AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Jarosław Cierpich
Arkadiusz Kasprzak

kierunek studiów: **informatyka stosowana**

Rozbudowa i uaktualnienie oprogramowania systemu GGSS detektora ATLAS TRT

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

Kraków, styczeń 2020

Oświadczenie studenta

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

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

.....

(czytelny podpis)

Spis treści

1. Wstęp

1.1. Wprowadzenie do systemu GGSS

1.2. Cel pracy

2. Zastosowane technologie

2.1. Język C++

C++ jest kompilowanym językiem programowania ogólnego przeznaczenia **BJARNE** opartym o statyczne typowanie. Został stworzony jako obiektowe rozszerzenie języka C (z którym jest w dużej mierze wstecznie kompatybilny), lecz wraz z rozwojem pojawiło się w nim wsparcie dla innych paradygmatów, w tym generycznego i funkcyjnego. Sprawiło to, że język ten stał się bardzo wszechstronny - pozwala zarówno na szybkie wykonywanie operacji niskopoziomowych **BJARNE strona 41**, jak i na tworzenie wysokopoziomowych abstrakcji**BJARNE, PRACA GOSCIA**. Dodatkową cechą wyróżniającą C++ wśród innych języków umożliwiających programowanie obiektowe jest jego wysoka wydajność.

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

- C++11 - wprowadza funkcjonalności takie jak: wsparcie dla wielowątkowości, wyrażenia lambda, referencje do *r-wartości*, biblioteka do obsługi wyrażeń regularnych, dedukcja typów za pomocą słowa kluczowego *auto* czy pętla zakresowa. Standard ten uważany jest za przełom w rozwoju języka.

- C++14 - rozszerza zmiany wprowadzone w C++11. Nie zawiera tak wielu przełomowych zmian jak poprzedni standard - twórcy skupili się na poprawie istniejących błędów oraz rozwoju istniejących rozwiązań **wiki:** np. dedukcji typu zwracanego z funkcji za pomocą słowa kluczowego *auto*.

- C++17 - wprowadza m.in. nowe typy danych (*std::variant* czy *std::optional*, algorytmy współbieżne,

2.2. Biblioteki statyczne i dynamiczne

2.3. Narzędzie CMake

2.4. Język Python

2.5. Powłoka systemu operacyjnego - Bash

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

Listing 2.1. Komenda wypisująca tekst na standardowe wyjście wykonana z linii poleceń

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

Bash, czyli **Bourne Again Shell** jest powłoką systemu początkowo napisaną dla systemu operacyjnego GNU. Obecnie Bash jest kompatybilny z większością systemów Unixowych, gdzie zwykle jest powłoką domyślną oraz posiada kilka portów na inne platformy, tj.: MS-DOS, OS/2, Windows. WSTAWIC REFERENCJE DO www.gnu.orgsoftwarebashmanualbash.html#What-is-Bash_003f Oprócz pełnienia wyżej wymienionej funkcji, Bash jest również językiem programowania pozwalającym na tworzenie skryptów, które są kolejną metodą wprowadzania poleceń do powłoki systemu.

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

Listing 2.2. Skrypt wykorzystujący argumenty wejściowe, instrukcję warunkową oraz polecenie echo

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

Listing 2.3. Przykład działania Skryptu z Listingu 2.2

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

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

Bash posiada również wiele zaawansowanych funkcjonalności, które pozwalają na kontrolowanie przepływu informacji w trakcie wykonywania poleceń. Przykładem jest wpisywanie tekstu do pliku ukazane na Listingu 2.4.

