AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

KATEDRA ODDZIAŁYWAŃ I DETEKCJI CZĄSTEK

Praca Dyplomowa

Rozbudowa i uaktualnienie systemu GGSS detektora ATLAS TRT

Update and upgrade of the GGSS system for ATLAS TRT detector

Autorzy: **Arkadiusz Kasprzak, Jarosław Cierpich**

Kierunek studiów: **Informatyka Stosowana**

Opiekun pracy: **dr hab. inż. Bartosz Mindur, prof. AGH**

Kraków, 2021

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)

Kraków, ?? września 2021

Tematyka pracy magisterskiej i praktyki dyplomowej Jarosława Cierpicha, studenta drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności modelowanie i analiza danych

Temat pracy magisterskiej: **Rozbudowa i uaktualnienie systemu GGSS detektora ATLAS TRT**

Opiekun pracy:	dr hab. inż. Bartosz Mindur, prof. AGH
Recenzenci pracy:	
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - udział w Krakow Applied Physics and Computer Science Summer School '20
   - zapoznanie z materiałami (wykłady i szkolenia praktyczne) obejmującymi zagadnienia z dziedziny fizyki cząstek, informatyki oraz detektorów i elektroniki
   - praca nad projektem GGSS w dwuosobowym zespole, obejmująca zmiany w oprogramowaniu i architekturze projektu
   - prezentacja rezultatów wykonanej pracy przed uczestnikami oraz opiekunami szkoły
   - prezentacja wykonanych prac podczas wydarzenia TRT Days
4. Kontynuacja prac nad projektem:
   - wykonanie dalszych zmian w oprogramowaniu systemu GGSS, w tym dodanie nowych funkcjonalności
   - przeprowadzanie okresowych testów działania systemu w środowisku docelowym
   - wykonanie prac nad infrastrukturą projektu
5. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie: ?? września 2021

....

(podpis kierownika katedry) (podpis opiekuna)

Kraków, ?? września 2021

Tematyka pracy magisterskiej i praktyki dyplomowej Arkadiusza Kasprzaka, studenta drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności modelowanie i analiza danych

Temat pracy magisterskiej: **Rozbudowa i uaktualnienie systemu GGSS detektora ATLAS TRT**

Opiekun pracy:	dr hab. inż. Bartosz Mindur, prof. AGH
Recenzenci pracy:	
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - udział w Krakow Applied Physics and Computer Science Summer School '20
   - zapoznanie z materiałami (wykłady i szkolenia praktyczne) obejmującymi zagadnienia z dziedziny fizyki cząstek, informatyki oraz detektorów i elektroniki
   - praca nad projektem GGSS w dwuosobowym zespole, obejmująca zmiany w oprogramowaniu i architekturze projektu
   - prezentacja rezultatów wykonanej pracy przed uczestnikami oraz opiekunami szkoły
   - prezentacja wykonanych prac podczas wydarzenia TRT Days
4. Kontynuacja prac nad projektem:
   - wykonanie dalszych zmian w oprogramowaniu systemu GGSS, w tym dodanie nowych funkcjonalności
   - przeprowadzanie okresowych testów działania systemu w środowisku docelowym
   - wykonanie prac nad infrastrukturą projektu
5. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie: ?? września 2021

....

(podpis kierownika katedry) (podpis opiekuna)

Spis treści

1. Wstęp (AK i JC) ... 11

1.1. Wprowadzenie do systemu GGSS (AK) ... 11

1.2. Cel pracy (JC) ... 12

2. Wykorzystane technologie (AK i JC) ... 13

2.1. Język C++ (AK) ... 13

2.2. Język Python (AK) ... 18

2.3. Narzędzia do analizy oprogramowania (AK) ... 19

2.4. System kontroli wersji Git (JC) ... 25

2.5. Portal GitLab (JC) ... 25

2.6. Narzędzie CMake (AK) ... 26

2.7. Menadżer pakietów RPM (JC) ... 26

3. Budowa i działanie systemu GGSS (AK) ... 27

3.1. Wysokopoziomowa architektura systemu GGSS ... 27

3.2. Urządzenia elektroniczne ... 28

3.3. Warstwa oprogramowania ... 29

3.4. Oprogramowanie WinCC OA ... 30

3.5. Środowisko docelowe i ograniczenia ... 31

4. Prace nad architekturą i infrastrukturą projektu (AK i JC) ... 33

4.1. Zmiany w architekturze projektu (AK) ... 33

4.1.1. Wprowadzenie do problematyki ... 33

4.1.2. Motywacja do wprowadzenia zmian ... 36

4.1.3. Uproszczenie architektury projektu ... 36

4.1.4. Dodanie możliwości odtworzenia pierwotnej wersji kodu źródłowego ... 40

4.1.5. Pomniejsze zmiany ... 40

4.1.6. Podsumowanie ... 42

4.2. Automatyzacja pracy z submodułami (JC) ... 43

4.2.1. Wprowadzenie do problematyki ... 43

4.2.2. Motywacja do wprowadzenia zmian ... 44
4.2.3. Automatyzacja z użyciem GITIO ... 45
4.2.4. Dokumentacja sposobu pracy z submodułami ... 46
4.3. Rozwój systemu budowania projektu (AK) ... 47
4.3.1. Wprowadzenie do problematyki ... 47
4.3.2. Motywacja do wprowadzenia zmian ... 49
4.3.3. Zastosowanie funkcji i makr narzędzia CMake ... 49
4.3.4. Wsparcie dla testów jednostkowych i dokumentacji ... 50
4.3.5. Rozbudowa skryptu konfigurującego proces budowania projektu ... 52
4.4. Automatyzacja i centralizacja wersjonowania projektu (JC) ... 54
4.4.1. Wprowadzenie do problematyki ... 54
4.4.2. Motywacja do wprowadzenia zmian ... 54
4.4.3. 54
4.5. Pakietowanie i rozlokowanie projektu (JC) ... 55
4.5.1. Wprowadzenie do problematyki ... 55
4.5.2. Motywacja do wprowadzenia zmian ... 55
4.5.3. 55
4.6. Rozwój infrastruktury do testowania warstwy sprzętowej (JC) ... 56
4.6.1. Wprowadzenie do problematyki ... 56
4.6.2. Motywacja do wprowadzenia zmian ... 56
4.6.3. Rozwiązanie oparte o język Python ... 56
4.6.4. Rozwiązanie oparte o język C++ ... 56
4.6.5. Podsumowanie ... 56
5. Prace nad kodem źródłowym projektu (AK) ... 57
5.1. Wprowadzenie - analiza aplikacji *ggss-runner* ... 57
5.2. Metodyka prac ... 57
5.2.1. Testy jednostkowe ... 57
5.2.2. Problem *mockowania* ... 57
5.2.3. Statyczna analiza kodu źródłowego ... 57
5.2.4. Przyjęte ograniczenia ... 57
5.3. Poprawa jakości kodu źródłowego ... 57
5.3.1. Migracja do standardu C++11 ... 57
5.3.2. Naprawa błędów w kodzie ... 57
5.3.3. Zmiany w strukturze bibliotek ... 57

5.3.4. Prace nad biblioteką *ggss-lib* ... 57
5.4. Wprowadzenie nowych funkcjonalności ... 57
5.4.1. Biblioteka *hvcommand-lib* ... 57
5.4.2. Zmiany w sposobie aktualizacji parametrów i danych ... 57
5.4.3. Wprowadzenie dodatkowych komend ... 57
5.4.4. Zmiany w algorytmie dopasowywania krzywej ... 57
5.5. Podsumowanie ... 57
6. Testy systemu (AK i JC) ... 59
6.1. Cykliczne testy systemu (AK) ... 59
6.2. Testy po migracji systemu (JC) ... 59
6.3. Testy wersji finalnej (AK i JC) ... 59
7. Podsumowanie (AK i JC) ... 61
A. Przegląd praktyk stosowanych podczas prac nad projektem (JC) ... 63
A.1. Wprowadzenie do problematyki ... 63
A.2. Motywacja do wprowadzenia zmian ... 64
A.3. Zmiana praktyk ze względu na nieregularność prac ... 64
A.4. Dokumentacja projektu ... 66
A.5. Konwencja kodowania ... 69
B. Wybrane instrukcje i poradniki ... 71
B.1. Adding modules to the project using existing CMake templates ... 71
B.2. Working with git submodules ... 71
B.3. Using DIM HV commands ... 71

1. Wstęp (AK i JC)

1.1. Wprowadzenie do systemu GGSS (AK)

Europejska Organizacja Badań Jądrowych CERN jest jednym z najważniejszych ośrodków naukowo-badawczych na świecie i miejscem rozwoju zarówno fizyki, jak i informatyki. Będąc miejscem powstania wielu znaczących technologii (m.in. protokół *HTTP - Hypertext Transfer Protocol*), CERN kojarzony jest dziś przede wszystkim z Wielkim Zderzaczem Hadronów (*LHC - Large Hadron Collider*) - największym akceleratorem cząstek na świecie. Jednym z pracujących przy LHC eksperymentów jest detektor ATLAS (*A Toroidal LHC ApparatuS*), pełniący kluczową rolę w rozwoju współczesnej fizyki - przyczynił się on do potwierdzenia istnienia tzw. bozonu Higgsa w 2012 roku.

Detektor ATLAS zbudowany jest z kilku pod-detektorów, tworzących strukturę warstową. Najbardziej wewnętrzną część stanowi tzw. Detektor Wewnętrzny (ang. *Inner Detector*), składający się z kolei z trzech kolejnych podsystemów. Jednym z tychże podsystemów, szczególnie istotnym w kontekście niniejszej pracy, jest detektor promieniowania przejścia (*TRT - Transition Radiation Tracker*).

System Stabilizacji Wzmocnienia Gazowego (*GGSS - Gas Gain Stabilization System*) jest jednym z podsystemów detektora TRT, mającym zapewnić jego poprawne działanie. Projekt ten zintegrowany jest z systemem kontroli detektora ATLAS (*DCS - Detector Control System*). W skład systemu GGSS wchodzi zarówno warstwa oprogramowania, jak i szereg urządzeń. Ze względu na jego rolę, jednym z najważniejszych wymagań stawianych przed projektem jest wysoka niezawodność.

W niniejszej pracy autorzy przybliżą najważniejsze zmiany dokonane przez nich w czasie półtorarocznych prac nad rozwojem i usprawnieniem systemu GGSS. Prace obejmują przede wszystkim zmiany w warstwie oprogramowania, mające na celu zarówno wprowadzenia nowych funkcjonalności do systemu, jak również uczynienie go bardziej przystępnym dla korzystających z niego osób, m.in. poprzez automatyzację procesów związanych z cyklem życia oprogramowania (np. tworzenie nowych wydań).

1.2. Cel pracy (JC)

Niniejsza praca jest kontynuacją rozwoju Systemu Stabilizacji Wzmocnienia Gazowego rozpoczętego w ramach pracy inżynierskiej o tytule *Rozbudowa i uaktualnienie oprogramowania systemu GGSS detektora ATLAS TRT*. Praca inżynierska skupiała się na aspektach infrastruktury oraz architektury projektu. Przeprowadzono migrację projektu na system kontroli wersji Git. Dokonano przebudowę architektury projektu na bardziej modularną oraz prostszą do zrozumienia. Wprowadzono wiele zmian w projekcie, których celem było udoskonalenie procesu wytwarzania oraz wdrażania oprogramowania w środowisko produkcyjne, np.: wykorzystanie technologii CMake oraz GitLab CI/CD.

Główny nacisk pracy magisterskiej został położony na część aplikacyjną projektu - kod źródłowy odpowiedzialny za główną logikę został rozbudowany oraz udoskonalony. W ramach zmian w kodzie zostały dodane nowe funkcjonalności, nieużywany kod został usunięty z projektu, jakość kodu została poprawiona, a jego poprawne działanie zostało zabezpieczone poprzez testy automatyczne. Ze względu na ciągłą pracę z systemem, poznawanie newralgicznych punktów oraz środowiska, w ramach którego system jest uruchamiany, część pracy magisterskiej zostanie poświęcona kontynuacji prac nad infrastrukturą oraz architekturą. W ramach projektu skupiono się również na aspektach organizacji pracy oraz technik zastosowanych w celu jej poprawienia. Ważnym aspektem pracy, który zostanie uwzględniony w manuskrypcie były zarówno testy automatyczne, testy manualne jak i przygotowanie infrastruktury potrzebnej do ich przeprowadzenia.

Ze względu na bardzo szeroki zakres tematów podejmowanych w tejże pracy zdecydowano się na podział, który odchodzi od standardowego. W celu ułatwienia korzystania z manuskryptu wprowadzenie do opisywanego problemu oraz wykonane prace zostaną zamieszczone w jednym miejscu. Zatem zarówno nakreślenie problemu, stan początkowy oraz sposób jego rozwiązania będą następować zaraz po sobie. Schemat ten zostanie powtórzony dla każdego zagadnienia poruszanego w ramach pracy. Autorzy chcą w ten sposób ułatwić użycie tegoż dokumentu biorąc pod uwagę, iż manuskrypt ma być stosowany zarówno jako wprowadzenie, jak i dokumentacja projektu w celu dalszego rozwoju.

Ostatnim z celów postwionych autorom było odpowiednie udokumentowanie projektu tak, aby ewentualne przyszłe zmiany można było wykonywać z jak największą łatwością, a wprowadzenie nowych osób w projekt było jak najprostsze. Oprócz obszernego opisu zawartego w ramach tego manuskryptu wymogiem było, aby przygotować krótkie, lecz treściwe pliki instruktażowe, opisowe oraz odpowiednio udokumentować kod źródłowy.

2. Wykorzystane technologie (AK i JC)

W niniejszym rozdziale zawarty został opis stosowanych przez autorów pracy technologi. Przedstawione informacje obejmują zarówno wykorzystane języki programowania, jak również narzędzia, których zadaniem jest ułatwienie tworzenia wysokiej jakości kodu źródłowego, czy też automatyzacja procesów zachodzących na różnych etapach cyklu życia oprogramowania. Niniejszy rozdział stanowi krótkie wprowadzenie do każdego z omawianych zagadnień - zaawansowane aspekty każdej z opisywanych technologii przedstawione zostały w dalszej części pracy, podczas omawiania konkretnych, osiągniętych za ich pomocą, rozwiązań.

2.1. Język C++ (AK)

C++ to stworzony w latach 80-tych XX wieku wszechstronny język programowania, oryginalnie mający stanowić rozszerzenie popularnego języka C o mechanizmy pozwalające na programowania obiektowe. Wraz z jego rozwojem pojawiło się natomiast wsparcie dla innych paradygmatów programowania, dzięki czemu nowoczesny C++ pozwala stosować (poza paradygmatem proceduralnym oraz obiektowym) programowanie funkcyjne oraz generyczne. Z tego też powodu język ten znajduje współcześnie bardzo szerokie zastosowanie, od rozwiązań telekomunikacyjnych po oprogramowanie dla eksperymentów fizycznych, takich jak detektor ATLAS w CERN. Jest on ponadto bardzo istotny z punktu widzenia niniejszej pracy, ponieważ za jego pomocą napisana została większość oprogramowania systemu GGSS.

C++ jest wydajnym językiem kompilowanym, opartym o statyczne typowanie. Udostępnia mechanizmy pozwalające programiście na działanie na wielu poziomach abstrakcji - możliwe są zarówno niskopoziomowe operacje, takie jak manualne zarządzanie pamięcią, jak również modelowanie wysokopoziomowej logiki biznesowej. W ciągu ostatnich dziesięciu lat miał miejsce szczególnie intensywny rozwój języka, czego początek stanowi pojawienie się przełomowego standardu C++11. Od tego czasu regularnie, co trzy lata, wydawana jest nowa wersja standardu języka, a zatem od 2011 roku pojawiły się następujące wydania:

– C++11 - uznawane za przełomowe, zawiera modyfikacje w znacznym stopniu zmieniające sposób tworzenia oprogramowania w języku C++. Wprowadzone zostały zarówno rozszerzenia w rdzeniu języka, jak i w bibliotece standardowej. Najważniejsze elementy standardu C++11 to m.in.: semantyka przenoszenia i referencje do *r-wartości*, wyrażenia

lambda, słowa kluczowe `override` i `final` ułatwiające programowanie obiektowe, inferencja typów za pomocą słowa kluczowego `auto`, wsparcie dla wielowątkowości (model pamięci oraz funkcjonalności w bibliotece standardowej), tzw. inteligentne wskaźniki (ang. *smart pointers*) ułatwiające zarządzanie pamięcią czy nowe kontenery biblioteki standardowej.

- C++14 - mniejsze wydanie, stanowiące uzupełnienie standardu C++11 o pomniejsze funkcjonalności oraz poprawki. Wprowadzone zmiany to m.in. ułatwienia w korzystaniu ze słowa kluczowego `constexpr` oraz generyczne wyrażenia lambda.
- C++17 - wprowadza wiele nowych funkcjonalności, m.in. bibliotekę do obsługi systemu plików, typ `std::optional` czy możliwość wykonania inicjalizacji w wyrażeniu warunkowym
- C++20 - najnowsze wydanie języka, pod względem liczby wprowadzonych zmian większe od dwóch poprzednich. Przykładowe elementy tego standardu to: koncepty (ang. *concepts*), moduły czy biblioteka pozwalająca na operacje na zakresach (ang. *ranges*).

Obecnie trwają pracę nad nowym standardem języka, którego publikacja planowana jest na rok 2023. Poza wprowadzaniem funkcjonalności, nowe wydania języka C++ eliminują te elementy języka, które uznawane są za przestarzałe. Przykładem może być obecny w bibliotece standardowej od wczesnych wersji języka inteligentny wskaźnik `std::auto_ptr` - został on oznaczony jako przestarzały (ang. *deprecated*) po pojawieniu się nowych rozwiązań w standardzie C++11, a następnie został usunięty z języka wraz z wprowadzeniem standardu C++17. Tego typu zmiany mają na celu wspieranie tzw. *dobrych praktyk* - zasad ułatwiających tworzenie łatwego w utrzymaniu i rozwoju oprogramowania (np. poprzez odpowiednie zarządzanie zasobami).

Z punktu widzenia niniejszej pracy szczególnie istotne są zmiany wprowadzone w standardzie C++11 - z uwagi na ograniczenia w środowisku docelowym systemu GGSS jest to najnowsze dostępne tam wydanie języka (więcej informacji na ten temat przedstawione zostanie w dalszej części pracy). Dlatego też zaprezentowany zostanie krótki przykład obrazujący część funkcjonalności wprowadzonych właśnie w tym standardzie. W przykładzie tym zaimplementowana została prosta hierarchia klas reprezentujących zasilacze: jedna abstrakcyjna klasa bazowa `PowerSupply` oraz dwie implementacje: `CaenPowerSupply` oraz `MockPowerSupply`. W funkcji `main()` tworzony i wypełniany jest kontener przechowujący wskaźniki zawierające adresy obiektów reprezentujących różne typy zasilaczy. Następnie dla każdego z istniejących zasilaczy następuje polimorficzne wywołanie metody pozwalającej na zmianę wartości zasilania (tutaj powoduje jedynie wypisanie odpowiedniej wiadomości na standardowe wyjście). Celem przykładu jest zaprezentowanie prostego scenariusza, w którym nowe funkcjonalności języka wpływają pozytywnie na jakość i bezpieczeństwo kodu - nie prezentuje więc on w sposób bezpośredni zaawansowanych elementów języka (takich jak możliwość metaprogramowania za pomocą szablonów).

