AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

Projekt dyplomowy

Generator pakietów dla środowiska obliczeniowego R

Package generator for R platform

Autor: Magdalena Pastuła, Paweł Tyszko
Kierunek studiów: Informatyka
Opiekun pracy: dr Leszek Grzanka

Kraków, 2022

Spis treści

1. Cel prac i wizja produktu

Cel i wizja produktu

Poniższy rozdział stanowi wprowadzenie w tematykę projektu, w tym opis problemu, motywacja za sformułowaniem i podjęciem tematu, przewidywanych użytkowników końcowych, wykorzystane narzędzia, główne obszary funkcjonalne, na jakie można podzielić projekt, przegląd konkurencyjnych rozwiązań, analizę zagrożeń i problemów mogących się pojawić w trakcie rozwijania projektu, studium wykonalności oraz słownik pojęć używanych w tym dokumencie.

1.1. Problematyka

Środowisko R, pomimo ekstensywnego wykorzystania niskopoziomowego kodu do obliczeń numerycznych, nie daje trywialnej metody przekonwertowania biblioteki napisanej w czystym C na paczkę kompatybilną z tym środowiskiem. Istnieją wprawdzie metody bezpośredniego wywołania w skrypcie w języku R funkcji z kodu napisanego w języku C, jednakże w takim przypadku każda zmiana nazwy funkcji lub przyjmowanych przez nią argumentów w pierwotnym kodzie wymagałyby ręcznej zmiany skryptu R, który z nich korzysta.

Ręczne tworzenie paczki przy każdym wydaniu nowej wersji biblioteki z wrapperami funkcji z pierwotnego kodu nie rozwiązuje tych problemów, a wręcz dokłada kolejne, takie jak potrzeba znajomości języka programowania R czy znajomość procesu tworzenia paczki R i tego, co może być w niej zawarte. Niewygodne jest również ręczne publikowanie paczki na repozytorium. Obie te czynności podatne są na błąd ludzki i może się zdarzyć, że pewne wersje paczki mogą nie zostać opublikowane na repozytorium w wyniku zapomnienia lub zaniedbania.

Ten projekt ma na celu dostarczenie narzędzi, które przy minimalnej ilości konfiguracji i wiedzy o środowisku R pozwoli wygenerować i opublikować w repozytorium CRAN paczkę bez zewnętrznych zależności na podstawie zadanego kodu źródłowego C.

1.2. Motywacja projektu

AmTrack jest biblioteką współrozwijaną przez Opiekuna pracy i służy do predykcji odpowiedzi detektora oraz radiobiologicznej efektywności w silnie naładowanych wiązkach cząsteczek (była używana m.in. w publikacjach [10] i [11]). Kod źródłowy jest napisany w języku programowania C, dostępne są również programy pozwalające na użycie tej biblioteki w językach Python, Matlab i Java. Główną motywacją do stworzenia generatora, który pozwalałby na użycie tej biblioteki również w języku R jest fakt, że R zostało stworzone z myślą o analizie statystycznej danych oraz ich wizualizacji. W związku z tym czynności te są względnie łatwiejsze do wykonania niż w innych językach oraz dostarczona jest większa liczba opcji. Ponadto język R jest językiem programowania szeroko używanym w dziedzinie bioinformatyki [8], do której należy biblioteka AmTrack, zatem możliwość stosowania jej w tym języku mogłaby pozytywnie wpłynąć na jej popularność w tym środowisku.

Dodatkową motywacją za utworzeniem takiego systemu był fakt, że nie istnieje powszechnie dostępne rozwiązanie tego problemu, a przynajmniej takiego nie znaleźliśmy. Część wymaganych zadań była realizowana przez generator paczek już obecny w projekcie Libamtrack, lecz jest on niekompletny, a ponadto nie działa przy obecnych narzędziach wykorzystywanych w budowie biblioteki.