Listing 2.4. Przykład zapisu tekstu do pliku

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

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

2.6. System kontroli wersji Git i portal Gitlab

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

Listing 2.5. Inicjalizacja repozytorium git

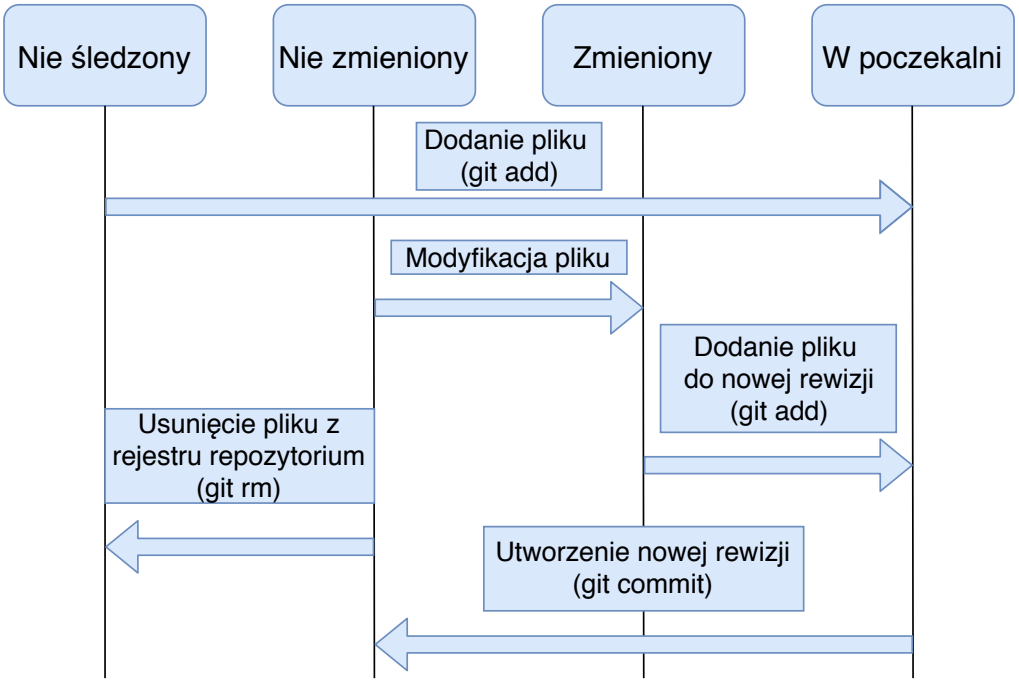
```
1 user@host:/ścieżka/do/projektu$ git init
2 Initialized empty Git repository in /ściezka/do/projektu
3 user@host:/ściezka/do/projektu$ ls .git
4 branches  config  description  HEAD  hooks  info  objects  refs
```

Taka inicjalizacja nie spowoduje żadnego dodatkowego działania oprócz utworzenia repozytorium. Żadne pliki nie są jeszcze poddawane rewizji. W celu rejestracji plików należy wykonać jeszcze kilka kroków. Pierwszym z nich jest wykonanie komendy **git add**, która poprzedza nazwę plików lub folderów, które chcemy poddać wersjonowaniu. Elementy te zostają dodane do tzw. poczekalni, czyli są

one kandydatami do utworzenia kolejnej rewizji. Przydatną komendą w tym przypadku jest również **git status** pozwalająca na sprawdzenie obecnego stanu repozytorium. Wyświetla ono krótkie podsumowanie nt. nowych plików, usuniętych plików oraz plików zmodyfikowanych. Informuje nas również o tym, które pliki są brane pod uwagę do utworzenia kolejnej rewizji.

Listing 2.6. Dodawanie elementów do poczekalni

```
1  user@host:/ściezka/do/projektu# git add plik1 folder1
2  user@host:/ściezka/do/projektu# git status
3  On branch master
4
5  No commits yet
6
7  Changes to be committed:
8    (use "git rm --cached <file>..." to unstage)
9
10         new file:   folder1/plik3
11         new file:   folder1/plik4
12         new file:   plik1
13
14 Untracked files:
15   (use "git add <file>..." to include in what will be committed)
16
17         plik2
```



Rys. 2.1. Możliwe stany pliku w repozytorium GIT WSTAWIC REFERENCJE DO TEGO

Tworzenie nowej wersji w ramach repozytorium odbywa się za pomocą komendy **git commit**. Sprowadza się do 'zamrożenia' obecnych wersji plików zarejestrowanych do rewizji oraz przypisanie im wspólnego, unikalnego dla każdej z nich, identyfikatora. Git udostępnia komendy pozwalające na przeglądanie oraz przywracanie plików do wcześniej utworzonych wersji. Listing 2.7 przedstawia utworzenie nowej wersji oraz wyświetlenie podsumowania o utworzonych do tej pory rewizjach.

Listing 2.7. Utworzenie nowej rewizji

```
1 user@host:/sciezka/do/projektu# git commit -m "Pierwsza rewizja"
2 user@host:/sciezka/do/projektu# git log
3 commit 1d2445e961beb25940dffa9d73963f887ee553ad
4 Author: user <user@host.localdomain>
5 Date:   Wed Dec 4 17:29:55 2019 +0100
6
7     Pierwsza rewizja
```

Przejścia między rewizjami nie powodują utraty danych, gdyż zachowywana jest informacja o stanie plików dla każdej z nich, co ukazuje Listing 2.8. Tworzona jest nowa rewizja zawierająca dodatkowo **plik2**, natomiast po powrocie do poprzedniej wersji plik ten nie występuje. Gdy powrócimy do nowszej wersji ponownie pojawi się **plik2**.

Listing 2.8. Podsumowanie rewizji, powrót do starszej wersji

```
1  user@host:/sciezka/do/projektu# git add plik2
2  user@host:/sciezka/do/projektu# git commit -m "Druga rewizja"
3  user@host:/sciezka/do/projektu# git log
4  commit b58836df55fc2a8eb2a43aa96273853776924807
5  Author: user <user@host.localdomain>
6  Date:   Wed Dec 4 17:30:10 2019 +0100
7
8      Druga rewizja
9
10 commit 1d2445e961beb25940dffa9d73963f887ee553ad
11 Author: user <user@host.localdomain>
12 Date:   Wed Dec 4 17:29:55 2019 +0100
13
14     Pierwsza rewizja
15
16 user@host:/sciezka/do/projektu# ls
17 folder1  plik1  plik2
18 user@host:/sciezka/do/projektu# git checkout 1d2445e961beb25940dffa9d73963f887ee553ad
19 user@host:/sciezka/do/projektu# ls
20 folder1  plik1
21 user@host:/sciezka/do/projektu# git checkout b58836df55fc2a8eb2a43aa96273853776924807
22 user@host:/sciezka/do/projektu# ls
23 folder1  plik1  plik2
```