Na listingu 2.1 przedstawiona została implementacja przykładu zgodna ze standardem C++03. Najważniejsze cechy zaprezentowanego kodu, charakterystyczne dla kodu źródłowego powstającego przed pojawieniem się standardu C++11, to:

- konieczność manualnego zarządzania pamięcią - widoczne zastosowanie operatora `delete` pod koniec funkcji `main()`
- brak bezpośredniej możliwości zadeklarowania klasy jako *finalna* - tzn. taka, po której nie można dziedziczyć (przed standardem C++11 istniały jednak techniki, wykorzystujące zaawansowane funkcjonalności języka, pozwalające osiągnąć podobny rezultat - ze względu na stopień skomplikowania nie zostały tu jednak zaprezentowane)
- brak możliwości wskazania, że metoda w klasie pochodnej nadpisuje (ang. *override*) metodę wirtualną z klasy bazowej
- rozbudowana, nieczytelna składnia pętli `for` operującej na kontenerze za pomocą iteratora

Listing 2.1. Przykładowa implementacja prostej hierarchii dziedziczenia oraz polimorficznego wykonania metody - standard C++03. Należy zwrócić uwagę na konieczność manualnego zarządzania pamięcią oraz brak możliwości wskazania, że metoda w klasie pochodnej nadpisuje metodę wirtualną z klasy bazowej.

```
#include <vector>
#include <iostream>

// klasa abstrakcyjna reprezentująca zasilacz, udostępnia interfejs pozwalający
// na ustawienie wartości napięcia
class PowerSupply {
public:
    virtual ~PowerSupply() {};
    virtual void set_voltage(double value) const = 0;
};

// klasa reprezentująca konkretny typ zasilacza
class CaenPowerSupply : public PowerSupply {
public:
    void set_voltage(double value) const {
        std::cout << "Setting voltage to value: " << value << std::endl;
    }
};

// klasa reprezentująca atrapę (mock) zasilacza
class MockPowerSupply : public PowerSupply {
public:
    void set_voltage(double value) const {
        std::cout << "Setting mock voltage to value: " << value << std::endl;
    }
};
```

```
int main() {
    // utworzenie i wypełnienie kontenera przechowującego wskaźniki
    // zawierające adresy obiektów reprezentujących zasilacze
    std::vector<PowerSupply*> power_supply_units;
    power_supply_units.push_back(new CaenPowerSupply());
    power_supply_units.push_back(new MockPowerSupply());

    // iteracja po kontenerze, polimorficzne wywołanie metody set_voltage
    for(std::vector<PowerSupply*>::const_iterator it = ↩
        power_supply_units.begin(); it != power_supply_units.end(); ++it) {
        (*it)->set_voltage(2.5);
    }

    // zwalnianie pamięci
    delete power_supply_units[0];
    delete power_supply_units[1];
}
```

Na listingu 2.2 przedstawiona została natomiast implementacja przykładu za pomocą języka C++ w standardzie 11. Najważniejszą zmianą jest zastosowanie inteligentnego wskaźnika `std::unique_ptr<PowerSupply>`, automatyzującego zarządzanie wskazywanym zasobem (pamięć zostaje zwolniona, gdy wskaźnik ulega destrukcji). Powoduje to, że programista nie jest odpowiedzialny za manualne sprawowanie kontroli nad pamięcią, a co za tym idzie zmniejsza prawdopodobieństwo wystąpienia związanych z tym błędów (takich jak wycieki pamięci). Kolejną zmianą jest zastosowanie słowa kluczowego `override` w celu zadeklarowania, że metoda w klasie pochodnej (np. `CaenPowerSupply`) nadpisuje metodę z klasy bazowej (tutaj `PowerSupply`). W przypadku, gdy powyższe nie jest prawdą (np. za sprawą błędnej pisowni lub braku słowa kluczowego `const`), kompilator zgłosi błąd. W klasach pochodnych zastosowane zostało ponadto słowo kluczowe `final`, powodujące, że po klasach tych nie można dziedziczyć - jego stosowanie może wynikać m.in. z zalecenia mówiącego, że dziedziczenie powinno być możliwe jedynie w przypadku klas, które są z myślą o nim projektowane (np. klasy abstrakcyjne). Wymienione powyżej zmiany, możliwe dzięki stosowaniu funkcjonalności nowoczesnego języka C++, pozwalają na zwiększenie niezawodności tworzonego kodu źródłowego, m.in. poprzez zabezpieczenie go przed prostymi błędami oraz dostarczenie dodatkowej, wbudowanej wprost w język, dokumentacji. Innym typem zmiany, nastawionym w większym stopniu na zwiększenie czytelności kodu, jest natomiast zastosowanie w przykładzie pętli zakresowej do iteracji po kontenerze oraz wykorzystanie słowa kluczowego `default` w deklaracji domyślnego destruktora wirtualnego klasy bazowej. W standardzie C++11 wprowadzonych zostało znacznie więcej podobnych udoskonaleń, a ponadto wprowadzone zostały mechanizmy ułatwiające optymalizację oprogramowania (np. ze względu na szybkość wykonania lub ilość zużytej pamięci) - jednak ze względu na konieczność zachowania prostego charakteru przykładu nie zostały one zaprezentowane.

Listing 2.2. Przykładowa implementacja prostej hierarchii dziedziczenia oraz polimorficznego wykonania metody - standard C++11. Widoczne zastosowanie słów kluczowych `default`, `final` oraz `override`, pętli zakresowej oraz inteligentnego wskaźnika.

```
#include <vector>
#include <iostream>
#include <memory>

// klasa abstrakcyjna reprezentująca zasilacz, udostępnia interfejs pozwalający
// na ustawienie wartości napięcia
class PowerSupply {
public:
    virtual ~PowerSupply() = default;
    virtual void set_voltage(double value) const = 0;
};

// klasa reprezentująca konkretny typ zasilacza
class CaenPowerSupply final : public PowerSupply {
public:
    void set_voltage(double value) const override {
        std::cout << "Setting voltage to value: " << value << std::endl;
    }
};

// klasa reprezentująca atrapę (mock) zasilacza
class MockPowerSupply final : public PowerSupply {
public:
    void set_voltage(double value) const override {
        std::cout << "Setting mock voltage to value: " << value << std::endl;
    }
};


int main() {
    // utworzenie i wypełnienie kontenera przechowującego inteligentne
    // wskaźniki zawierające adresy obiektów reprezentujących zasilacze
    std::vector<std::unique_ptr<PowerSupply>> power_supply_units{};
    power_supply_units.emplace_back(new CaenPowerSupply());
    power_supply_units.emplace_back(new MockPowerSupply());

    // iteracja po kontenerze, polimorficzne wywołanie metody set_voltage
    for(const auto& psu : power_supply_units) {
        psu->set_voltage(2.5);
    }
}
```

Język C++ charakteryzuje się ponadto istnieniem dodatkowych, nie będących częścią standardu, bibliotek poszerzających zestaw dostarczanych przez niego narzędzi. Z punktu widzenia systemu GGSS istotne są: zestaw bibliotek Boost, biblioteka GNU Scientific Library (GSL) oraz zestaw bibliotek i narzędzi Qt.

Boost to popularna kolekcja bibliotek dla języka C++, ułatwiająca wiele aspektów tworzenia oprogramowania poprzez dostarczenie zróżnicowanego zestawu narzędzi programistycznych: zarówno ogólnego przeznaczenia, jak i bardzo wyspecjalizowanych. Pakiet Boost rozwijany jest znacznie szybciej niż biblioteka standardowa języka C++, a ponadto niektóre jego elementy stanowiły podstawę dla funkcjonalności dodawanych do języka w nowych jego wydaniach (np. wprowadzona w standardzie C++17 biblioteka `filesystem`, służąca do zarządzania systemem plików, oparta jest na analogicznym module z zestawu Boost). Przykładowe funkcjonalności udostępniane przez pakiet Boost to: rozszerzona względem biblioteki standardowej obsługa łańcuchów znakowych, dodatkowe kontenery (np. `Boost.MultiIndex`), implementacja algorytmów grafowych czy asynchroniczne programowanie sieciowe. Ze względu na szeroki zakres oferowanych funkcjonalności, pakiet Boost stosowany jest również przez warstwę oprogramowania omawianego w niniejszej pracy systemu GGSS.

GNU Scientific Library (GSL) to napisana w języku C biblioteka udostępniająca narzędzia programistyczne do wykonywania obliczeń numerycznych. Pozwala na wykonywanie operacji takich jak: znajdowanie miejsc zerowych funkcji, dopasowywanie krzywej do danych czy całkowanie metodą Monte Carlo. Ze względu na drugą z wymienionych tu funkcjonalności biblioteka ta znalazła zastosowanie w oprogramowaniu systemu GGSS.

Qt jest zestawem narzędzi programistycznych umożliwiających tworzenie przenośnych aplikacji okienkowych z wykorzystaniem języka C++. Technologia ta nie została zastosowana w rdzeniu oprogramowania systemu GGSS, natomiast przy jej pomocy stworzone zostały pomniejsze narzędzia wchodzące w skład projektu. Opisane w niniejszej pracy rozwiązania nie są oparte o Qt, dlatego też szczegółowy opis dostarczanych przez zestaw narzędzi nie zostanie w niej zamieszczony.

2.2. Język Python (AK)

Python jest opartym na dynamicznym systemie typów językiem programowania ogólnego przeznaczenia, charakteryzującym się bardzo szerokim obszarem zastosowań, obejmującym m.in. automatyzację za pomocą skryptów, tworzenie aplikacji internetowych czy eksplorację danych. Cechą najczęściej kojarzoną z tym językiem jest intuicyjna składnia, ułatwiająca zarówno jego naukę, jak i zrozumienie napisanych z jego pomocą programów. Python jest językiem wieloparadygmatowym, pozwalającym pisać zarówno w sposób proceduralny, jak i obiektowo i funkcyjnie. W systemie GGSS język ten stosowany jest jako narzędzie pomocnicze, rozumiane przede wszystkim jako język skryptowy wykorzystywany do tworzenia infrastruktury projektu.

Obecnie język Python istnieje w dwóch szeroko stosowanych wersjach: Python 2 oraz Python 3. Oficjalnie wspieraną wersją jest wydanie trzecie (wsparcie dla Pythona w wersji drugiej zakończone zostało na początku 2020 roku), w rzeczywistości jednak, ze względu na fakt, iż wersje te nie są ze sobą w pełni kompatybilne, oprogramowanie napisane za pomocą Pythona 2 wciąż znaleźć można w wielu projektach. Z punktu widzenia systemu GGSS różnice między tymi wydaniami nie są w dużym stopniu znaczące (ponieważ język ten używany jest jako pomocnicze narzędzie), jednakże preferowana jest wersja trzecia.

Na listingu 2.3 przedstawiony został prosty przykład zastosowania języka Python jako narzędzia do tworzenia skryptów. W tym przypadku jest to prosty skrypt, generujący sto pierwszych elementów ciągu Fibonacciego i zapisujący je do pliku tekstowego w postaci kolumny. Przykład obrazuje wykorzystanie takich elementów języka jak generatory, pętle oraz operacje na plikach.

Listing 2.3. Przykładowy skrypt napisany w języku Python, którego działanie polega na wygenerowaniu stu pierwszych elementów ciągu Fibonacciego i zapisaniu otrzymanego wyniku do pliku tekstowego.

```
# generator zwracający kolejne wyrazy ciągu Fibonacciego
def fibonacci(n):
    first, second = 0, 1
    for _ in range(n):
        yield first
        first, second = second, first + second

# wykorzystanie generatora - zapis stu pierwszych
# elementów ciągu Fibonacciego do pliku tekstowego
with open('data.txt', 'w') as file:
    for elem in fibonacci(100):
        file.write(str(elem) + '\n')
```

2.3. Narzędzia do analizy oprogramowania (AK)

Podczas prac nad systemem GGSS autorzy wykorzystali szereg narzędzi ułatwiających utrzymanie jakości oraz niezawodności tworzonego oprogramowania. Współcześnie, wraz z rosnącym skomplikowaniem powstających projektów informatycznych, rośnie zapotrzebowanie na narzędzia wspierające analizę tworzonego oraz rozwijanego oprogramowania - jest ono często utrzymywane przez wiele osób, przez co zachowanie jego wysokiej jakości staje się coraz bardziej wymagającym zadaniem. Opisywane narzędzia charakteryzują się szerokim zakresem oferowanych możliwości, m.in. monitorowanie zużycia zasobów czy też badanie kodu źródłowego pod kątem zgodności z pewnymi przyjętymi standardami, oraz wspierają wykonywanie analizy programu na wiele zróżnicowanych sposobów. W najprostszym ujęciu wskazać można dwie możliwości klasyfikacji technik analizy oprogramowania:

– analiza kodu źródłowego (ang. *source analysis*) oraz analiza kodu maszynowego (ang. *binary analysis*) - w przypadku pierwszego typu badaniom poddawany jest kod źródłowy, napisany np. z wykorzystaniem języka C++, drugie podejście polega natomiast na analizie programu na poziomie kodu maszynowego lub reprezentacji pośredniej, takiej jak kod bajtowy (ang. *bytecode*).
– analiza statyczna (ang. *static analysis*) oraz analiza dynamiczna (ang. *dynamic analysis*) - podejścia te różni to, czy badanie wykonywane jest bez uruchamiania programu (pierwszy typ), czy też w trakcie jego wykonywania (drugi rodzaj).

Powyższe dwa podziały pozwalają ostatecznie określić cztery kategorie analizy oprogramowania: statyczna analiza kodu źródłowego, statyczna analiza kodu maszynowego, dynamiczna analiza kodu źródłowego oraz dynamiczna analiza kodu maszynowego. Każde z tych podejść charakteryzuje się pewnymi zaletami względem pozostałych, należy je jednak traktować jako komplementarne - wskazane jest, w zależności od potrzeb, wykorzystywanie różnego typu narzędzi ułatwiających utrzymanie wysokiej jakości tworzonego oprogramowania. Szczegółowa analiza każdej z wymienionych możliwości wykracza poza zakres niniejszej pracy, w dalszej jej części przedstawione zostały jedynie najważniejsze informację na temat szczególnie istotnych narzędzi, które wykorzystywali autorzy podczas prac nad systemem GGSS. W zamieszczonym opisie pominięte zostało jednak oprogramowanie, którego znajomość nie jest konieczna w celu zrozumienia dalszych części niniejszego manuskryptu, takie jak GNU Debugger (GDB).

Z punktu widzenia testów, jakie wykonywane były podczas prac nad systemem GGSS, bardzo istotna jest platforma Valgrind. Jej działanie opiera się na przeprowadzaniu tzw. instrumentacji - umieszczaniu w kodzie dodatkowych instrukcji, pozwalających na wykonywanie jego analizy, co w tym przypadku ma miejsce w czasie wykonywania programu. Valgrind pozwala na tworzenie wyspecjalizowanych narzędzi wykorzystujących dynamiczną instrumentację binarną, dzięki udostępnieniu rdzenia (ang. *core*), którego zadaniem jest deasemblacja kodu maszynowego do tzw. postaci pośredniej. Postać ta wykorzystywana jest przez poszczególne narzędzia, które dodają odpowiedni dla danego zastosowania kod analizujący. Szczególnie istotne z punktu widzenia niniejszej pracy są narzędzia Memcheck oraz Massif, których zadaniem jest monitorowanie, w jaki sposób badany program wykorzystuje dostępną pamięć.

Memcheck jest domyślnym narzędziem udostępnianym przez platformę Valgrind, pozwalającym na wykrywanie błędów związanych z zarządzaniem pamięcią w programach napisanych w językach C i C++. Przykładem tego typu błędów są wycieki pamięci, użycie zmiennych, którym nie została nadana żadna wartość, oraz próby dostępu do zwolnionej pamięci. Memcheck jest szczególnie istotny w kontekście testów systemu GGSS, ponieważ pozwala upewnić się, że podczas działania aplikacji wchodzących w skład projektu nie mają miejsca żadne błędy mogące powodować stopniowy wzrost zużycia zasobów, co w przypadku systemu działającego nieprzerwanie przez wiele miesięcy jako usługa mogłoby doprowadzić do niespodziewanej awarii.

Na listingu 2.4 przedstawiony został fragment kodu źródłowego napisanego w języku C++, zawierającego przykłady prostych błędów, które mogą zostać wykryte przy użyciu narzędzia Memcheck: użycie niezainicjalizowanej zmiennej oraz wyciek pamięci. W funkcji `main()` tworzona jest zmienna `no_value`, której nie zostaje nadana wartość. Następnie na jej podstawie podejmowana jest decyzja o zmianie wartości zmiennej `value` - sprawia to, że działanie programu jest niemożliwe do przewidzenia. W ostatniej linii funkcji `main()` tworzona jest dynamiczna tablica dziesięciu liczb całkowitych - ponieważ w programie nie następuje jej zwolnienie, jest to prosta forma wycieku pamięci.

Listing 2.4. Przykład prostego programu zawierającego błędu możliwe do wykrycia za pomocą narzędzia Memcheck: użycie zmiennej, której nie została nadana wartość oraz wyciek pamięci.

```
#include <iostream>

int main() {
    int no_value; // niezainicjalizowana zmienna
    int value{10};
    if(no_value == 0) { // użycie niezainicjalizowanej zmiennej
        std::cout << "No-value equal to zero. Setting value to 5." << std::endl;
        value = 5;
    }
    std::cout << "Value: " << value << std::endl;

    int* leak = new int[10]; // wyciek pamięci - brak delete[]
}
```

Na listingu 2.5 przedstawione zostało natomiast wywołanie narzędzia Memcheck w celu zbadania stanowiącego przykład programu. W zamieszczonym wyjściu widoczne są informacje na temat wykrycia obu błędów. Ponadto, ponieważ nastąpiło wywołanie z użyciem opcji `--track-origins=yes`, wskazywane jest źródło wystąpienia nieprawidłowości (np. w przypadku niezainicjalizowanej zmiennej jest to linia, w której następuje jej definicja). Ponadto wyświetlane są: podsumowanie zawierające informacje na temat dynamicznych alokacji pamięci (czyli na tzw. stercie - ang. *heap*) oraz raport podsumowujący wykryte przez narzędzie wycieki pamięci, w zależności od ich rodzaju.