Naszą motywacją do podjęcia tego tematu była głównie chęć lepszego poznania środowiska i języka R, w tym procesu tworzenia paczek i ich publikacji. Jednocześnie projekt opiera się na

kodzie w języku C, który dosyć dobrze znaliśmy już wcześniej. Dodatkowo zainteresowała nas sama biblioteka Libamtrack i fakt, że jest ona wykorzystywana w ośrodku CERN.

1.3. Wizja produktu

Ze względu na swoją tematykę projekt można podzielić na dwa obszary funkcjonalne: program generujący paczkę na podstawie kodu źródłowego w C oraz testowanie kodu i automatyzacja pewnych procesów. Oba te obszary są szerzej opisane w sekcjach poniżej.

Docelowo użytkownikiem końcowym całego systemu będzie programista - twórca biblioteki pisanej w czystym C. Użytkownik powinien być w stanie wykorzystać oprogramowanie nawet przy niewielkiej - lub nawet żadnej - znajomości środowiska R. W domyśle jednym z użytkowników końcowych będzie Opiekun, wykorzystując oprogramowanie we współtworzonej przez siebie bibliotece Libamtrack. Jednakże, ponieważ generator będzie udostępniony na osobnym repozytorium, możliwe będzie jego wykorzystanie także do innych projektów przez innych ludzi.

1.3.1. Generowanie paczek

Paczka R w najprostszej wersji to zbiór funkcji wraz z plikiem DESCRIPTION zawierającym jej formalny opis, w tym w jakim celu została stworzona, przez kogo, numer wersji, listę paczek lub bibiotek, od których jest zależna czy też na jakiej licencji można jej używać. Dodatkowo, paczka może i często powinna zawierać dokumentację zawartych w niej funkcji, przykłady użycia, zbiór danych, testy, a także opis zmian wprowadzonych w aktualnej wersji.

Generowanie paczek będzie się odbywało poprzez parsowanie plików źródłowych w języku C lub dokumentacji, w celu otrzymania nazw funkcji oraz przyjmowanych przez nie parametrów i wartości zwracanych, a następnie tworzenie na tej podstawie funkcji owijających funkcje dostępne w bibliotece C (wrapperów). Użytkownik powinien miec możliwość zdefiniowania, które z funkcji powinny być dostępne na zewnątrz paczki R - możliwa jest sytuacja, w której nie wszystkie funkcje interfejsu biblioteki powinny być dostępne w intefejsie paczki.

Co więcej, na podstawie dokumentacji biblioteki w języku C powinna również generować się dokumentacja poszczególnych funkcji paczki R. W tym celu można wykorzystać dokumentację znajdującę się w plikach nagłówkowych C w formacie doxygen i odpowiednio ją przekształcać do formatu używanego w języku R.

1.3.2. Testowanie i publikacja

Ukończony projekt powinien mieć możliwość przetestowania wygenerowanej paczki pod kątem poprawności działania. Testowanie paczki będzie się odbywało poprzez sprawdzenie wyniku wywołania funkcji dostępnych na zewnątrz paczki z pewnymi argumentami.

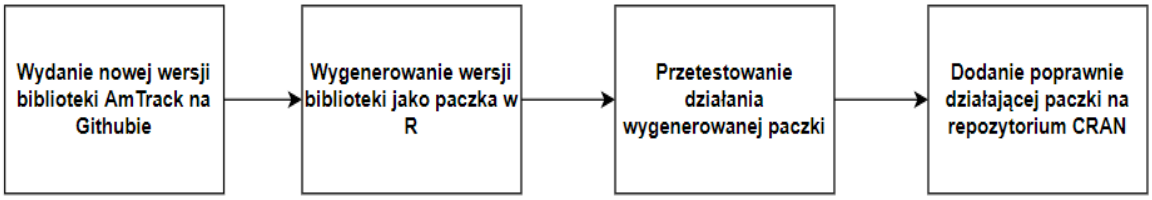
Po przetestowaniu paczka powinna zostać opublikowana na repozytorium paczek R. Preferowanym repozytorium jest CRAN, jednakże możliwe jest użycie w zastępstwie lub dodatkowo innego repozytorium, na przykład R-Forge, jeżeli będzie to miało dodatkowe korzyści.