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

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

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

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

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

2.7. Manager pakietów - RPM

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

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

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

- RedHat Linux
- CentOS
- Fedora
- openSUSE

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

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

Listing 2.9 pokazuje przykładową zawartość pakietu, czyli moduł kernela **modułAplikacji**, plik w formacie **JSON** pozwalający na konfigurację aplikacji oraz plik wykonywalny **aplikacji**, natomiast listing 2.10 pokazuje przykładową logikę pakietu RPM w postaci skryptów shell. Skrypty te nie zawierają ani kopiowania, ani usuwania plików zawartych pakiecie, proces ten odbuwa się automatycznie na podstawie ścieżek ukazanych na Listingu **??** w trakcie instalacji/dezinstalacji.

Listing 2.9. Przykładowa zawartość pakietu RPM

```
1  user@host:~# rpm -qpl pakiet.rpm
2  /ścieżka
3  /ścieżka/do
4  /ścieżka/do/konfiguracjaAplikacji.json
5  /ścieżka/do/aplikacji
6  /usr
7  /usr/lib
8  /usr/lib/modules
9  /usr/lib/modules/3.10.0-862
10 /usr/lib/modules/3.10.0-862/extra
11 /usr/lib/modules/3.10.0-862/extra/modułAplikacji.ko
```

Listing 2.10. Skrypty pakietu RPM

```
1  user@host:~# rpm -qp --scripts pakiet.rpm
2  preinstall program: /bin/sh
3  postinstall scriptlet (using /bin/sh):
4
5  #!/bin/sh
6  echo "Post-instajacja przykładowego pakietu"
7  echo "Przypisywanie uprawnień"
8  chmod 777 /ścieżka/do/konfiguracjaAplikacji.json
9  echo "Ładowanie modułu aplikacji"
10 /sbin/modrpobe modułAplikacji
11
12 preuninstall scriptlet (using /bin/sh):
13 #!/bin/sh
14 echo "Odinstalowywanie przykładowego pakietu"
15 echo "Usuwanie modułu aplikacji"
16 /sbin/rmmod modułAplikacji
17
18 postuninstall program: /bin/sh
```

2.8. Technologie wirtualizacji i konteneryzacji

Wirtualizacja, czyli proces uruchamiania instancji wirtualnego systemu komputerowego odseparowanego od rzeczywistego systemu komputerowego oraz jego sprzętu (ang. hardware). Pozwala na uruchomienie **wielu różnych** systemów operacyjnych na jednym komputerze **jednocześnie**. Wykorzystywany przede wszystkim do separacji środowisk dla aplikacji, czy też całych systemów. Pozwala na uruchomienie oprogramowania nieprzystosowanego do naszego systemu operacyjnego, wystarczy utworzyć instancję maszyny wirtualnej z odpowiednim systemem operacyjnym. Aplikacje uruchamiane w takiej instacji zachowują się tak, jakby znajdowały się na odseparowanym komputerze z własnym, dedykowanym systemem operacyjnym, bibliotekami oraz innym oprogramowaniem. Dużym plusem jest pełna separacja instancji uruchomionych na systemie gospodarza. Jedna instancja nie jest afektowana przez procesy innej instancji.

3. Stan początkowy projektu

3.1. Architektura

3.2. Budowanie

3.3. Dostarczanie i uruchamianie

3.4. Kontrola wersji

4. Stan docelowy projektu

5. Ograniczenia dostępnej infrastruktury

5.1. Ograniczone uprawnienia w środowisku docelowym

5.2. Wersje kompilatorów i interpreterów

5.3. Wersje narzędzia budującego CMake

5.4. Związek projektu z wersją jądra systemu

6. Wykonane prace

6.1. Wykorzystanie funkcjonalności portalu Gitlab wspierających zarządzanie projektem

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

6.3. Zastosowanie podejścia CI/CD

6.4. Zmiana sposobu budowania aplikacji

6.5. Budowanie i dystrybucja sterownika oraz aplikacji testującej

6.6. Maszyna wirtualna oraz konteneryzacja - Docker

6.7. Pomniejsze prace

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

6.7.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake

6.8. Dokumentacja projektu

7. Dalsza ścieżka rozwoju projektu

7.1. Wprowadzenie zautomatyzowanego systemu testowania projektu

7.2. Migracja do nowego standardu języka C++

7.3. Automatyzacja procesu publikowania produktu

8. Podsumowanie oraz wnioski

8.1. Statystyki projektu

A. Dodatki/Appendixes

A.1. Adding new modules to the project using existing CMake templates

A.2. Preparing virtual machine to work as a runner