Listing 2.5. Wywołanie narzędzia Memcheck w celu zbadania programu zamieszczonego na listingu 2.4 - widoczne informacje na temat obu obecnych w przykładzie błędów.

```
user@host:~/test$ valgrind --track-origins=yes --leak-check=full ./a.out
==127== Memcheck, a memory error detector
==127== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==127== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==127== Command: ./a.out
==127==
==127== Conditional jump or move depends on uninitialised value(s)
```

```
==127==    at 0x109200: main (memcheck_code.cpp:7)
==127==  Uninitialised value was created by a stack allocation
==127==    at 0x1091E9: main (memcheck_code.cpp:3)
==127==
Value: 10
==127==
==127== HEAP SUMMARY:
==127==     in use at exit: 40 bytes in 1 blocks
==127==   total heap usage: 3 allocs, 2 frees, 76,840 bytes allocated
==127==
==127== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==127==    at 0x483C583: operator new[](unsigned long) (in ↩
    /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==127==    by 0x109272: main (memcheck_code.cpp:13)
==127==
==127== LEAK SUMMARY:
==127==    definitely lost: 40 bytes in 1 blocks
==127==    indirectly lost: 0 bytes in 0 blocks
==127==      possibly lost: 0 bytes in 0 blocks
==127==    still reachable: 0 bytes in 0 blocks
==127==         suppressed: 0 bytes in 0 blocks
==127==
==127== For lists of detected and suppressed errors, rerun with: -s
==127== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

W niniejszej pracy pominięty został szczegółowy opis technik stosowanych przez narzędzie Memcheck w celu wykrywania błędów występujących w badanym programie. Istotna jest natomiast informacja, że przeprowadzana analiza oparta jest na wykorzystywaniu dodatkowych zasobów, w tym znaczącej ilości pamięci. W ten sposób działa m.in. mechanizm wykrywanie użycia niezainicjalizowanych zmiennych - każdy bit danych zarządzany przez badany proces jest śledzony przez narzędzie za pomocą metadanych w postaci innego bitu (tzw. *valid-value bit*). Powoduje to znaczący wzrost zużycia pamięci, a zatem uniemożliwia jednoczesne przeprowadzanie (np. za pomocą innych, zewnętrznych narzędzi) badań dotyczących ilości wykorzystywanych przez analizowany program zasobów.

Drugim, istotnym z uwagi na prezentowane w niniejszym manuskrypcie treści, narzędziem opartym o platformę Valgrind jest Massif. Umożliwia on generowanie szczegółowych raportów opisujących wykorzystanie pamięci sterty (oraz opcjonalnie stosu) przez badany program. Pozwala to wykrywać scenariusze takie jak stopniowo rosnący, wraz z długotrwałym działaniem programu, rozmiar dynamicznie alokowanych struktur danych. Ponieważ w takim przypadku wyciek pamięci nie następuje w sposób jawny, problem tego typu jest trudny do wykrycia za pomocą narzędzia Memcheck. Uproszczony przykład tego typu scenariusza przedstawiony został na listingu 2.6. Zamieszczony tam program symuluje sterowany przez użytkownika pomiar: w każdej iteracji pętli `do-while` następuje alokacja pamięci, a następnie na podstawie wprowadzonego na

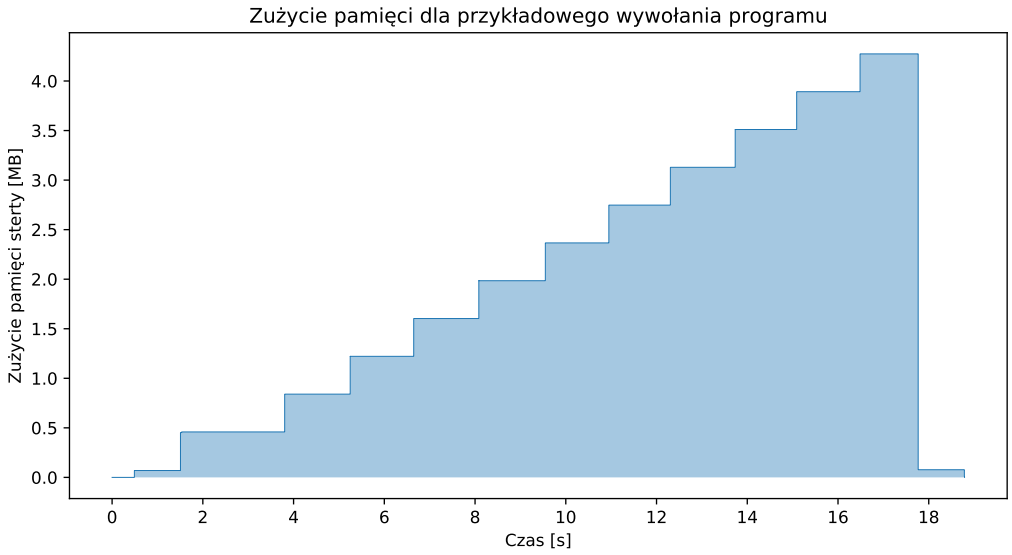
standardowe wejście znaku podejmowana jest decyzja o kontynuacji lub zakończeniu działania. Po zakończeniu wykonywania pętli następuje zwolnienie pamięci. Przedstawiony program nie zawiera błędów związanych z zarządzaniem pamięciom - zastosowania narzędzia Memcheck nie wskazuje na występowanie żadnych problemów. Każda iteracja pętli `do-while` wiąże się jednak ze zwiększeniem rozmiaru wykorzystywanej przez program pamięci sterty, co ostatecznie może doprowadzić do jej wyczerpania, a co za tym idzie, do błędu kończącego działanie aplikacji. Tego typu scenariusz jest szczególnie niebezpieczny w przypadku aplikacji, od których oczekuje się bezawaryjnego działania przez bardzo długi czas (np. kilka miesięcy) - w tego typu przypadkach mechanizm sterujący alokacją pamięci może być znacząco bardziej skomplikowany, niż ten przedstawiony w przykładzie, a co za tym idzie wczesne wykrycie potencjalnego problemu może być niemożliwe bez zastosowania narzędzia takiego jak Massif.

Listing 2.6. Przykład prostego programu symulującego cykliczne wykonywanie pomiaru. W każdej iteracji wykonywana jest dynamiczna alokacja pamięci, a użytkownik za pomocą standardowego wejścia decyduje o zaprzestaniu lub kontynuacji działania programu. Załączony kod obrazować ma sytuację, w której program stopniowo zwiększa zużycie dostępnych zasobów, co w konsekwencji doprowadzić może do przerwania jego działania, gdy zostaną one wyczerpane.

```
#include <iostream>
#include <vector>
#include <unistd.h>

constexpr unsigned int NUMBER_OF_SAMPLES{100000};

int main() {
    std::vector<int*> measurement_history{};

    // pętla symulująca następujący cyklicznie pomiar
    // o przerwaniu lub kontynuacji pomiaru decyduje użytkownik
    do {
        sleep(1);
        measurement_history.push_back(new int[NUMBER_OF_SAMPLES]);
        std::cout << "Measurement performed. Press q to stop or anything else to ↩
            continue ..." << std::endl;
    } while(std::cin.get() != 'q');

    // zwolnienie zaalokowaneej pamięci
    for(int* iteration: measurement_history) {
        delete [] iteration;
    }
    sleep(1);
}
```

Na rysunku 2.1 przedstawiony został natomiast wykres wykonany za pomocą danych, jakie znalazły się w raporcie wygenerowanym przez narzędzie Massif. Przedstawia on zużycie pamięci

sterty w funkcji czasu dla przykładowego uruchomienia zaprezentowanego programu - wyraźnie widoczny jest wzrost ilości wykorzystywanych zasobów wraz z kolejnymi alokacjami następującymi w pętli `do-while`. Poza danymi liczbowymi na temat zużycia przez badany program pamięci, generowany przez narzędzie raport zawiera informacje na temat następujących alokacji, co ułatwia znajdowanie źródeł wzrostu wykorzystania zasobów. Ze względu na znaczny rozmiar generowanego raportu nie został on zaprezentowany w niniejszym manuskrypcie.



Rys. 2.1. Wykres przedstawiający ilość wykorzystywanej pamięci sterty w funkcji czasu. Zaprezentowane dane pochodzą z przykładowego uruchomienia programu zawartego na listingu 2.6 - ich pozyskanie możliwe było dzięki wykorzystaniu wygenerowanego przez narzędzie Massif raportu.

Innym typem narzędzi wykorzystywanych przez autorów niniejszej pracy są te służące do statycznej analizy kodu źródłowego w celu określenia jego zgodności z przyjętymi konwencjami. Wykonywane w ten sposób badanie pozwala wykryć nieprawidłowości związane zarówno z tak prostymi zagadnieniami jak przyjęty styl (np. nazewnictwo zmiennych czy kolejność załączania plików nagłówkowych), jak również z wykorzystaniem poszczególnych elementów języka programowania (np. wykrywanie przestarzałych, niezalecanych konstrukcji, takich jak wspomniany w części opisującej język C++ wskaźnik `std::auto_ptr`). Tego typu narzędzia w znaczącym stopniu ułatwiają refaktoryzację (ang. *refactoring*), czyli proces wprowadzania w istniejącym kodzie źródłowym zmian, których celem jest zwiększenie jego jakości bez zmiany funkcjonalności. Przykładem bardzo popularnego wśród programistów C++ narzędzia przeprowadzającego statyczną analizę kodu źródłowego jest Clang-Tidy, oparty na kompilatorze Clang. Ponadto nowoczesne środowiska programistyczne bardzo często posiadają zintegrowaną funkcjonalność pozwalającą przeprowadzić tego typu badanie. Tego typu środowiskiem jest, wykorzystywany przez autorów do pracy nad systemem GGSS, CLion stworzony przez firmę JetBrains.

2.4. System kontroli wersji Git (JC)

2.5. Portal GitLab (JC)

2.6. Narzędzie CMake (AK)

CMake jest rozwijanym przez firmę *Kitware* narzędziem automatyzującym procesy związane z cyklem życia oprogramowania, w tym przede wszystkim jego budowanie, testowanie, instalację oraz tworzenie pakietów. Udostępnia intuicyjny, oparty o prosty język skryptowy interfejs, umożliwiający tworzenie konfiguracji w sposób niezależny od docelowej platformy, dzięki czemu możliwe jest konstruowanie zaawansowanych, dostosowanych do potrzeb konkretnego projektu systemów budowania. Stanowi trzon systemu budowania i pakietowania przygotowanego na potrzeby systemu GGSS przez autorów niniejszego manuskryptu w ramach napisanej przez nich pracy inżynierskiej. Z narzędziem CMake ściśle zintegrowane są systemy CPack oraz CTest, który zadaniem jest kolejno: tworzenie pakietów instalacyjnych z oprogramowaniem (np. `.rpm`) oraz tworzenie konfiguracji umożliwiających testowanie automatyczne.

Na listingu 2.7 przedstawiony został bardzo prosty przykład pliku `CMakeLists.txt`, zawierającego konfigurację pozwalająca na zbudowanie aplikacji napisanej w języku C++. Na załączonym fragmencie kodu widoczne są komendy pozwalające określić informacje takie jak: minimalna wersja narzędzia CMake, nazwa oraz wersja projektu czy lista plików wchodzących w skład tworzonej aplikacji.

Listing 2.7. Przykład pliku `CMakeLists.txt`, zawierającego komendy pozwalające na zbudowanie prostej, składającej się z jednego pliku źródłowego, aplikacji napisanej w języku C++

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

# Określenie nazwy i wersji projektu oraz stosowanego języka.
project(SampleApp VERSION 1.0.0 LANGUAGES CXX)

# Dodanie docelowego pliku wykonywanego.
add_executable(Sample main.cpp)
```

2.7. Menadżer pakietów RPM (JC)

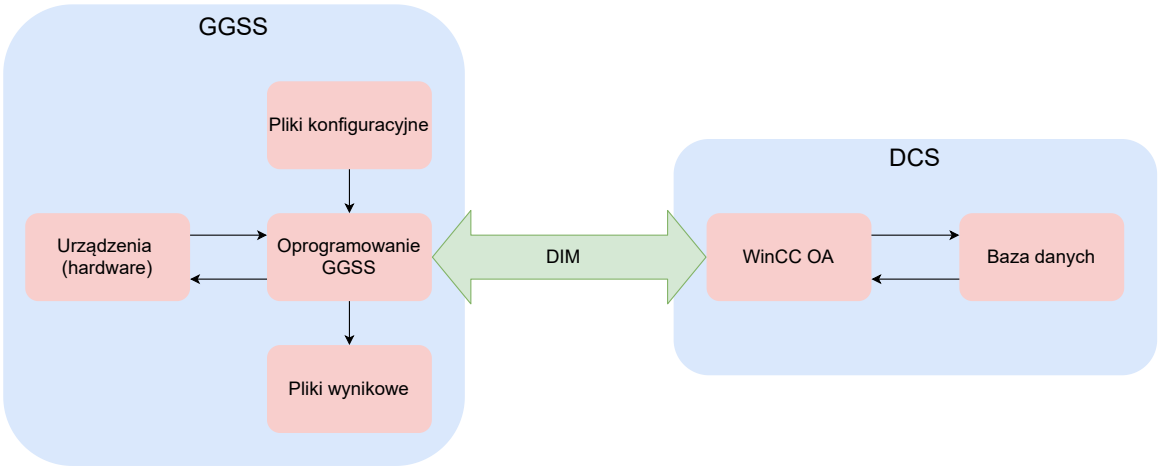
3. Budowa i działanie systemu GGSS (AK)

Niniejszy rozdział zawiera ważne, z punktu widzenia przeprowadzonych prac, informacje na temat systemu GGSS. Przedstawione tu opisy dotyczą zagadnień takich jak: wysokopoziomowa architektura systemu, struktura warstwy oprogramowania, opis wykorzystywanych przez system urządzeń oraz omówienie cech charakterystycznych środowiska docelowego.

3.1. Wysokopoziomowa architektura systemu GGSS

System GGSS składa się z kilku współpracujących ze sobą elementów, przedstawionych (wraz z występującymi między nimi interakcjami) na rysunku 3.1. Znaczenie poszczególnych komponentów projektu jest następujące:

- **urządzenia (ang. *hardware*)** - zestaw urządzeń elektronicznych (m.in. liczniki słomkowe, zasilacze wysokiego napięcia i multipleksery)
- **oprogramowanie GGSS** - zestaw aplikacji wraz z otaczającą je infrastrukturą, których zadaniem jest sterowanie urządzeniami wchodzącymi w skład systemu GGSS oraz przetwarzanie zbieranych za ich pomocą danych
- **pliki konfiguracyjne** - proste pliki tekstowe w formacie XML (*Extensible Markup Language*), zawierające informacje o oczekiwanym sposobie działania systemu (np. maksymalna możliwa wartość napięcia, jakie może zostać ustawione na każdym z zasilaczy)
- **pliki wynikowe** - pliki tekstowe zawierające wyniki pomiarów wykonywanych przez system oraz rejestr zdarzeń
- ***SIMATIC WinCC Open Architecture*** - system typu SCADA (*Supervisory Control And Data Acquisition*), stanowiący część systemu kontroli detektora ATLAS, pozwalający na obserwację i kontrolę działania poszczególnych poddetektorów
- ***Distributed Information Management System (DIM)*** - protokół komunikacyjny dla środowisk rozproszonych, oparty o architekturę klient-serwer, zapewniający komunikację między oprogramowaniem systemu GGSS a systemem WinCC OA



Rys. 3.1. Wysokopoziomowa architektura projektu GGSS. Strzałkami oznaczono przepływ danych pomiędzy poszczególnymi komponentami systemu.

Szczegóły działania najważniejszych z punktu widzenia niniejszej pracy elementów systemu omówione zostaną w dalszej części tego rozdziału. Znaczna część prac opisanych w niniejszym manuskrypcie skupiona była na udoskonaleniu warstwy oprogramowania systemu GGSS.

3.2. Urządzenia elektroniczne

Z punktu widzenia warstwy sprzętowej system GGSS składa się z zestawu tzw. słomkowych liczników proporcjonalnych, zasilanych za pomocą 4-kanałowych zasilaczy wysokiego napięcia. Sygnały generowane przez liczniki przetwarzane są przez wielokanałowy analizator amplitudy (MCA - *Multi-Channel Analyzer*), natomiast wybór licznika słomkowego używanego do wykonania pomiarów następuje za pomocą 8-kanałowego multipleksera sygnałów analogowych. Urządzenia podłączone są do komputera PC, który steruje nimi za pomocą oprogramowania systemu GGSS. W tabeli 3.1 zamieszczone zostało zestawienie informacji na temat wykorzystywanych przez projekt urządzeń. Sposób działania systemu (jego podstawa fizyczna oraz znaczenie przeprowadzanych pomiarów) wykracza poza zakres niniejszej pracy, został natomiast szczegółowo opisany w pracy *Wybrane zagadnienia związane z pracą słomkowych liczników proporcjonalnych w detektorze TRT eksperymentu ATLAS*, której autorem jest dr hab. inż. Bartosz Mindur, prof. AGH.

Tabela 3.1. Zestawienie istotnych z punktu widzenia niniejszej pracy urządzeń wchodzących w skład systemu GGSS.

Urządzenie	Informacje
4-kanałowy zasilacz wysokiego napięcia	CAEN N1470
wielokanałowy analizator amplitudy	CAEN N957
multiplekser sygnałów analogowych	urządzenie autorstwa Pana Pawła Zadrożniaka

3.3. Warstwa oprogramowania

Poprzez warstwę oprogramowania systemu GGSS autorzy rozumieją zarówno zestaw aplikacji napisanych w języku C++ (standard 11), jak i otaczającą je infrastrukturę (pomocnicze skrypty, system budowania, testowania i tworzenia nowych wydań).

Trzon warstwy oprogramowania projektu GGSS stanowi aplikacja *ggss-runner*, zawierająca logikę odpowiedzialną za komunikację z systemem za pomocą protokołu DIM, gromadzenie i walidację danych oraz sterowanie urządzeniami wchodzącymi w skład warstwy sprzętowej. W skład systemu wchodzi ponadto szereg pomniejszych aplikacji (niektóre z nich stanowią element dodany przez autorów niniejszej pracy, zostaną więc omówione ze szczegółami w dalszych jej częściach):

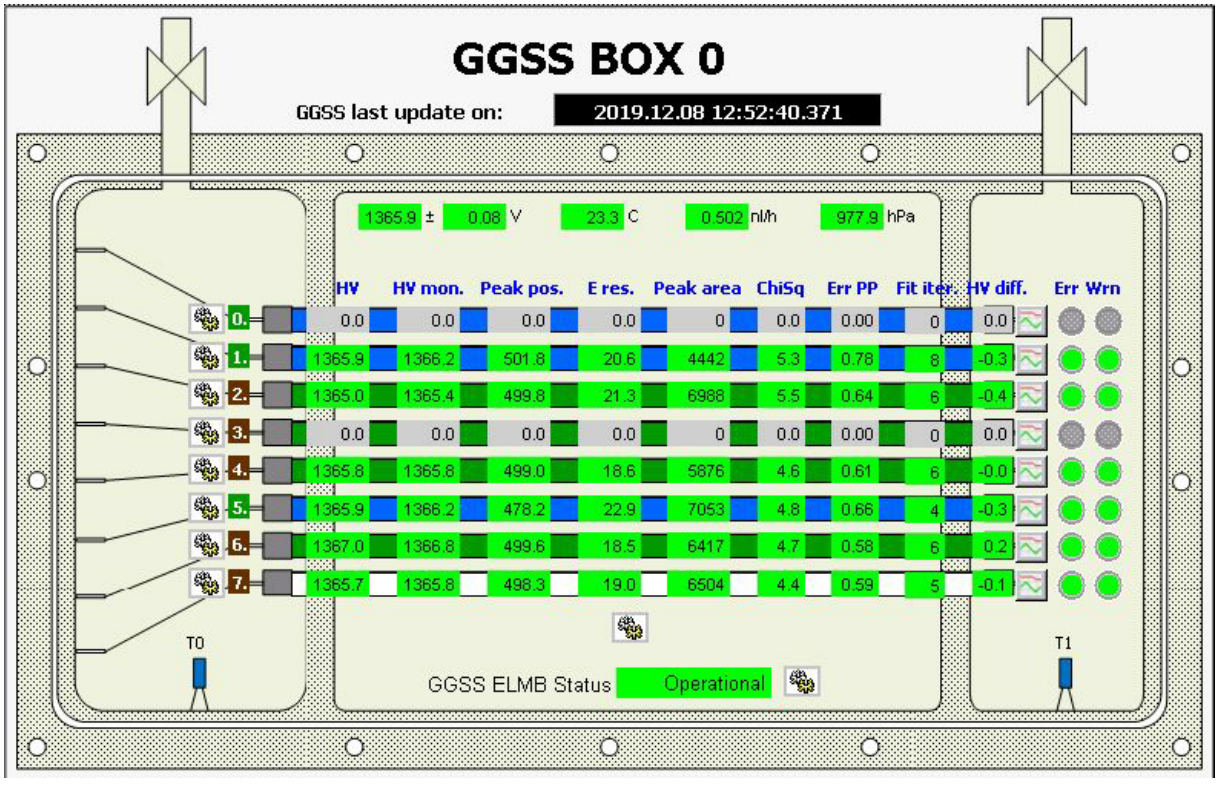
- *ggss-spector* - aplikacja okienkowa służąca do wizualizacji zebranych przez system danych (zapisanych w plikach wynikowych)
- *ggss-reader* - niezależna aplikacja przeznaczona do wykorzystywania na maszynach deweloperskich, pozwalająca na odtwarzanie działania oprogramowania sterującego GGSS, tzn. wysyłająca do systemu kontroli detektora archiwalne dane z pominięciem warstwy sprzętowej
- *ggss-dim-cs* - aplikacja pozwalająca na prowadzenie interakcji z systemem poprzez udostępnienie możliwości wysyłania do niego komend za pomocą protokołu DIM
- zestaw aplikacji *ggss-hardware-service-apps* - proste narzędzia pozwalające na wykonywanie operacji na wchodzących w skład systemu urządzaniach, w tym na wykonywanie testów ich działania.