Wygenerowana paczka powinna być również odpowiednio przygotowana i spełniać wymagania publikowanych paczek wybranego repozytorium. Dla przykładu, repozytorium CRAN nakłada dodatkowe wymagania na dane zawarte w pliku opisującym paczkę, jak imię i nazwisko autora oraz jego dane kontaktowe.

Wszystkie wyżej wymienione procesy są częścią mechanizmu publikacji paczki R i powinny być zautomatyzowane i zintegrowane z Github Actions, na przykład poprzez użycie narzędzia Continuous Integration (CI). Każdy z etapów tego mechanizmu powinien automatycz-

nie się zacząć po zakończeniu poprzedniego, gdzie punktem początkowym jest wydanie nowej wersji biblioteki w języku C, a efektem publikacja wygenerowanej na tej podstawie paczki na repozytorium CRAN lub innym.

Rysunek 1 pokazuje kolejne etapy automatycznego procesu wydawania nowej wersji biblioteki.



Rysunek 1: Schemat blokowy kolejnych etapów docelowego procesu publikowania nowych wersji biblioteki AmTrack

Cały proces publikacji paczki powinien również być kompatybilny ze środowiskiem CMake, którego używa biblioteka Libamtrack.

1.4. Analiza zagrożeń

Podczas rozwijania produktu istnieje ryzyko pojawienia się następujących sytuacji:

- problemy z uruchomieniem wygenerowanych paczek na systemie operacyjnym Windows lub iOS
- niepoprawnie napisane testy spowodowane złym zrozumieniem działania biblioteki
- niepoprawna implementacja konwersji funkcji napisanych w C do funkcji w R znajdujących się w wygenerowanej paczce
- niepoprawna implementacja dołączania zewnętrznych zależności paczki
- problemy z publikacją paczek do repozytorium CRAN wynikające z braku wiedzy
- trudności ze stworzeniem paczki bez zewnętrznych (binarnych) zależności - repozytorium CRAN nie dopuszcza plików wykonywalnych, obiektowych, bibliotek itp.
- zbyt duże bazowanie na poprzednim generatorze i w efekcie ponowienie pewnych błędów
- problemy z integracją z Githubem
- problemy ze skalowalnością systemu zintegrowanego z Githubem (rozszerzenie możliwości użycia na inne projekty)

Podczas rozwoju projektu część z tych problemów faktycznie się pojawiła i były to: problemy z publikacją paczki na repozytorium CRAN ze względu na brak API czy problemy ze skalowalnością systemu na inne biblioteki. Szerzej problemy te zostały opisane w sekcjach 5.1 i 5.4.

1.5. Studium wykonalności

Do rozwijania projektu wybrano język programowania Python z powodu prostego w obsłudze mechanizmu parsowania plików. Jako dodatkowe narzędzie wybrano generator dokumentacji do języka C - doxygen.

Pierwszy zrealizowany prototyp został zatem napisany w języku Python, z wykorzystaniem zewnętrznej biblioteki robotpy-cppheaderparser. Program parsuje plik nagłówkowy C w celu zdobycia informacji o typach parametrów oraz wyciągnięcia docstringów. Prototyp pozwala na wygenerowanie funkcji w języku R analogicznej do funkcji napisanej w języku C na podstawie podanego jako argument programu pliku nagłówkowego. Jednakże, program posiada ograniczenia i jednym z nich jest fakt, że obsługuje tylko funkcje posiadające parametry anotowane jako wyjściowe. Następnym krokiem w rozwoju prototypu będzie dodanie do niego możliwości generowania wrapperów w języku C dla funkcji dających wynik przez wartość zwracaną oraz automatyczna generacja docstringów dla generatora dokumentacji dla R - roxygen.

