *Jenkins (software).* URL: *https://en.wikipedia.org/wiki/Jenkins_(software)* (term. wiz. 2019-12-10).

[28] GitLab. *Auto DevOps.* URL: *https://docs.gitlab.com/ee/topics/autodevops/* (term. wiz. 2019-12-10).

[29] DevKR. *Continuous Integration z GitLab CI.* URL: *https://devkr.pl/2018/01/15/continuous-integration-gitlab-ci/* (term. wiz. 2019-12-10).

[30] GitLab. *GitLab Continuous Integration (CI) and Continuous Delivery (CD).* URL: *https://about.gitlab.com/product/continuous-integration/* (term. wiz. 2019-12-10).

[31] Wikipedia The Free Encyclopedia. *YAML.* URL: *https://en.wikipedia.org/wiki/YAML* (term. wiz. 2019-12-10).