Projekt GGSS charakteryzuje się ponadto rozbudowaną infrastrukturą, w której skład wchodzą systemy odpowiedzialne za budowanie projektu, zarządzanie zależnościami zewnętrznymi oraz pomiędzy jego komponentami, automatyzację procesu testowania poszczególnych komponentów oraz automatyzację tworzenia i wersjonowania wydań. Projekt zawiera ponadto skrypty pomocnicze (napisane przy użyciu popularnych języków skryptowych), pozwalające na zarządzanie systemem w jego środowisku docelowym. Gruntowna przebudowa infrastruktury systemu

GGSS stanowiła temat pracy inżynierskiej autorów. W dalszej części niniejszego manuskryptu omówione zostaną wprowadzone w ramach pracy magistersiej rozszerzenia.

3.4. Oprogramowanie WinCC OA

SIMATIC WinCC Open Architecture jest oprogramowaniem typu SCADA firmy SIEMENS służącym do wizualizacji i sterowania procesami produkcyjnymi. Stanowi ono trzon systemu kontroli detektora ATLAS i pozwala na monitorowanie i sterowanie pracą wchodzących w jego skład podsystemów. WinCC OA pozwala m.in. na tworzenie specjalnych paneli, przedstawiających w przyjaznej dla użytkownika formie graficznej zebrane dane oraz procesy zachodzące w monitorowanym systemie - przykład tego typu panelu, obrazujący pracę słomkowych liczników proporcjonalnych wchodzących w skład warstwy sprzętowej systemu GGSS, przedstawiony został na rysunku 3.2.



Rys. 3.2. Fragment przykładowego panelu informacyjno-administracyjnego stworzonego z wykorzystaniem technologii WinCC OA. Widoczne są m.in.: parametry związane z pomiarami wykonywanymi za pomocą słomkowych liczników proporcjonalnych (np. *Peak pos.* i *Peak area*), data ostatniej aktualizacji oraz wskaźniki informujące o ostrzeżeniach i błędach.

Autorzy niniejszego dokumentu nie byli odpowiedzialni za przeprowadzanie prac związanych z rozwojem oraz utrzymanem systemu WinCC OA funkcjonującego w ramach infrastruktury CERN. Z tego też powodu szczegóły jego działania nie zostaną omówione. Istotna, z punktu

widzenia niniejszej pracy, jest natomiast możliwość zastosowania go jako narzędzia ułatwiajacego przeprowadzanie okresowych testów systemu GGSS. Wynika to przede wszystkim z wygodnej w użytkowaniu funkcjonalności paneli, pozwalających na monitorowanie działania projektu w czasie rzeczywistym oraz natychmiastowe wykrywanie wszelkich nieprawidłowości.

3.5. Środowisko docelowe i ograniczenia

Charakterystyka środowiska docelowego, w jakim działa system GGSS, jest z punktu widzenia niniejszej pracy bardzo istotna, przede wszystkim ze względu na bardzo znaczący związek projektu z infrastrukturą dostarczaną przez CERN. Stawia to przed autorami szereg szereg ograniczeń dotyczących wersji wykorzystywanych narzędzi, jak również wymusza dodatkowe działania w przypadku wykonywania pewnych operacji. Do najważniejszych ograniczeń narzucanych przez środowisko docelowe i specyfikę projektu należą:

- dostępna wersja kompilatora języka C++ - w ramach infrastruktury CERN dostępny jest kompilator *g++ (GCC) 4.8.5*. Wersja ta wspiera w większości standard C++11, a zatem funkcjonalności takie jak wyrażenia lambda czy semantyka przenoszenia. Niestety oferowane przez nią wsparcie nie jest pełne - brakuje m.in. poprawnej implementacji biblioteki odpowiedzialnej za przetwarzanie wyrażeń regularnych. Ze względu na wymóg zapewnienia możliwości budowania projektu na maszynie docelowej, ograniczenie to stanowiło znaczące utrudnienie podczas prac nad kodem źródłowym aplikacji wchodzących w skład systemu.
- dostępna wersja narzędzia CMake - na maszynach docelowych dostępna jest wersja *2.8.12.2*, stanowiąca bardzo stare wydanie narzędzia. Oprogramowanie w znacząco nowszej wersji (tzn. o numerze wyższym od *3.0*) dostępne jest jedynie na wybranych komputerach wchodzących w skład infrastruktury CERN, przez co zdecydowano o pozostaniu przy starym jego wydaniu. Stosowana wersja nie zawiera wielu powszechnie stosowanych współcześnie funkcjonalności oraz charakteryzuje się innym podejściem do zarządzania zależnościami (operacje na poziomie katalogów, uznawane za tzw. *złą praktykę*).
- związek projektu z wersją jądra systemu - jednym z modułów wchodzących w skład systemu GGSS jest *ggss-driver*, zawierający sterownik dla wielokanałowego analizatora amplitudy CAEN N957. Istnienie tego modułu wymusza zgodność wersji jądra systemu operacyjnego pomiędzy środowiskiem deweloperskim i produkcyjnym, co w konsekwencji prowadzi do komplikacji infrastruktury budowania projektu - konieczne jest stosowanie maszyn wirtualnych oraz narzędzia konteneryzacyjnego Docker podczas procesu budowania komponentu *ggss-driver* (stosowane rozwiązanie opisane zostało przez autorów szczegółowo w ich pracy inżynierskiej).
- ograniczone uprawnienia w środowisku docelowym - infrastruktura na której uruchamiany jest projekt GGSS jest środowiskiem CERN o zaostrzonym rygorze. Wszelkie instalowane aplikacje, zmiany w systemie, bibliotekach, czy też prostych ustawieniach użytkownika

muszą być konsultowane z administratorami systemowymi. Autorzy nie mają możliwości wprowadzania na własną rękę praktycznie żadnych zmian.

– możliwość przeprowadzania testów tylko w określonych momentach prac nad projektem - nad systemami GGSS oraz DCS pracuje wielu ekspertów, testowanie projektu możliwe jest zatem tylko wtedy, gdy nie zakłóca to prac innych osób i jest fizycznie możliwe (np. gdy nie są wykonywane prace nad warstwą sprzętową systemu). Wymusza to dostosowanie tempa prac w taki sposób, by jednocześnie testowany był ograniczony, ale znaczący zakres zmian (m.in. by możliwe było szybkie wprowadzenie poprawek w przypadku wykrycia błędu).
– konieczność zachowania kompatybilności wstecznej - zmiany wprowadzane w systemie nie mogą powodować, że starsze wersje komponentów, z jakich składa się system GGSS (rys. 3.1) staną się niezdatne do użycia, np.: dodanie nowego parametru do pliku konfiguracyjnego nie powinno wykluczać możliwości użycia starszej wersji tegoż pliku oraz starszej wersji oprogramowania. Tego typu ograniczenia obowiązują również w kontekście danych wymienianych pomiędzy aplikacją GGSS a systemem kontroli detektora za pomocą protokołu DIM - dane mają odgórnie ustalony, niemożliwy do zmiany format.

4. Prace nad architekturą i infrastrukturą projektu (AK i JC)

Niniejszy rozdział zawiera opis prac wykonanych przez autorów w ramach rozwoju architektury i infrastruktury systemu GGSS. Rozdział ten stanowi bezpośrednią kontynuację pracy inżynierskiej autorów, gdzie przygotowane zostały pierwsze wersje rozwijanych w ramach pracy magisterskiej rozwiązań. Przedstawione tu informację dotyczą szerokiego zakresu zagadnień związanych z inżynierią oprogramowania, takich jak: zarządzanie strukturą projektu oraz jego zależnościami, automatyzacja procesów towarzyszących wytwarzaniu oprogramowania czy przygotowanie infrastruktury ułatwiającej testy warstwy sprzętowej systemu.

4.1. Zmiany w architekturze projektu (AK)

Przez zmiany w architekturze projektu autorzy rozumieją stopniowy rozwój zaimplementowanego przez nich w ramach pracy inżynierskiej rozwiązania. Rozwój ten obejmuje przede wszyskim uproszczenie powstałej hierarchii zależności między poszczególnymi elementami warstwy oprogramowania (rozumianymi zarówno jako repozytoria, jak i biblioteki), uczynienie systemu bardziej przystępnym dla użytkownika (np. poprzez nadanie komponentom nazw dobrze oddających ich przeznaczenie) oraz przygotowanie systemu pozwalającego w prosty sposób odtworzyć kod źródłowy w wersji bez wprowadzonych w ramach pracy magisterskiej modyfikacji (jako rodzaj zabezpieczenia przed skutkami potencjalnych błędów, które mogły zostać wprowadzone do oprogramowania podczas prac nad nim). Znaczna część zmian opisanych w niniejszej części pracy była możliwa do wprowadzenia z uwagi na trwające jednocześnie prace nad kodem źródłowym systemu GGSS i zmiany zachodzące w ich czasie.

4.1.1. Wprowadzenie do problematyki

Przeprowadzone przez autorów w ramach pracy inżynierskiej modyfikacje architektury systemu GGSS obejmowały przede wszystkim migrację projektu do systemu kontroli wersji Git, wprowadzenie spójnego nazewnictwa poszczególnych komponentów oraz zastosowanie funkcjonalności submodułów będącej częścią technologii Git do stworzenia hierarchicznej struktury repozytoriów (w odróżnieniu od pierwotnej, płaskiej architektury opartej o katalogi). Celem

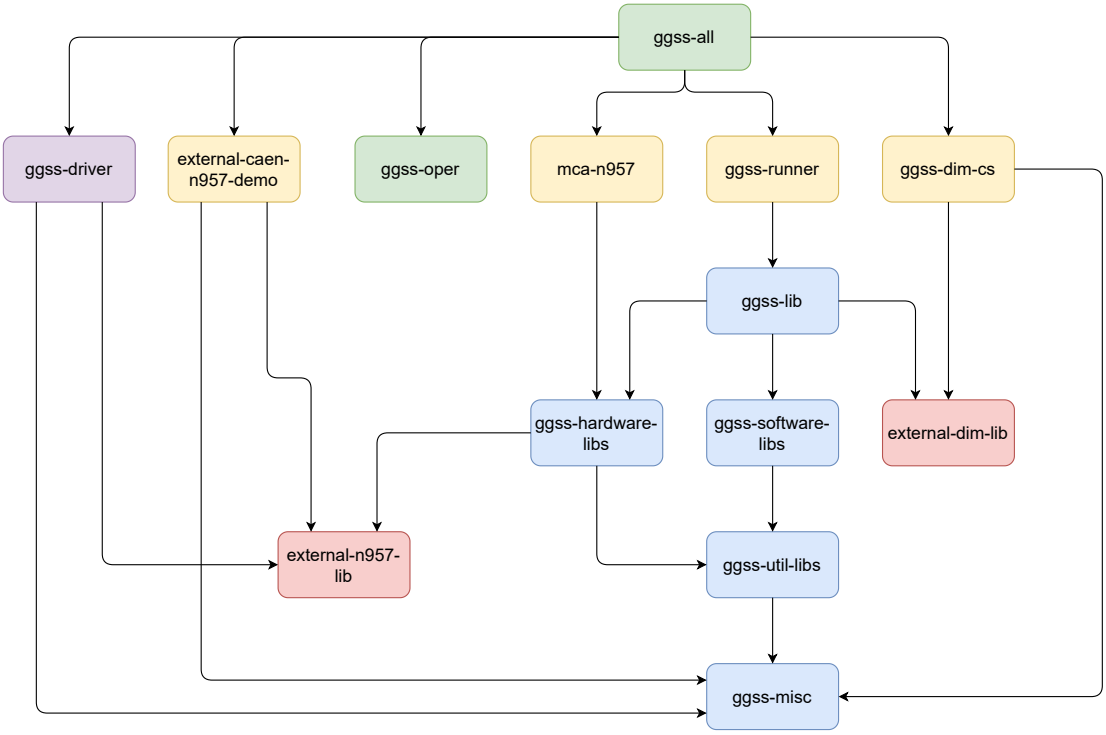
tych zmian było ułatwienie pracy nad pojedynczymi komponentami projektu oraz uczynienie struktury projektu przyjazną dla użytkownika, co zostało zdaniem autorów osiągnięte.

Architektura stanowiąca punkt wyjściowy zmian wykonanych w ramach niniejszej pracy przedstawiona została na rysunku 4.1 (z pominięciem repozytoriów pomocniczych, zawierających np. dokumentację). Zawarte na schemacie kolory obrazują rolę każdego modułu (zielony - repozytorium pomocnicze, czerwony - zależność zewnętrzna, żółty - aplikacja, fioletowy - sterownik urządzenia oraz niebieski - kod źródłowy oraz pliki nagłówkowe bibliotek). Projekt składał się zatem z 14 tworzących strukturę hierarchiczną repozytoriów, zawierających elementy takie jak: aplikacje, pomocnicze skrypty, infrastruktura budowania oraz kod źródłowy bibliotek implementujących poszczególne funkcjonalności systemu. W kontekście tej części pracy szczególnie istotne są repozytoria zawierające kod źródłowy bibliotek statycznych oraz pliki nagłówkowe, stanowiące trzon projektu (tzn. wykorzystywane przez aplikację *ggss-runner*): *ggss-lib*, *ggss-software-libs*, *ggss-hardware-libs*, *ggss-util-libs* oraz *ggss-misc* (repozytoria te oznaczone zostały na rys. 4.1 kolorem niebieskim). Ich rola w pierwotnej wersji projektu prezentowała się następująco:

- ***ggss-hardware-libs*** - przechowywanie bibliotek odpowiedzialnych za obsługę urządzeń wchodzących w skład warstwy sprzętowej systemu GGSS. W pierwotnej wersji projektu były to następujące biblioteki statyczne:
  - *caenhv-lib* oraz *caenn1470-lib* - odpowiedzialne za komunikację z zasilaczami wysokiego napięcia CAEN N1470
  - *mca-lib* oraz *ortecmcb-lib* - odpowiedzialne za obsługę wielokanałowego analizatora amplitudy CAEN N957
  - *usbrm-lib* - odpowiedzialna za obsługę multipleksera sygnałów analogowych
- ***ggss-software-libs*** - przechowywanie bibliotek odpowiedzialnych za implementację wykorzystywanych przez system algorytmów i struktur danych związanych ściśle z warstwą oprogramowania (tzn. nie mających związku z warstwą sprzętową). W pierwotnej wersji projektu były to następujące biblioteki statyczne:
  - *xml-lib* - odpowiedzialna za implementację operacji odczytu oraz zapisu plików w formacie XML oraz operacji na strukturze drzewiastej powstałej w wyniku sparsowania zapisanych w tym formacie danych.
  - *fifo-lib* - odpowiedzialna za implementację prostej struktury danych, stanowiącej kolejkę typu FIFO (*First In, First Out*) o ograniczonym rozmiarze.
  - *fit-lib* - odpowiedzialna za implementację operacji wykonywanych na zebranych przez system danych, w tym przede wszystkim za mechanizm dopasowania do nich krzywej.
  - *daemon-lib* - odpowiedzialna za implementację mechanizmu pozwalającego uruchomić aplikację *ggss-runner* jako tzw. demon (ang. *daemon*) - usługa działająca „w tle”
- ***ggss-util-libs*** - przechowywanie bibliotek, od których zależne są zarówno komponenty odpowiedzialne za obsługę warstwy sprzętowej projektu, jak i związane wyłącznie z warstwą

oprogramowania. Innymi słowy, były to biblioteki wykorzystywane przez zawartość obu wyżej wymienionych repozytoriów, a zatem nie mogące znaleźć się w żadnym z nich. W pierwotnej wersji projektu były to następujące biblioteki statyczne:

  - *log-lib* - odpowiedzialna za implementację mechanizmu dziennika zdarzeń, zapisującego w plikach `.log` informacje o zdarzeniach mających miejsce w systemie
  - *utils-lib* - odpowiedzialna za implementację pomniejszych funkcjonalności, takich jak konwersja między łańuchem znakowym a liczbą (przed pojawieniem się standardu C++11 tego typu funkcjonalności nie były częścią biblioteki standardowej)
  - *handle-lib* - odpowiedzialna za implementację wykorzystywanego w projekcie mechanizmu slotów i sygnałów
  - *thread-lib* - odpowiedzialna za implementację wykorzystywanego w projekcie mechanizmu wielowątkowości
- ***ggss-misc*** - przechowywanie plików nagłówkowych (niebędących cześcią żadnej z bibliotek statycznych) oraz plików `.cmake` tworzących infrastrukturę budowania projektu
- ***ggss-lib*** - przechowywanie kodu źródłowego zawierającego główną logikę systemu GGSS oraz przesyłane za pomocą protokołu DIM struktury danych



Rys. 4.1. Architektura projektu przed wprowadzeniem modyfikacji (sytuacja wyjściowa). Groty strzałek wskazują repozytoria bazowe, kolory natomiast opisują rolę poszczególnych modułów: zielony oznacza repozytoria pomocnicze, żółty - aplikacje, czerwony - biblioteki zewnętrzne, fioletowy - sterownik a niebieski - biblioteki i pliki nagłówkowe projektu GGSS.

4.1.2. Motywacja do wprowadzenia zmian

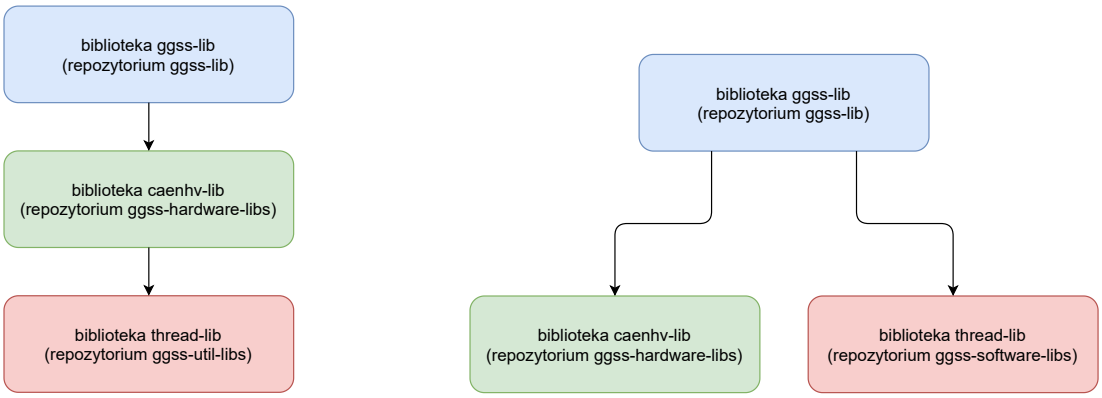
Przygotowane przez autorów w ramach pracy inżynierskiej rozwiązanie było w pełni funkcjonalne, charakteryzowało się jednak pewnymi wadami i ograniczeniami, wynikającymi przede wszystkim z ograniczeń czasowych, niewielkiego doświadczenia autorów w pracy z projektem oraz istniejącego wtedy założenia o niemodyfikowaniu kodu źródłowego aplikacji i bibliotek wchodzących w skład projektu. Najważniejsze z występujących w tym rozwiązaniu problemów to:

- głęboka hierarchia zależności, mająca negatywny wpływ na wydajność działania mechanizmu submodułów
- istnienie repozytorium *ggss-misc*, zawierającego (poza szablonami CMake) elementy kodu źródłowego niepasujące do pozostałych bibliotek wchodzących w skład systemu: bazowe klasy wyjątków stosowanych w całym projekcie oraz flagi konfigurujące projekt w zależności od systemu operacyjnego (konieczność zastosowania tego typu zabiegu wynikła wprost z założenia o niemodyfikowaniu kodu źródłowego w czasie tworzenia pracy inżynierskiej)
- zachowanie oryginalnych nazw bibliotek i aplikacji, dostosowując je jedynie do przyjętej konwencji. Jedną z bibliotek wchodzących w skład projektu była biblioteka statyczna *handle-lib*, odpowiedzialna za implementację mechanizmu slotów i sygnałów, na co, zdaniem autorów, jej nazwa nie wskazuje.
- wnioskowanie o zależnościach pomiędzy bibliotekami na podstawie dyrektyw preprocesora *include* zawartych w kodzie źródłowym, a nie wykorzystywanych funkcjonalności, co wynikało z niewielkiego doświadczenia i wiedzy autorów na temat systemu podczas tworzenia pracy inżynierskiej oraz wspomnianego już założenia o niemodyfikowaniu kodu źródłowego.
- założenie o tworzeniu oddzielnego repozytorium dla każdej z występujących w projekcie aplikacji, niezależnie od jej rozmiarów, co ostatecznie znacznie skomplikowało powiązania pomiędzy repozytoriami (np. repozytoria *external-caen-n957-demo* oraz *mca-n957* charakteryzują się podobnymi zależnościami i oba zawierają niewielkie aplikacje, których zadaniem jest współpraca z wielokanałowym analizatorem amplitudy CAEN N957 - mogłoby być więc połączone w jedno repozytorium).
- brak łatwego sposobu na odtworzenie pierwotnej postaci kodu źródłowego - mechanizm ten nie był potrzebny na etapie pracy inżynierskiej, ponieważ nie dokonywano wtedy modyfikacji we wspomnianym kodzie.

4.1.3. Uproszczenie architektury projektu

Pierwszym podjętym przez autorów działaniem mającym na celu modyfikację struktury projektu była próba jej uproszczenia poprzez analizę zależności wewnętrzych systemu (tzn. zależności pomiędzy poszczególnymi bibliotekami). Prowadzone równolegle prace nad kodem źródłowym projektu pozwoliły autorom zaobserwować, iż pewna część występujących w nim dyrektyw

preprocesora `#include` nie oddaje w poprawny sposób faktycznej struktury zależności między bibliotekami. Najważniejszy przykład stanowi łańcuch zależności występujących pomiędzy biblioteką *ggss-lib*, a bibliotekami *caenhv-lib* oraz *thread-lib*. W oryginalnej wersji projektu zależności między wymienionymi komponentami prezentowały się tak, jak na rysunku 4.2a, tzn. bibliteka *ggss-lib* zależna była od biblioteki *caenhv-lib*, która natomiast zawierała dyrektywę `#include` dołączającą plik nagłówkowy z biblioteki *thread-lib*.



(a) Oryginalna struktura.

(b) Struktura po wprowadzeniu zmian.

Rys. 4.2. Zestawienie oryginalnej oraz nowej struktury zależności pomiędzy bibliotekami *ggss-lib*, *caenhv-lib* oraz *thread-lib*. Groty strzałek wskazują w stronę modułów bazowych.

W rzeczywistości biblioteka *caenhv-lib* nie wykorzystywała zawartości wspomnianego pliku nagłówkowego - pełniła jedynie formę swego rodzaju pośrednika, udostępniając znajdujące się tam klasy bibliotece *ggss-lib*. Przeniesienie dyrektywy `#include` do biblioteki *ggss-lib* spowodowało, iż żadna z bibliotek wchodzących w skład repozytorium *ggss-hardware-libs* nie zawierała zależności do biblioteki *thread-lib*. Rozwiązanie to pozwoliło dokonać migracji tejże biblioteki, wraz z wykorzystywaną przez nią biblioteką *handle-lib*, do repozytorium *ggss-software-libs*, redukując tym samym liczbę bibliotek znajdujących się w repozytorium *ggss-util-libs*. Rysunek 4.2b przedstawia w sposób schematyczny strukturę otrzymanego rozwiązania.

W związku z opisanymi powyżej zmianami ilość kodu źródłowego znajdującego się w repozytorium *ggss-util-libs* znacznie spadła - pozostałe tam biblioteki *log-lib* oraz *utils-lib* charakteryzowały się niewielkim rozmiarem. Spowodowało to, iż jednoczesne istnienie modułów *ggss-misc* oraz *ggss-util-libs* (po wprowadzonych zmianach spełniających tą samą rolę przechowywania niewielkiej liczby komponentów wykorzystywanych przez wiele modułów projektu GGSS) przestało być uzasadnione. Kolejny etap wykonanych prac stanowiło więc przeprowadzenie integracji tychże repozytoriów - w tym celu zdecydowano się na likwidację modułu *ggss-misc* po wcześniejszym przeniesieniu jego zawartości do *ggss-util-libs*.

Migracja znajdujących się w repozytorium *ggss-misc* plików `.cmake` (modułów wykorzystywanych przez infrastrukturę budowania projektu) wymagała, poza wykonaniem trywialnej czynności przeniesienia katalogu, aktualizacji (na poziomie całego projektu) ścieżek wskazujących lokalizację tychże plików. Działanie to było konieczne, ponieważ narzędzie CMake wymaga od programisty, by wyspecyfikował on lokalizację modułów `.cmake` dołączanych do projektu (np. za pomocą komendy `include()`) poprzez dodanie ścieżki z ich lokalizacją do listy `CMAKE_MODULE_PATH` (przykład wykorzystania tejże listy przedstawiony został na listingu 4.1). Oznaczało to więc konieczność wykonania, w każdym module wykorzystującym pliki `.cmake`, zmiany wspomnianej ścieżki tak, by wskazywała na katalog *cmake-templates* w repozytorium *ggss-util-libs*.

Listing 4.1. Przykładowy fragment pliku `CMakeLists.txt`, obrazujący sposób użycia listy `CMAKE_MODULE_PATH` w celu wskazania lokalizacji plików zawierających często wykorzystywane w projekcie, pomocnicze funkcje.

```
# Przypisanie pojedynczej wartości (zawierającej ścieżkę do katalogu
# cmake-templates, w którym znajdują się wykorzystywane w projekcie
# pliki .cmake) do listy CMAKE_MODULE_PATH
set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/../ggss-util-libs/cmake-templates")

# Dołączenie znajdujących się w katalogu cmake-templates plików .cmake
include(BuildStaticLibrary)     # ggss_build_static_library
include(SetupTests)             # ggss_setup_tests

# Wykorzystanie znajdującej się w pliku .cmake funkcji
ggss_build_static_library(
    TARGET_NAME "fifo"
)
```

Poza wspomnianymi plikami `.cmake` w repozytorium *ggss-misc* znajdował się katalog `include`, zawierający trzy pliki nagłówkowe z kodem napisanym w języku C++:

- pliki `ggssExceptions.h` oraz `HardwareException.h` zawierające klasy bazowe wyjątków wykorzystywanych w całym projekcie GGSS
- plik `CompatibilityFlags.h`, zawierający flagi konfigurujące projekt w zależności od platformy docelowej (Windows lub Linux)

Pliki te nie wchodziły oryginalnie w skład żadnej z bibliotek projektu GGSS, nie mogły zostać do nich również dodane przez autorów podczas przygotowywania pracy inżynierskiej, ponieważ wymagałoby to modyfikacji kodu źródłowego systemu. Podczas przeprowadzanej w ramach niniejszej pracy migracji tych plików do repozytorium *ggss-util-libs* zdecydowano się na likwidację katalogu `include` i rozdysponowanie jego zawartości do istniejących lub nowych bibliotek. Plik `CompatibilityFlags.h` przeniesiony został więc do biblioteki *utils-lib*, natomiast na potrzebę dwóch pozostałych nagłówków przygotowana została nowa biblioteka *exceptions-lib*.

Finalna struktura repozytorium *ggss-util-libs* przedstawiona została na listingu 4.2. Poza wspomnianymi do tej pory zmianami nowość stanowi katalog `doxygen-config`, zawierający prosty plik konfigurujący działanie narzędzia Doxygen służącego do generowania dokumentacji programów napisanych w języku C++. Rozszerzenie projektu o możliwość generowania dokumentacji zostanie jednak opisane szczegółowo w dalszej części pracy.

Listing 4.2. Zawartość repozytorium *ggss-util-libs* po wprowadzeniu opisanych zmian. Widoczne są biblioteki wchodzące w skład repozytorium: *exceptions-lib*, *log-lib* oraz *utils-lib*, katalog *cmake-templates* zawierający szablony wykorzystywane przez system budowania, katalog *doxygen-config* zawierający konfigurację narzędzia Doxygen, nadrzędny plik *CMakeLists.txt* służący do budowania wszystkich bibliotek w repozytorium oraz plik *README.md* zawierający opis repozytorium.

```
.
|-- CMakeLists.txt
|-- README.md
|-- cmake-templates
|-- doxygen-config
|-- exceptions-lib
|-- log-lib
`-- utils-lib
```

Aby zachować możliwość wglądu w historię zmian przeprowadzanych w repozytorium *ggss-misc* zdecydowano, że jego likwidacja polegać będzie na przeprowadzeniu archiwizacji oraz usunięciu z pozostałych modułów odniesień do niego. Udostępniany przez portal GitLab mechanizm archiwizacji polega na oznaczeniu repozytorium jako niemodyfikowalne i przeniesieniu go do osobnej zakładki w widoku organizacji. Dzięki takiemu rozwiązaniu repozytorium wciąć dostępne jest do wglądu (a zatem istnieje możliwość zbudowania tych wersji systemu GGSS, które korzystały z niego jako z submodułu), niemożliwy jest natomiast jego dalszy rozwój.

Poza wspomnianymi do tej pory repozytoriami zmianami objęte zostały ponadto moduły przechowujące aplikacje służące do testowania i obsługi urządzeń elektronicznych wchodzących w skład warstwy sprzętowej systemu GGSS. Motywacją do wprowadzenia modyfikacji była konieczność rozbudowy projektu o kolejne tego typu aplikacje - tworzenie dla każdej z nich osobnego repozytorium znacząco komplikowałoby strukturę projektu. Zdecydowano zatem, iż repozytoria *mca-n957* oraz *external-caen-n957-demo* zostaną dołączone do nowo powstałego repozytorium *ggss-hardware-service-apps*, grupującego niewielkie programy służące do operowania na urządzeniach.

Poza zmniejszeniem progu wejścia do projektu poprzez uczynienie jego struktury prostszą, opisane do tej pory zmiany korzystnie wpłynęły na działanie mechanizmu submodułów systemu Git, na którym oparty został proces zarządzania zależnościami między repozytoriami w projekcie. Redukcja liczby repozytoriów i powiązań między nimi oraz zmniejszenie głębokości drzewa zależności (poprzez likwidację repozytorium *ggss-misc*) miało pozytywny wpływ na wydajność systemu zarządzającego architekturą projektu.

4.1.4. Dodanie możliwości odtworzenia pierwotnej wersji kodu źródłowego

Wprowadzanie zmian w kodzie źródłowym aplikacji, której jedną z najważniejszych cech jest jej niezawodność, stanowi znaczące ryzyko. Tego typu aplikacją jest program *ggss-runner*, stanowiący trzon projektu GGSS, a którego źródła podlegały modyfikacjom w ramach opisanych w niniejszym manuskrypcie prac. Naturalnym było więc stworzenie mechanizmu pozwalającego na stosunkowo łatwy powrót do oryginalnej wersji aplikacji, tzn. takiej niezawierającej opisanych w niniejszej pracy zmian w kodzie źródłowym.

Możliwość odtworzenia pierwotnej wersji kodu źródłowego osiągnięta została poprzez utworzenie, dla każdego repozytorium biorącego udział w procesie budowania aplikacji *ggss-runner*, specjalnej gałęzi nazwanej *legacy*. Gałęzie te zawierają oryginalną wersję kodu źródłowego napisanego w języku C++, natomiast pozostałe elementy (infrastruktura budowania oraz ciągłej integracji i dostarczania) znalazły się tam w swoich najnowszych wersjach, co gwarantuje ich spójność w całym projekcie (a co za tym idzie, mogą być używane w taki sam sposób, jak na gałęzi głównej).

Opisanymi zmianami objęte zostały następujące repozytoria: *ggss-all*, *ggss-runner*, *ggss-lib*, *ggss-software-libs*, *ggss-hardware-libs*, *external-dim-lib*, *external-n957-lib* oraz *ggss-util-libs*. W przypadku repozytorów o nazwach zawierających przedrostek *external-* zmiany te polegały jedynie na utworzeniu nowej gałęzi - zawartość bibliotek zewnętrznych nie była przez autorów pracy modyfikowana.

Ostatecznie więc zbudowanie aplikacji *ggss-runner* w jej pierwotnej wersji jest bardzo proste, z poziomu repozytorium *ggss-all* sprowadza się do wykonania komend zamieszczonych na listingu 4.3. Dodatkowo, pliki *README.md* stanowiące dokumentację poszczególnych repozytorów zostały na gałęziach *legacy* odpowiednio zmodyfikowane, by opisywać obowiązującą tam procedurę budowania projektu oraz zawartość poszczególnych modułów.

Listing 4.3. Komendy pozwalające na pobranie kodu źródłowego oraz zbudowanie aplikacji *ggss-runner* w jej oryginalnej wersji.

```
git clone ssh://git@gitlab.cern.ch:7999/atlas-trt-dcs-ggss/ggss-all.git &&
mkdir ggss-all-build &&
cd ggss-all &&
git checkout legacy &&
git submodule update --init --recursive &&
git submodule foreach --recursive git checkout legacy &&
cd ../ggss-all-build &&
python ../ggss-all/build.py --staticboost --buildtype release
```

4.1.5. Pomniejsze zmiany

Poza do tej pory opisanymi, wykonanych zostało kilka pomniejszych modyfikacji mających na celu szeroko pojętą poprawę jakości struktury projektu. Poniżej krotko opisane zostały trzy

wybrane przez autorów modyfikacje, charakteryzujące się różnym poziomem skomplikowania, ale operujące na poziomie pojedynczych repozytoriów.

4.1.5.1. Archiwizacja repozytorium *ggss-oper*

Jednym z repozytoriów wprowadzonych przez autorów w ramach wykonywania pracy inżynierskiej był moduł *ggss-oper*, zawierający skrypty oraz pliki konfiguracyjne stanowiące znaczną część infrastruktury przeznaczonej do użytkowania wraz z oprogramowaniem GGSS na maszynie docelowej. Zawartość tego repozytorium, nie stanowiąca wkładu wniesionego przez autorów niniejszej pracy w system, obejmowała m.in.:

- pierwsze wersje skryptów służących do przeprowadzania testów urządzeń wchodzących w skład warstwy sprzętowej projektu (napisane z wykorzystaniem języka Python)
- skrypty zarządzające stanem środowiska docelowego (np. ustawiające wymagane zmienne środowiskowe)
- skrypty zarządzające oprogramowaniem systemu GGSS, np. `ggss_monitor.sh` pozwalający na uruchamianie, zatrzymywanie oraz sprawdzanie stanu aplikacji *ggss-runner*

Wraz z postępami prac nad projektem, część z wymienionej powyżej zawartości zastąpiona została przez autorów pracy rozwiązaniami alternatywnymi (np. skrypty służące do przeprowadzania operacji na urządzeniach zastąpione zostały aplikacjami napisanymi w języku C++), pozostałe przeniesione zostały natomiast do repozytorium *ggss-all*. Ostatecznie moduł został więc zarchiwizowany.

4.1.5.2. Utworzenie biblioteki *asyncserial-lib*

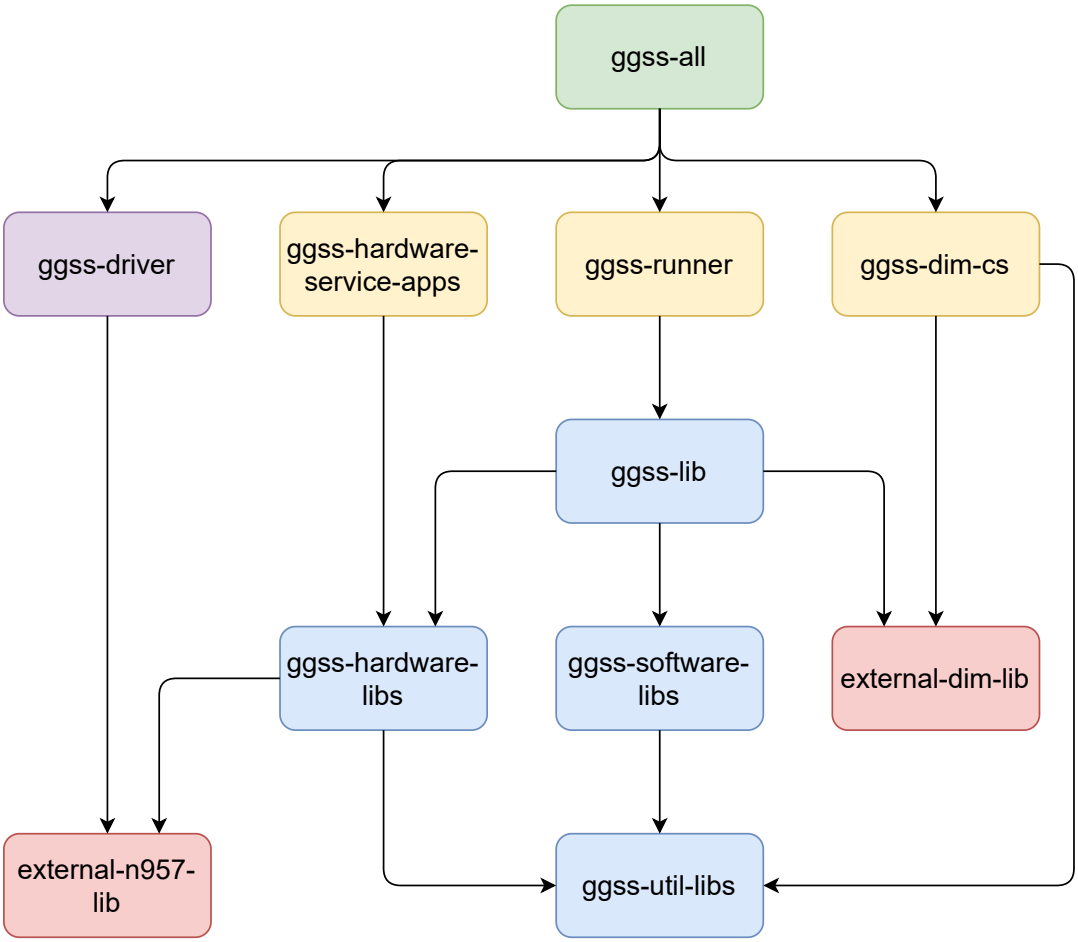
Podczas prac nad kodem źródłowym bibliotek statycznych wchodzących w skład repozytorium *ggss-hardware-libs* zaobserwowano, że w katalogach bibliotek *usbrm-lib* oraz *caenn1470-lib* zamieszczony został, poza właściwym dla nich kodem źródłowym, zestaw plików zawierających implementację asynchronicznej komunikacji z urządzeniami za pomocą interfejsu szeregowego. Ponieważ znalezione w obu przypadkach pliki nie różniły się od siebie, i jednocześnie stanowiły niezbędny element wspomnianych komponentów systemu (zawierały kluczową dla działania projektu funkcjonalność), zdecydowano o utworzeniu nowej biblioteki zawierającej omawiane pliki. Biblioteka nazwana została, zgodnie ze swoim przeznaczeniem, *asyncserial-lib* i weszła w skład repozytorium *ggss-hardware-libs*.

4.1.5.3. Zmiana nazwy biblioteki *handle-lib*

Jedną z bibliotek będących częścią systemu GGSS była biblioteka *handle-lib*, odpowiedzialna za implementację mechanizmu slotów i sygnałów. Oryginalnie biblioteka ta znajdowała się w repozytorium *ggss-util-libs*, jednak wraz z postępem prac przeniesiona została, wraz z biblioteką *thread-lib*, do repozytorium *ggss-software-libs*. Nazwa biblioteki nie pozwalała użytkowniki domyślić się, jakie jest jej zastosowanie - z tego powodu zdecydowano się wprowadzić nową nazwę: *sigslot-lib* (od angielskiego *signals and slots*).

4.1.6. Podsumowanie

W ramach przeprowadzonych prac wykonane zostały zmiany pozwalające na uproszczenie architektury projektu, czyniąc ją przyjaźniejszą dla użytkownika. Finalna struktura przedstawiona została na rysunku 4.3 (podobnie jak w przypadku hierarchii wyjściowej - z pominięciem repozytoriów pomocniczych, nie wchodzących bezpośrednio w jej skład) - aktualnie składa się ona z 11 repozytoriów.



Rys. 4.3. Finalna struktura projektu, po wprowadzeniu wszystkich zmian opisanych w niniejszej pracy. Strzałki wskazują w stronę modułów bazowych. Widoczne jest znaczące uproszczenie struktury projektu względem wersji oryginalnej (rys. 4.1).

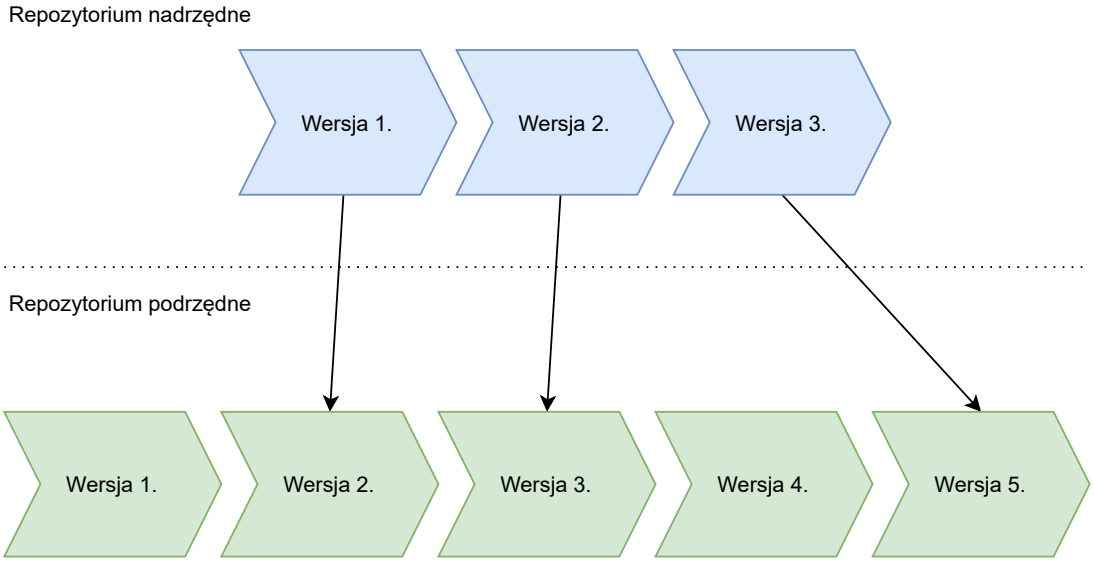
Dla repozytoriów biorących udział w procesie budowania aplikacji *ggss-runner* utworzone zostały ponadto gałęzie *legacy*, zawierające kod źródłowy projektu bez wprowadzonych w ramach niniejszej pracy zmian - dzięki temu możliwy jest stosunkowo łatwy powrót do oryginalnej wersji aplikacji, co stanowi zabezpieczenie na wypadek wprowadzenia do jej źródeł błędów.

4.2. Automatyzacja pracy z submodułami (JC)

Ninejszy rozdział jest poświęcony obsługi wielopoziomowej struktury opartej o *git submodules* obecnej w projekcie GGSS. Przedstawione zostaną plusy oraz minusy zastosowanego w trakcie pracy inżynierskiej rozwiązania. Omówiona zostanie przygotowana przez autorów infrastruktura mająca na celu ułatwienie pracy z submodułami. Dodatkowo krótko zostanie opisane przygotowane *how-to* oraz praktyki które powinno się stosować pracując z taką architekturą.

4.2.1. Wprowadzenie do problematyki

W trakcie pracy inżynierskiej, a konkretnie migracji całego projektu GGSS do systemu kontroli wersji *git* zdecydowano się na wykorzystanie technologii *git submodules*. Ze względu na nacisk na zwiększenie modularyzacji projektu technologia ta idealnie wpasowywała się w docelową architekurę. Zasada działania submodułów jest bardzo zbliżona do dowiązań symbolicznych stosowanych między innymi w systemach UNIX. Zamiast wskazywać na ścieżkę do folderu na lokalnym systemie submoduł wskazuje na ścieżkę do konkretnej wersji repozytorium na zewnętrznym serwerze od którego zależy nasz moduł. Rysunek 4.4 przedstawia zasadę działania submodułów oraz wpływ wersjonowania na tenże mechanizm. Wykorzystanie submodułów pozwala na w pełni odseparowaną pracę nad wybranym komponentem systemu. Nie potrzebujemy pobierać żadnych dodatkowych plików, czy też zależności w celu zmienienia kodu źródłowego. Rozwiązanie to pozwala też na skorzystanie z bardzo szybkiej inicjalizacji całego projektu jedną komendą, co zostało przedstawione w listingu 4.4.



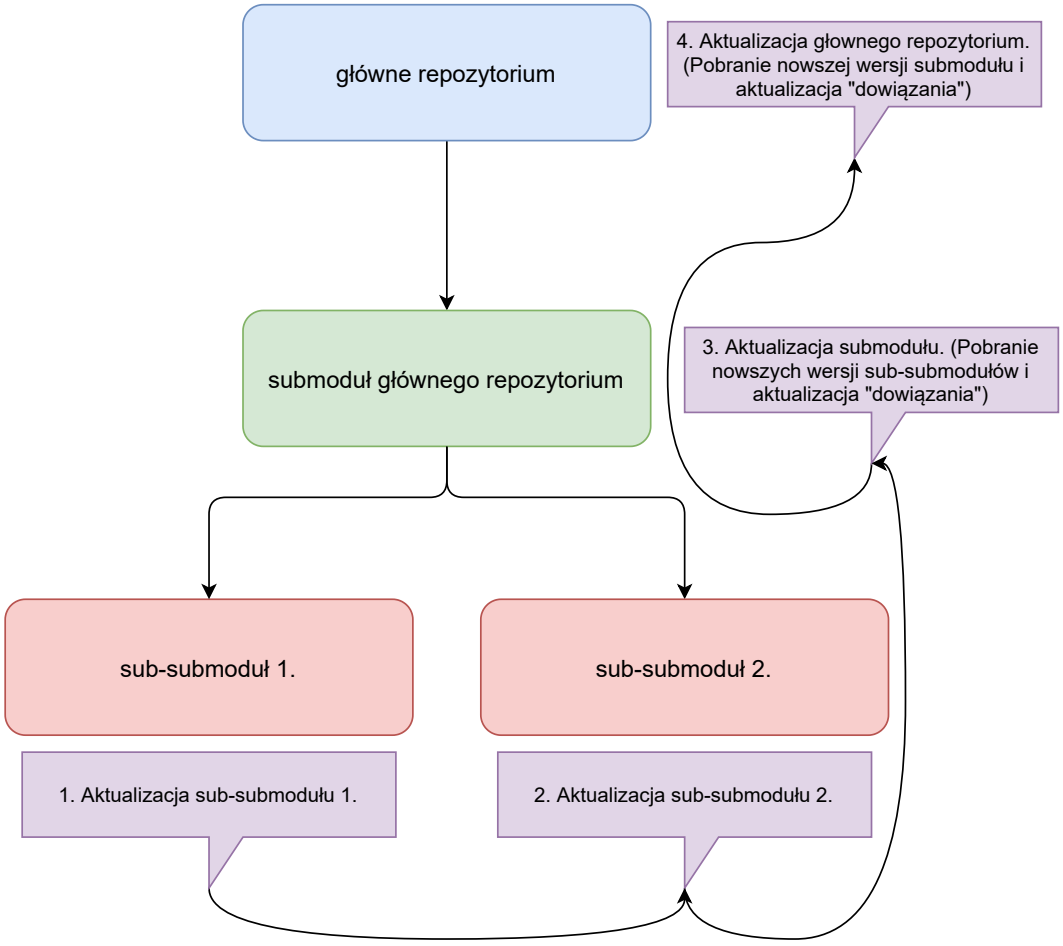
Rys. 4.4. Zasada działania submodułów.

Listing 4.4. Inicjalizacja pełnej sturktury projektu jedną komendą.

```
root@host:/# git clone ↩
    ssh://git@gitlab.cern.ch:7999/atlas-trt-dcs-ggss/ggss-all.git && cd ggss-all ↩
    && git submodule update --init --recursive
Cloning into '/CERN/ggss-all/ggss-dim-cs'...
Cloning into '/CERN/ggss-all/ggss-driver'...
Cloning into '/CERN/ggss-all/ggss-oper'...
Cloning into '/CERN/ggss-all/ggss-runner'...
Cloning into '/CERN/ggss-all/ggss-spector'...
Cloning into '/CERN/ggss-all/mca-n957'...
Cloning into '/CERN/ggss-all/ggss-dim-cs/external-dim-lib'...
Cloning into '/CERN/ggss-all/ggss-dim-cs/ggss-misc'...
Cloning into '/CERN/ggss-all/ggss-driver/external-n957-lib'...
Cloning into '/CERN/ggss-all/ggss-driver/ggss-misc'...
...(13 lines truncated)
```

4.2.2. Motywacja do wprowadzenia zmian

Pomimo wielu aspektów *git submodules*, które bardzo dobrze wpasowały się w, kreowaną przez autorów w trakcie pracy inżynierskiej, sturkturę technologia ta posiada też swoje minusy. Pierwszy znaczącym problemem napotkanym w trakcie pracy z submodułami było nietypowe zachowanie repozytoriów w trakcie ich inicjalizacji, a konkretnie automatyczne odłączanie ich od głównej gałęzi. Co więcej praca z submodułami wymaga od programisty zwiększonej czujności oraz stosowania dodatkowych zasad, ponieważ więcej jest miejsc na pomyłkę, co może doprowadzić do niepoprawnego działania wykorzystanych narzędzi. Kolejnym problemem napotkanym w trakcie pracy z submodułami jest czasochłonność niektórych operacji, w szczególności aktualizacji repozytorium na samym dole "drzewa zależności". Zmiana taka wymaga ręcznej aktualizacji po kolei każdego z repozytorium, aż do samej góry tejże skruktury co przedstawia rysunek 4.5. Każda z aktualizacji przedstawiona na wyżej wymieionym rysunku, to tak na prawdę cztery lub więcej akcji do których wliczają się: aktualizacja repozytorium podrzędnego, dodanie wszystkich zmian do rejestru odpowiedzialnego za ich śledzenie, utworzenie nowej wersji repozytorium, opublikowanie nowej wersji na zewnętrznym serwerze.



Rys. 4.5. Przykładowa architektura oparta o submoduły z krokami jakie należy podjąć, aby wprowadzić zmiany na “najniższym“ poziomie.

4.2.3. Automatyzacja z użyciem GITIO

Problemem, którego rozwiązanie pochłonęło najwięcej czasu i wymagało największego wkładu pracy przez autorów było monotonne, wielokrokowie wprowadzanie zmian do projektu, szczególnie u dołu struktury zależności. W celu rozwiązanie tego problemu przygotowano aplikacje *gitio* z wykorzystaniem języka *Python*. Ze względu na to, że metadane technologii *git* są bardzo złożone, a opanowanie zasad wewnętrznego działania tejże technologii wymagałoby bardzo dużo czasu skorzystano z dedykowanej, do tej technologii, bilbioteki napisanej również w języku *Python*.

Zasada działania aplikacji jest dosyć prosta, natomiast znacząco ułatwia działania z wielopoziomową strukturą opartą o *git submodules*. Argumenty wejściowe jako przyjmuje *gitio* to:

– `-h, --help` - pozwala na wyświetlnie informacji o przeznaczeniu programu oraz przyjmowanych argumentach wraz z krótkim opisem

- `-p PATH, --path PATH` -
- `-b BIN, --bin BIN` -

4.2.4. Dokumentacja sposobu pracy z submodułami

4.3. Rozwój systemu budowania projektu (AK)

Pierwsze w pełni przystosowane do pracy z opartą o mechanizm submodułów strukturą projektu wydanie systemu budowania aplikacji oraz bibliotek wchodzących w skład warstwy oprogramowania GGSS przygotowane zostało przez autorów w ramach napisanej przez nich pracy inżynierskiej. Spełniało ono wszystkie podstawione przed nim wtedy wymagania, takie jak możliwość niezależnego budowania pojedynczych komponentów projektu. Jednakże wraz z kontynuacją przez autorów prac nad projektem w ramach niniejszej pracy magisterskiej, pojawiła się możliwość rozwoju stworzonego systemu, co pozwoliłoby uczynić go bardziej niezawodnym i przyjaźniejszym dla użytkownika. Na kolejnych stronach manuskryptu opisane zostały zatem zmiany w systemie budowania projektu GGSS, wprowadzone przez autorów w czasie przygotowywania niniejszej pracy dyplomowej.

4.3.1. Wprowadzenie do problematyki

Przygotowany w ramach pracy inżynierskiej system budowania oparty został o narzędzie CMake oraz, w mniejszym stopniu, o proste skrypty napisane w języku Python. Rezultatem było w pełni działające rozwiązanie, spełniające wszystkie postawione wtedy przed nim wymagania:

- niezależność od platformy - system powinien działać poprawnie zarówno na urządzeniach wykorzystujących system operacyjny Linux, jak i na komputerach z Windowsem.
- możliwość budowania projektu o skomplikowanej, hierarchicznej strukturze, jakiego jak aplikacja *ggss-runner* wchodząca w skład warstwy oprogramowania GGSS
- możliwość budowania każdego z komponentów systemu z osobna, bez wykorzystywania pozostałych, niepowiązanych z nim modułów
- czytelna, jednopoziomowa (brak zagnieżdżeń) wynikowa struktura katalogów, w której każdy zbudowany komponent znajduje się tylko raz, niezależnie od tego, jak wiele modułów było od niego zależnych

Ponieważ oprogramowanie systemu GGSS składa się z wielu bibliotek i aplikacji, dla których proces tworzenia przebiega bardzo podobnie, zdecydowano się zastosować rozwiązanie pozwalające na ponowne wykorzystanie poszczególnych komponentów systemu budowania. W tym celu wykorzystywane często funkcjonalności wyodrębnione zostały w postaci osobnych plików `.cmake`. W ten sposób do osobnych plików wydzielony został kod odpowiedzialny m.in. za: budowanie bibliotek statycznych, budowanie zależności danego komponentu czy dołączanie do projektu bibliotek Boost i GSL. Następnie tego typu pliki (zwane w dalszej części pracy szablonami CMake) wykorzystywane były w plikach `CMakeLists.txt` odpowiedzialnych za budowanie poszczególnych komponentów projektu, co zostało zobrazowane w formie przykładu na listingu 4.5. Zaprezentowany fragment kodu zawiera konfigurację procesu budowania biblioteki *thread-lib*, wraz z komentarzami w języku polskim.

Listing 4.5. Fragment wykorzystywanego w pierwotnej wersji systemu budowania pliku `CMakeLists.txt` zawierający konfigurację procesu tworzenia biblioteki *thread-lib*. Widoczne są przyjęte podczas tworzenia pracy inżynierskiej konwencje dotyczące sposobu wykorzystywania szablonów CMake.

```
# Przykład pliku CMakeLists.txt zgodnego z przygotowanym w ramach pracy
# inżynierskiej systemem budowania.

cmake_minimum_required(VERSION 2.8)

set(target_name "thread")

# Sprawdzenie, czy komponent nie został już przetworzony.
if(NOT TARGET ${target_name})

    # Ustawienie ścieżki wskazującej na szablony CMake.
    set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/../ggss-misc/cmake_templates")

    # Załączenie zawartości wybranych szablonów.
    include(BuildLibrary)
    include(FindLibraryBoost)

    # Pliki nagłówkowe.
    target_include_directories(${target_name} PUBLIC ↩
        ${CMAKE_CURRENT_SOURCE_DIR}/include/ThreadLib)

    # Dodanie zależności - wykorzystanie szablonu BuildDependencies.
    set(dependency_prefix "${CMAKE_CURRENT_SOURCE_DIR}/..")
    set(dependencies "handle" "log")
    include(BuildDependencies)

endif()
```

Przykład obrazuje sposób wykorzystywania szablonów CMake, jaki przyjęty został przez autorów podczas tworzenia przez nich pracy inżynierskiej - załączenie pliku za pomocą komendy `include()` powodowało zamieszczenie znajdujących się w nim instrukcji bezpośrednio w danym miejscu kodu. Dodatkowo, jeśli zawartość tego typu pliku wymagała od użytkownika wyspecyfikowania dodatkowych informacji, takich jak lista koniecznych do zbudowania zależności, następowało to po prostu poprzez utworzenie (przed miejscem załączenia pliku) zmiennej o odpowiedniej nazwie - na załączonym przykładzie jest to widoczne w przypadku pliku `BuildDependencies.cmake`, wymagającego do poprawnego działania istnienia zmiennych `dependency_prefix` oraz `dependencies`.

4.3.2. Motywacja do wprowadzenia zmian

Przygotowane przez autorów w ramach pracy inżynierskiej rozwiązanie charakteryzowało się pewnymi ograniczeniami, takimi jak brak wsparcia dla testów jednostkowych oraz tworzenia dokumentacji kodu źródłowego za pomocą narzędzia Doxygen. Niewątpliwą wadą przygotowanego rozwiązania był ponadto sposób wykorzystywania szablonów CMake, wymagający od użytkownika znajomości ich zawartości (by wiedzieć, jakie zmienne powinny zostać utworzone, by system działał poprawnie). Znacznie bardziej przyjazną dla użytkownika alternatywą wydaje się być wykorzystanie funkcji i makr, gdzie wszystkie potrzebne informacje przekazywane byłyby w formie argumentów. Ponadto wraz z pojawianiem się w warstwie oprogramowania systemu GGSS kolejnych aplikacji konieczne stało się rozbudowanie skryptu `build.py`, znajdującego się w repozytorium *ggss-all* i odpowiadającego za wysokopoziomowe zarządzanie procesem budowania (m.in. wybór między wersja *debug* i *release* oraz statycznym i dynamicznym linkowaniem bibliotek wchodzących w skład pakietu Boost).

4.3.3. Zastosowanie funkcji i makr narzędzia CMake

Jedną z wprowadzonych podczas prac nad systemem budowania projektu GGSS zmian była przebudowa istniejących plików `.cmake` poprzez umieszczenie znajdującego się tam kodu w starannie nazwanych funkcjach i makrach. Celem tej modyfikacji było ułatwienie osobom rozwijającym system GGSS dodawania nowych bibliotek i aplikacji. Jak zostało wyżej wspomniane, przed wprowadzeniem omawianej modyfikacji do sprawnego korzystania z przygotowanych szablonów konieczna była znajomość ich zawartości, ponieważ oczekiwały one często wyspecyfikowania wymaganych informacji za pomocą odpowiednio nazwanych zmiennych. Zamiast tego zaproponowane zostało rozwiązanie, w którym wszystkie potrzebne informacje przekazywane są do odpowiedzialnej za daną operację funkcji (lub makra) poprzez nazwane argumenty wywołania. Z punktu widzenia narzędzia CMake funkcjonalność taka dostępna jest przy pomocy komendy `cmake_parse_arguments`. Ze względu na trywialny i powtarzalny charakter wprowadzonych w szablonach CMake przekształceń, w niniejszej pracy pominięty zostanie opis implementacji nowego rozwiązania, wskazana zostanie natomiast różnica widoczna z perspektywy osoby wykorzystującej istniejący system budowania. Na listingu 4.6 przedstawiony został przykład wykorzystania jednej ze stworzonych przez autorów funkcji: `ggss_build_static_library` pochodzącej z szablonu `BuildStaticLibrary.cmake`. Przedstawiony fragment kodu wykonuje dokładnie to samo zadanie, co widoczny wcześniej na listingu 4.5 - odpowiedzialny jest za konfigurację procesu budowania biblioteki *thread-lib*. Jest on znacznie krótszy od widocznego w poprzednim przykładzie - zostało to osiągnięte poprzez przeniesienie do wnętrza funkcji funkcjonalności takich jak sprawdzenie, czy dany komponent został już przetworzony. Widoczny w przykładzie sposób przekazywania informacji za pomocą nazwanych argumentów funkcji (np. `DEPENDENCIES`) jest zdaniem autorów znacznie czytelniejszy od mechanizmu istniejącego w pro-

jekcie wcześniej. Ponadto, dzięki wykorzystaniu funkcji, przedstawiony fragment przypomina kod źródłowy napisany za pomocą popularnych języków programowania, takich jak C i C++, co zdaniem autorów czyni go bardziej intuicyjnym.

Listing 4.6. Fragment wykorzystywanego w najnowszej wersji systemu budowania pliku `CMakeLists.txt` odpowiedzialnego za konfigurację procesu tworzenia biblioteki *thread-lib*. Widoczne jest wykorzystanie funkcji oraz nazwanych argumentów w celu zwiększenie czytelności prezentowanego kodu.

```
cmake_minimum_required(VERSION 2.8)
project(ggss-thread-lib)

# Załączenie szablonów CMake zawierających wykorzystywane funkcje.
set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/../ggss-util-libs/cmake-templates")
include(BuildStaticLibrary)     # ggss_build_static_library

# Funkcja tworząca bibliotekę statyczną.
ggss_build_static_library(
    TARGET_NAME "thread"
    DEPENDENCY_PREFIX "${CMAKE_CURRENT_SOURCE_DIR}/.."
    DEPENDENCIES "sigslot" "ggss-util-libs/log"
)
```

W sposób podobny do załączanego w przykładzie pliku `BuildStaticLibrary.cmake` zmodyfikowane zostały pozostałe wchodzące w skład projektu pliki `.cmake` (np. plik `BuildDependencies.cmake`, odpowiedzialny za konfigurację zależności danego modułu) dzięki czemu wzrosła jakość kodu wchodzącego w skład systemu budowania. W większości przypadków autorzy zdecydowali się na wykorzystanie funkcji - zastąpienie ich za pomocą makr konieczne było tylko w kilku szczególnych przypadkach. Najważniejszą różnicą między obiema wspomnianymi konstrukcjami jest fakt, iż w ciele funkcji, w przeciwieństwie do marka, tworzony jest nowy zasięg widoczności zmiennych (ang. *scope*), przez co skutki zachodzących w jej wnętrzu zmian, takich jak przypisania, nie są domyślnie widoczne poza nią. Zastosowane zmiany zgodne są ponadto z konwencją, wedle której napisany za pomocą narzędzia CMake kod systemu budowania powinien być traktowany na równi z kodem źródłowym przetwarzanych przez niego aplikacji i bibliotek.

4.3.4. Wsparcie dla testów jednostkowych i dokumentacji

Kolejną wprowadzona przez autorów modyfikacją było dodanie, na poziomie systemu budowania, wsparcia dla testów jednostkowych oraz generowania, za pomocą narzędzia Doxygen, dokumentacji kodu źródłowego. Ponieważ obie wprowadzone funkcjonalności wykorzystywane są przez wiele komponentów projektu, ich implementacja zamieszczona została w znajdujących się aktualnie w repozytorium *ggss-util-libs* plikach `.cmake`, kolejno: `SetupTests.cmake` i

`SetupDoxygen.cmake`. Zgodnie ze stosowaną w projekcie konwencją, również te funkcjonalności zaimplementowane zostały w postaci funkcji (w przypadku dokumentacji) oraz makra (w przypadku testów, gdzie zostało to wymuszone z uwagi na konieczność wywołania polecenia `enable_testing` w odpowiednim zasięgu widoczności zmiennych).

Znajdująca się w pliku `SetupDoxygen.cmake` funkcja `ggss_setup_doxygen` wywoływana jest w ciele odpowiedzialnej za konfigurację procesu budowania bibliotek statycznych funkcji `ggss_build_static_library` (listing 4.7). Dzięki temu proces generowania dokumentacji konfigurowany jest automatycznie dla każdego modułu dodawanego do projektu. Poza wspomnianym szablonem CMake, w repozytorium *ggss-util-libs* utworzony został katalog `doxygen-config`, zawierający plik konfigurujący działanie narzędzia Doxygen (np. poprzez aktywację funkcjonalności rekurencyjnego przeszukiwania katalogów wchodzących w skład dokumentowanego komponentu). Użytkownik ma możliwość generowania dokumentacji w formacie HTML dla każdego komponentu projektu poprzez wykonanie polecenia `make <nazwa-komponentu>-docs`, np. `make xml-docs` powoduje wygenerowanie dokumentacji biblioteki odpowiedzialnej za obsługę plików XML.

Listing 4.7. Fragment pliku `BuildStaticLibrary.cmake`, przedstawiający wywołanie funkcji odpowiedzialnej za konfigurację procesu generowania dokumentacji za pomocą narzędzia Doxygen.

```
# Konfiguracja procesu budowania dokumentacji za pomocą narzędzia Doxygen.
ggss_setup_doxygen (
    TARGET_NAME "${ARG_TARGET_NAME}"
)
```

Zgodnie z założeniem, że mechanizm testów jednostkowych jest opcjonalny dla każdego z modułów warstwy oprogramowania systemu GGSS, wywołanie znajdującego się w pliku `SetupTests.cmake` makra `ggss_setup_tests` nie następuje w żadnym ze zdefiniowanych przez autorów szablonów CMake. Zamiast tego programista przygotowujący dany komponent podejmuje decyzję o wykorzystaniu tej funkcjonalności, co zostało przedstawione na listingu 4.8, gdzie zamieszczony został fragment pliki `CMakeLists.txt` konfigurujący proces budowania biblioteki *fifo-lib* oraz przygotowanych dla niej testów jednostkowych. Z punktu widzenia systemu budowania wsparcie dla tego typu testów oparte zostało o narzędzie CTest. Każdy z plików `.cpp` znajdujących się w katalogu `test` (usytuowanym na tym samy poziomie co katalogi zawierające pliki źródłowe i nagłówkowe tworzonego komponentu) wykorzystywany jest do stworzenia osobnego testu za pomocą komendy `add_test`. Uruchomienie testów możliwe jest na kilka sposobów, jednym z nich jest wykonanie polecenia `ctest --verbose` z poziomu katalogu, w którym przeprowadzany jest proces budowania testowanego modułu.

Listing 4.8. Fragment pliku `CMakeLists.txt`, przedstawiający wykorzystanie funkcji konfigurującej proces budowania modułu *fifo-lib* oraz makra odpowiedzialnego za obsługę przygotowanych dla niego testów jednostkowych. W celu zapewnienia poprawnego działania systemu budowania konieczne jest zachowanie odpowiedniej, zgodnej z przykładem, kolejności wywołań.

```
# Konfiguracja procesu budowania biblioteki statycznej fifo-lib
ggss_build_static_library(
    TARGET_NAME "fifo"
)

# Konfiguracja procesu budowania testów jednostkowych dla
# biblioteki fifo-lib.
ggss_setup_tests (
    TARGET_NAME "fifo"
)
```

4.3.5. Rozbudowa skryptu konfigurującego proces budowania projektu

Znajdujący się w repozytorium *ggss-all* skrypt `build.py` odpowiedzialny jest za przeprowadzanie wysokopoziomowej konfiguracji procesu budowania całego projektu (możliwy jest m.in. wybór budowanych komponentów systemu). Względem wersji skryptu przygotowanej przez autorów w ramach pracy inżynierskiej wprowadzone zostały niewielkie zmiany. Najważniejszą z nich jest zmiana sposobu, w jaki użytkownik dokonuje wyboru budowanych modułów. W pierwotnej wersji skryptu dokonywane to było poprzez wywołanie go ze specjalnymi argumentami powodującymi wykluczenie poszczególnych komponentów z procesu budowania (np. zastosowanie argumentu `--norunner` powodowało pominięcie w procesie budowania aplikacji *ggss-runner*). Wraz ze wzrostem liczby budowanych komponentów takie podejście stało się niepraktyczne, zastąpione zostało zatem przekazywaniem do skryptu argumentu `--apps` wraz z listą budowanych modułów. Po wprowadzeniu zmian następujące wywołanie: `build.py --apps runner dimcs` powoduje zbudowanie dwóch komponentów systemu: aplikacji *ggss-runner* oraz *dim-cs*. Ponadto dodane zostało wsparcie dla omawianego w dalszej części pracy mechanizmu wersjonowania (argument `--version`). Na listingu 4.9 przedstawiony został wynik wywołania skryptu z argumentem `--help`, powodującym wypisanie informacji na temat sposobu jego użytkowania.

Listing 4.9. Wynik wywołania skryptu `build.py` z argumentem `--help`, powodującym wypisanie informacji na temat sposobu jego wykorzystywania.

```
user@host:~/ggss-all$ python3 build.py --help
usage: build.py [-h] [-d] [-s] [-v] --buildtype {debug,release}
                [--boostpath {default,variable}]
                (--buildall | --apps {runner,serviceapps,dimcs,driver}
                [{runner,serviceapps,dimcs,driver} ...])
                [--version VERSION]
```

```
This script can be used for building various GGSS-related applications,
like ggss-runner.

optional arguments:
  -h, --help            show this help message and exit
  -d, --dimonline       build project using latest DIM version. Disabled by default.
  -s, --staticboost     force static Boost library linking. Disabled by default.
  -v, --venv            build project using virtualenv. All required dependencies
                        will be installed. Disabled by default.
  --buildtype {debug,release}
                        enable debug or release mode.
  --boostpath {default,variable}
                        choose boost path: default /usr/local or the one specified
                        using BOOST environment variable. Variable is used by
                        default.
  --buildall            Build all available apps.
  --apps {runner,serviceapps,dimcs,driver} [{runner,serviceapps,dimcs,driver} ...]
                        Choose apps to build.
  --version VERSION     Version with which apps will be labeled.
                        Default: dev-<currenttime> (YYYY-MM-DD_HH-MM-SS)

Consider visiting https://gitlab.cern.ch/atlas-trt-dcs-ggss/ggss-all for further
info on GGSS software.
```

4.4. Automatyzacja i centralizacja wersjonowania projektu (JC)

4.4.1. Wprowadzenie do problematyki

4.4.2. Motywacja do wprowadzenia zmian

4.4.3. ...

4.5. Pakietowanie i rozlokowanie projektu (JC)

4.5.1. Wprowadzenie do problematyki

4.5.2. Motywacja do wprowadzenia zmian

4.5.3. ...

4.6. Rozwój infrastruktury do testowania warstwy sprzętowej (JC)

4.6.1. Wprowadzenie do problematyki

4.6.2. Motywacja do wprowadzenia zmian

4.6.3. Rozwiązanie oparte o język Python

4.6.4. Rozwiązanie oparte o język C++

4.6.5. Podsumowanie

5. Prace nad kodem źródłowym projektu (AK)

5.1. Wprowadzenie - analiza aplikacji *ggss-runner*

5.2. Metodyka prac

5.2.1. Testy jednostkowe

5.2.2. Problem *mockowania*

5.2.3. Statyczna analiza kodu źródłowego

5.2.4. Przyjęte ograniczenia

5.3. Poprawa jakości kodu źródłowego

5.3.1. Migracja do standardu C++11

5.3.2. Naprawa błędów w kodzie

5.3.3. Zmiany w strukturze bibliotek

5.3.4. Prace nad biblioteką *ggss-lib*

5.4. Wprowadzenie nowych funkcjonalności

5.4.1. Biblioteka *hvcommand-lib*

5.4.2. Zmiany w sposobie aktualizacji parametrów i danych

5.4.3. Wprowadzenie dodatkowych komend

5.4.4. Zmiany w algorytmie dopasowywania krzywej

5.5. Podsumowanie

6. Testy systemu (AK i JC)

Niniejszy rozdział stanowi szczegółowy raport z przeprowadzanych w środowisku docelowym testów systemu GGSS. Został on podzielony na trzy części, opisujące różne rodzaje przeprowadzanych przez autorów sprawdzeń. Pierwsza z nich przybliża informacje dotyczące przeprowadzanych w sposób cykliczny testów - nacisk położony został tutaj przede wszystkim na opis powtarzanej w każdej iteracji procedury pozwalającej zweryfikować poprawność działania systemu. Druga część stanowi opis sprawdzeń wykonywanych w czasie mającej miejsce w lipcu 2021 roku migracji systemu do nowego środowiska docelowego. Zamieszczone w niej informacje dotyczą wkładu autorów we wspomnianą migrację, obejmującego m.in. wykonanie testów warstwy sprzętowej systemu GGSS. Ostatnia część niniejszego rozdziału opisuje wykonane w sierpniu 2021 roku testy finalnej wersji projektu. W tym przypadku przedstawiony został szczegółowy raport, obejmujący weryfikację poprawności działania każdej wprowadzonej do systemu lub zmodyfikowanej funkcjonalności, badanie stabilności systemu ze względu na wykorzystywane zasoby oraz testy nowych elementów infrastruktury, takich jak skryptu zarządzające środowiskiem docelowym.

6.1. Cykliczne testy systemu (AK)

Praca nad projektem stanowiącym część dużego, rozwijanego przez wiele osób systemu, wymaga stosowania metod pozwalających na zapewnienie jego niezawodności. Dlatego też autorzy zdecydowali się na przeprowadzania okresowych testów systemu GGSS, dzięki czemu możliwe było wczesne wykrywanie i eliminowanie pojawiających się w projekcie błędów. Regularne przeprowadzanie weryfikacji poprawności działania najnowszej wersji warstwy oprogramowania systemu GGSS pozwoliło ponadto na wygodne testowanie wprowadzanych przez autorów funkcjonalności - duża częstotliwość oznacza w tym przypadku możliwość testowania niewielkiego zbioru zmian, co znacząco ułatwia wczesne wykrywanie związanych z nimi nieprawidłowości.

6.2. Testy po migracji systemu (JC)

6.3. Testy wersji finalnej (AK i JC)

7. Podsumowanie (AK i JC)

A. Przegląd praktyk stosowanych podczas prac nad projektem (JC)

Celem niniejszego dodatku jest przedstawienie najważniejszych praktyk stosowanych przez autorów podczas wykonywania prac nad systemem GGSS. Poruszone zostaną tematy organizacji pracy nad kodem w zespole, dokumentacja projektu, czy też konwencje zastosowane w celu uzyskania w całym projekcie jednolitego kodu źródłowego.

A.1. Wprowadzenie do problematyki

Ze względu na zespołowy charakter przygotowanej przez autorów pracy inżynierskiej, w trakcie jej wykonywania wprowadzone zostały praktyki mające na celu organizację i koordynację współpracy. W ramach platformy GitLab, wykorzystywanej przez CERN jako główne narzędzie do współpracy nad kodem, skorzystano z szeregu funkcjonalności ułatwiających śledzenie postępów, jak i zarządzanie projektem. Oprócz utworzenia zespołu, do którego został przypisany kod projektu oraz w ramach którego odbywała się kolaboracja, wykorzystano:

- *issue* - opis pojedynczego zadania/problemu. Zawiera podstawowe informacje, przypisaną osobę, etykietę, która oznacza obecny stan, termin wykonania oraz wagę
- *kanban board* - tablica kanban zawierająca wszystkie przypisane do projektu zadania. Kolumny takiej tabeli stanowią spersonalizowane do projektu etykiety. Pozwala na wysokopoziomowe zarządzanie projektem, sprawdzenie statusu, czy też łatwą zmianę etykiety przypisanej do zadań poprzez przeciągnięcie do odpowiedniej kolumny.
- *merge request* - dedykowany widok do wprowadzania zmian wprowadzonych w ramach kodu deweloperskiego do kodu produkcyjnego, który jest wykorzystywany do tworzenia i dostarczania aplikacji
- *milestone* - jednostka organizacyjna pozwalająca na grupowanie kilku zadań, które realizują większy cel opisany w ramach *milestone*. *Milestone* śledzi przypisane do niego zadania, przewidywany czas zakończenia oraz wagę pozostałych do wykonania zadań.

W trakcie wykonywania pracy inżynierskiej, szczególnie podczas początkowego etapu projektu, który odbywał się w trakcie 3-tygodniowego wyjazdu do CERN, wyżej wymienione praktyki sprawdzały się bardzo dobrze.

A.2. Motywacja do wprowadzenia zmian

Ze względu na dobre sprawowanie się wyżej wymienionych praktyk w trakcie pracy inżynierskiej postanowiono o kontynuowaniu ich wykorzystania również w trakcie pracy magisterskiej. Z powodu nieregularnego aspektu pracy nad projektem, wykonywanie czynności zdalnie, lepsze poznanie środowiska, wykorzystywanych narzędzi oraz samej platformy GitLab postanowiono dostosować stosowane praktyki do nowych realiów. Dodatkowo bardzo ważną zasadą, biorąc pod uwagę zakończenie pracy nad projektem i przekazanie go osobie odpowiedzialnej za dalsze utrzymanie, było odpowiednie udokumentowanie całego projektu. Wymagało się, aby możliwie proste było wprowadzanie zmian do systemu GGSS oraz sprawne, nieprzerwane działanie po zakończeniu pracy magisterskiej. Dlatego dostarczona dokumentacja musiała być obszerna oraz dobrze opisująca zastosowane rozwiązania. Dodatkowo biorąc pod uwagę mocny nacisk tejże pracy na część aplikacyjną projektu potrzebne było zdefiniowanie pewnych zasad pozwalających na ustandaryzowaną pracę z kodem źródłowym aplikacji. Pozwoliło to na zachowanie pewnych konwencji w całym projekcie, co zapobiegało różnicom w kodzie między komponentami, a co za tym idzie utrzymanie kodu oraz wdrożenie nowych osób do projektu jest znacznie uproszczone.

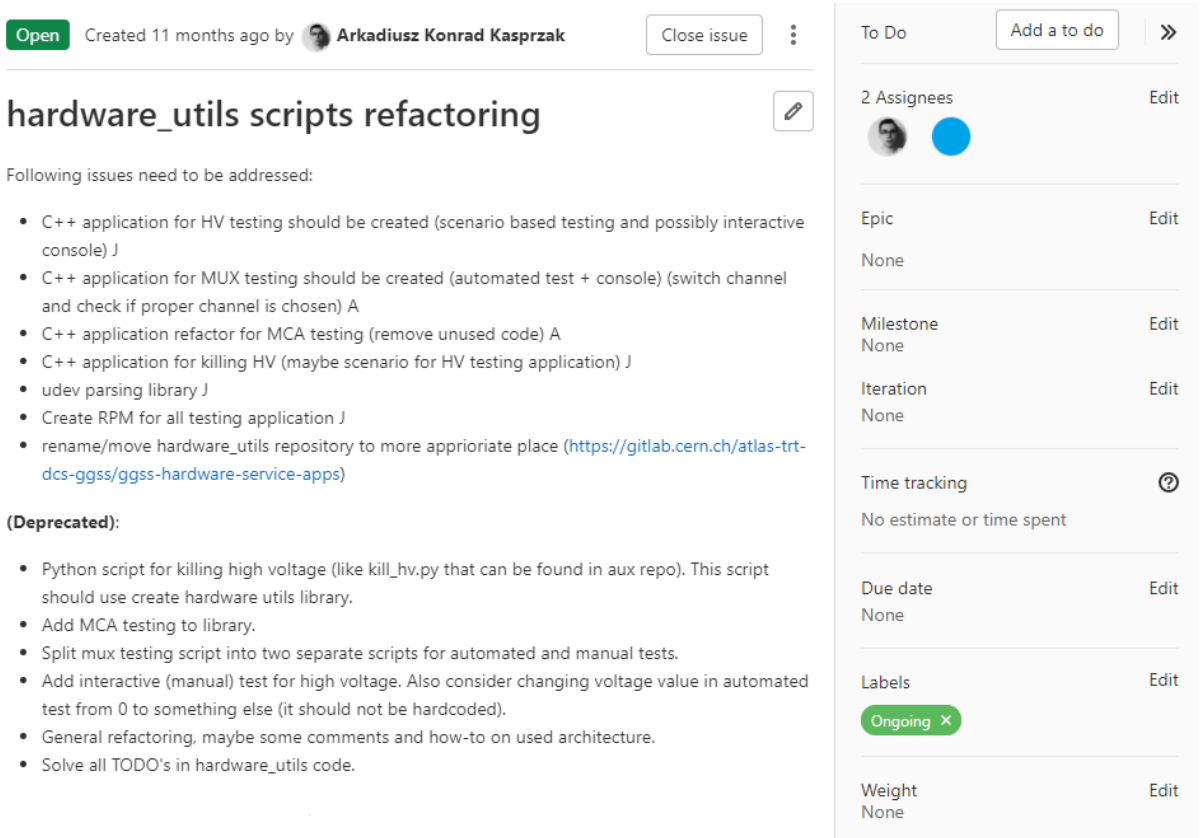
A.3. Zmiana praktyk ze względu na nieregularność prac

Prace nad systemem GGSS były kontynuowane, z mniejszymi przerwami, od obrony pracy inżynierskiej. Natomiast ich charakter był nieregularny. Każdy z autorów pracował nad systemem w wybranych przez siebie godzinach. Ze względu na to wszystkie praktyki, które opierały się o regularny czas pracy oraz przewidywanie czasu zakończenia danych zadań nie miały większego zastosowania. Postanowiono zatem zaprzestać przypisywania wag poszczególnym zadaniom. Oprócz wartości szacunkowej niewiele ona wnosiła w trakcie wykonywania zadania, dodatkowo przybrane wartości czasami różniły się od rzeczywistej wagi problemu, ponieważ często zadania wymagały w pierwszej kolejności zgłębienia tematu, a następnie określenia dokładnego rozwiązania problemu.

Zarzucono również praktykę wypełniania pola „termin oddania“ w ramach tworzonych zadań. Ze względu na wcześniej wspomniany czas poświęcany na pracę nad projektem, a szczególnie jego regularność, informacja ta często nie sprawdzała się z rzeczywistym czasem zakończenia zadania. Dodatkowo informacja ta nie była praktycznie w ogóle potrzebna w trakcie prac nad projektem ze względu na sposób formułowania zadań, które były możliwe do realizacji bez wpływu na pozostałe zadania - były od siebie niezależne. W przypadku zadań, które wymagały koordynacji, czy też pracy od obydwu autorów, organizowane były spotkania online z wykorzystaniem narzędzi takich jak Microsoft Teams, które pozwalały na tworzenie konferencji podczas których realizowane były wyżej wymienione zadania, czy też określane były ramy czasowe wykonania zadań

od siebie zależnych. Sposób ten sprawdził się bardzo dobrze i nie wymagana była dodatkowa koordynacja dla tego typu prac.

Rysunek A.1 przedstawia *issue* utworzone według nowo ustalonych zasad. Brak jest przypisanego *milestone*, czy też *due date*. Natomiast ważne, wartościowe informacje, przydatne w trakcie pracy nad projektem są wypełnione, tj.: rozbudowany opis pozwalający w krótkim czasie zrozumieć o co chodzi w konkretnym zadaniu, osoby przypisane do *issue* oraz etykiety oznaczające aktualny stan wykonania zadania, czy też jakikolwiek powód z którego *issue* nie zostało, bądź nie zostanie wykonane.



Rys. A.1. Przykładowe *issue* wg. nowo przyjętych praktyk

A.4. Dokumentacja projektu

Projekt GGSS ma być utrzymywany i pozostać w użyciu również po zakończeniu działań nad pracą dyplomową. Ze względu na to że rozwiązania wprowadzone do projektu były zarówno implementowane, jak i projektowane przez autorów w porozumieniu z promotorem, posiadają oni niezbędną do uwiecznienia wiedzę na temat: powodów zastosowania pewnych rozwiązań, sposobu ich działania, sposobu korzystania z nich, czy też zasad, które należy stosować w trakcie rozwoju aplikacji. Ze względu na te czynniki dużo uwagi poświęcono przygotowaniu odpowiedniej dokumentacji pozwalającej na swobodną pracę z projektem przez osoby, które ten projekt będą nadal utrzymywać.

Dokumentacja w postaci plików *README* napisanych w języku znaczników *Markdown* jest dedykowana dla każdego z repozytoriów. Zazwyczaj opisana jest w niej zawartość danego repozytorium, sposób użycia tejże zawartości, jeżeli wcześniejsze przygotowanie zawartości jest potrzebne opisane są kroki, które należy w takiej sytuacji poczynić. Dodatkowo w wyżej wymienionych plikach opisane są wszelkie niuanse, czy też bardziej zaawansowane kwestie dotyczące zawartości danego repozytorium.

Rysunek A.2 przedstawia przykładowy plik *README* dla repozytorium *ggss-all* zawierające infrastrukturę do budowy głównej aplikacji systemu GGSS. Wyżej wymieniony plik zawiera informacje o przeznaczeniu repozytorium, wymaganiach potrzebnych do spełnienia w celu uruchomienia infrastruktury budującej aplikację, krokach które należy podjąć, aby skorzystać z tejże infrastruktury. Oprócz tego plik ten zawiera gotowe do użycia komendy, które można skopiować i wkleić bezpośrednio do konsoli w celu skorzystania z infrastruktury. Plik ten zawiera również, a co nie jest widoczne na rysunku, informacje o sposobie uzyskania dostępu do kodu protokołu DIM, który jest wymagany do działania systemu GGSS.

Przygotowana w ten sposób dokumentacja pozwala osobie praktycznie niezapoznanej z projektem na skorzystanie z infrastruktury i przygotowanie gotowej do użycia, w środowisku docelowym, aplikacji. Również powrót do projektu po dłuższej przerwie nie powinien powodować większych trudności.

This is the **main repository** of the GGSS project. It contains scripts used for building most of the applications used by GGSS. It alse contains CI/CD configuration used to create production-ready packages. This `README` contains information about:

- repository content
- building procedure

Requirements

- CMake version 2.8 or higher
- C++ compiler
- Boost (version 1.57.0 or higher - it has to contain Boost.Log) - if such version is not available on used system, please download proper one and set BOOST environment variable to point to it (`export BOOST=<path_to_boost>`)
- GSL
- Python 3 - if necessary dependencies are not available, consider using virtualenv

Step by step

1. Create building directory: `mkdir build`
2. Go to newly created directory: `cd build`
3. Run `python <path_to_repo>/build.py <options>` from building directory
4. Applications will be built according to specified `<options>`

To clone and build all currently supported applications execute following commands:

```
git clone ssh://git@gitlab.cern.ch:7999/atlas-trt-dcs-ggss/ggss-all.git &&
mkdir ggss-all-build &&
cd ggss-all &&
git submodule update --init --recursive &&
cd ../ggss-all-build &&
python ../ggss-all/build.py --staticboost --buildall --buildtype release
```

Rys. A.2. Przykładowe *README* w ramach repozytorium *ggss-all*

W projekcie została zastosowana również dokumentacja na poziomie kodu źródłowego. Znajduje się ona między innymi: przed klasami, przed metodami, czy też na początku plików źródłowych. Dokumentacja ta stosuje format zgodny z narzędziem *doxygen*, co pozwoliło na jej ujednolicenie i zwiększenie czytelności. Dzięki wcześniej wspomnianej zgodności możliwe jest wygenerowanie dokumentacji w postaci plików *html*. Dokumentacja taka, w celu jej przeczytania, wymaga jedynie aktualnej przeglądarki internetowej. W celu pełnego wsparcia dokumentacji w postaci plików *html* generowanych z użyciem narzędzia *doxygen* potrzebne było również dosto-

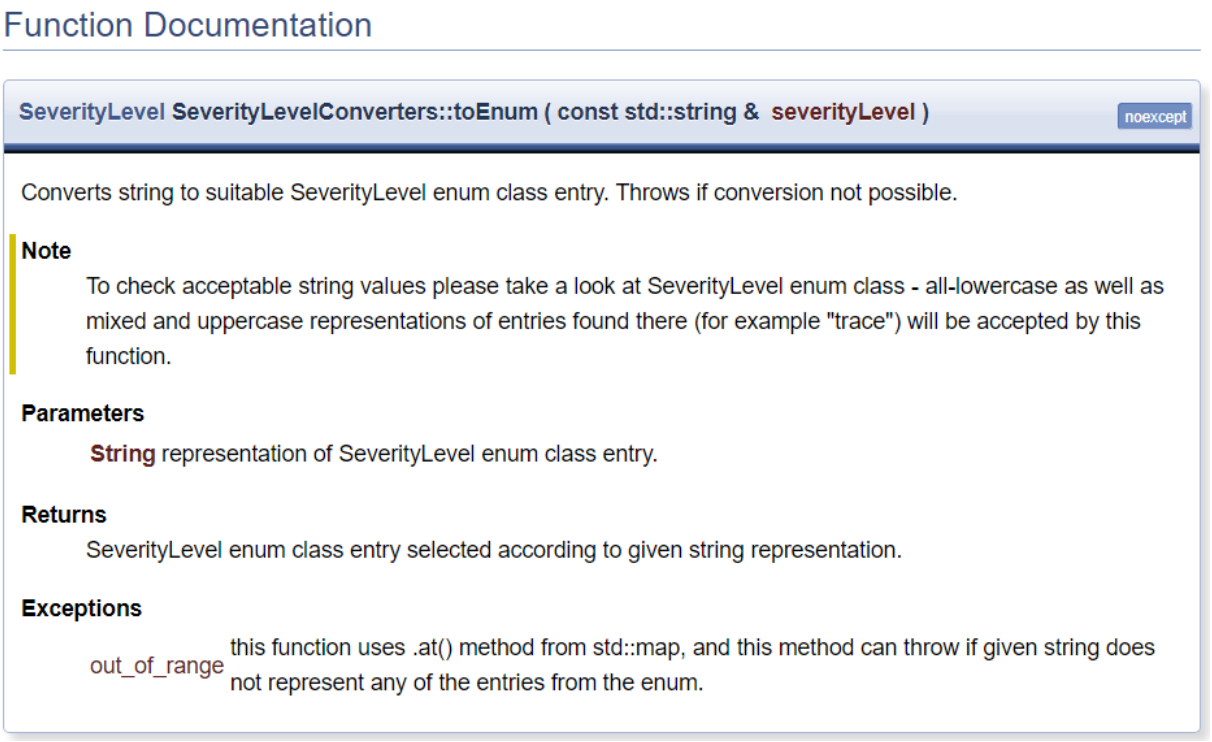
sowanie infrastruktury służącej do budowania projektu, a dokładnie plików *CMake*, dzięki czemu wygenerowane pliki *make* posiadają moduły odpowiedzialne za obsługę wcześniej wspomnianej dokumentacji.

Rysunek A.3 przedstawia przykładową dokumentację zgodną z formatem wspieranym przez narzędzie *doxygen*. Zawiera ona krótki opis dotyczący metody, następnie opis każdego z parametrów przyjmowanych przez daną metodą oraz wartość zwracaną przez metodę. Informacje te są bardzo przydatne w przypadku, gdy nie jesteśmy pewni, zważając na samą definicję metody, jej działania, parametrów wejściowych, czy też wyjścia. Opis taki rozwiewa częściowo wątpliwości i pozwala w poprawny sposób skorzystać z wcześniej napisanego kodu.

```
/**
 * \brief Computes initial peak position for Gauss fit.
 * \param fitData Data used for performing fit and finding peak position.
 * \param fitParams Structure with fit parameters (like fit range).
 * \return Initial peak position.
 */
double calculateInitialPeakPositionForGaussFit(const std::vector<double>& fitData,
                                               const FitParams& fitParams);


/**
 * \brief Calculates starting Gauss fit parameters for Xenon.
 * \param fitData Data used for performing fit.
 * \param fitParams Structure where initial fit parameters should be stored.
 */
void calculateStartParamsForXenonGaussFit(const std::vector<double>& fitData,
                                          FitParams& fitParams);
```

Rys. A.3. Przykładowa dokumentacja metod w bibliotece *fit-lib* w ramach projektu GGSS

Rysunek A.4 przedstawia dokumentację jednej z metod w bibliotece *log-lib*. Zawartość jest zbliżona jak w przypadku rysunku A.3, natomiast przedstawiona w bardziej przystępny sposób. Dokumentacja wygenerowana za pomocą narzędzia *doxygen* świetnie nadaje się na udostępnienie zewnętrznym użytkownikom. Pozwala również w łatwiejszy sposób przeglądać pełną dokumentację danego modułu bez potrzeby przeglądania kodu źródłowego.



Rys. A.4. Przykładowa dokumentacja metod w bibliotece *fit-lib* w ramach projektu GGSS

Ostatnim elementem dokumentacji zawartym w projekcie są dokumenty *how-to*. Napisane, podobnie jak pliki *README*, za pomocą języka znaczników *Markdown*, natomiast mają charakter globalny dla całego projektu - nie ograniczają się do jednego repozytorium. Dokumenty takie znajdują się w repozytorium *ggss-aux*. Opisane są tam krok po kroku bardziej zaawansowane aspekty pracy z projektem GGSS, jak np.: sposób obsługi architektury wielopoziomowej opartej o submoduły, czy też przygotowywanie wirtualnej maszyny do pracy jako *gitlab runner* w środowisku GitLab udostępnionym w ramach infrastruktury CERN.

A.5. Konwencja kodowania

Ze względu na to, że w trakcie pracy magisterskiej bardzo duży nacisk położono na część aplikacyjną projektu autorzy, jeszcze przed rozpoczęciem pracy nad kodem źródłowym, postanowili ustanowić konwencję kodowania, tak, aby na przestrzeni całego projektu GGSS utrzymać jednolity kod. Zasady, które zostały ustalony tyczą się nazewnictwa: klas, przestrzeni nazw, zmiennych, plików. Postanowiono wykorzystać, dobrze znane w środowisku, systemy notacji ciągów tekstowych *lower camel case* oraz *upper camel case*. Ze względu na różnorodność możliwych rozszerzeń plików w przypadku języka C++ postanowiono również ujednolicić ten aspekt. W przypadku plików z kodem źródłowym zastosowano rozszerzenia *.cpp* oraz *.h*.

Ustanawiając konwencję kodowania postanowiono ograniczyć się do wyżej wymienionych aspektów, sposób projektowania architektury, podziału na foldery, klasy, etc. wewnątrz da-

nego modułu pozostawiono bez większych obostrzeń. Oczywiście autorzy w każdym z dotkniętych miejsc stosowali dobre praktyki programistyczne oraz tak zwany *clean code*, natomiast, ze względu na to, że w większości przypadków prace nad projektem dotyczyły modyfikacji już istniejącego kodu oraz modułów była zachowana wcześniej zastosowana architektura.

B. Wybrane instrukcje i poradniki

B.1. Adding modules to the project using existing CMake templates

B.2. Working with git submodules

B.3. Using DIM HV commands