Poniżej wymieniamy alternatywne technologie oraz krótki powód ich odrzucenia.

1.5.1. Skrypty R ze starego generatora

Biblioteka AmTrack posiadała zbiór skryptów napisanych w 2012 roku przez dra Steffena Greilicha w języku R. Generator był dość zaawansowany, lecz ze względu na przystosowanie do starej wersji R oraz niedostosowania języka do operacji na tekście uznaliśmy, że próba rowijania i naprawiania nie będzie optymalnym rozwiązaniem.

1.5.2. Inne języki skryptowe

Wybór Pythona jako języka programowania dla skryptów był głównie podyktowany naszą znajomością i komfortem. Inne języki, takie jak Perl i JavaScript mają bardziej dojrzałe narzędzia do operacji na tekście, między innymi bardziej naturalne wkorzystanie wyrażeń regularnych, lecz ich zalety nie przeważałyby konieczności lepszego zapoznania się z językami.

1.5.3. Formalny opis funkcji

Podejściem zupełnie odmiennym do opisanych powyżej jest wykorzystanie prostego opisu nagłówków funkcji w sposób z założenia prosty do parsowania, wzorowany na lub wykorzystujący język opisu interfejsu (ang. IDL - Interface Description Language). Podejście takie jest mniej podatne na błędy użytkownika, jednak wymaga od niego najwięcej pracy oraz łamie regułę Don't Repeat Yourself. Te wady, a także ograniczenie dostępu takiego podejścia do dokumentacji, były powodem odrzucenia przez nas tego pomysłu.

1.6. Słownik pojęć

W opisie pracy wykorzystywane są następujące pojęcia i skróty:

- AmTrack - biblioteka pozwalająca na predykcję odpowiedzi detektora oraz radiobiologicznej efektywności na silnie naładowane wiązki cząsteczek rozwijana częściowo przez Opiekuna pracy. Jest to również kod napisany w języku programowania C, na którym zaprojektowany generator będzie docelowo używany
- CRAN (The Comprehensive R Archive Network) - publiczne repozytorium paczek R

- GSL (ang. GNU Scientific Library) - biblioteka zawierająca implementacje funkcji naukowych, z której korzysta biblioteka AmTrack
- submoduł - repozytorium podrzędne dołączone do innego repozytorium, które może korzystać z jego zawartości
- paczka R - rozszerzenie do języka programowania R zawierające ustandaryzowany kod, dane i dokumentację
- doxygen - generator dokumentacji między innymi do języka programowania C
- roxygen - generator dokumentacji dla języka R
- docstring - komentarz lub napis w kodzie o specjanym formacie pozwalającym automatyczne sparsowanie i wygenerowanie dokumentacji przez generator, np. doxygen, roxygen, javadoc...
- wrapper - kod, często automatycznie generowany, owijający (ang. wrap) funkcjonalność kodu w innym lub tym samym języku programowania w celu udostępnienia w sposób abstrakcyjny pewnej funkcjonalności (funkcji, klasy, modułu itp.)

2. Zakres funkcjonalności

W tym rozdziale został opisany zakres funkcjonalności projektu, czyli założenia na temat użytkowników systemu, opis systemów wykorzystywanych i współpracujących ze stworzonym systemem, wymagania funkcjonalne i niefunkcjonalne projektu, a także skalowalność systemu.

2.1. Użytkownicy systemu

Docelowymi użytkownikami systemu są autorzy bibliotek w języku C, którzy chcą generować paczki w R na bazie swojego kodu, oraz użytkownicy chcący korzystać z wygenerowanych paczek.

2.1.1. Użytkownicy generujący paczki w R

Zakładamy że autorzy bibliotek w C nie mają czasu lub wiedzy na temat języka i interfejsu platformy R, aby sami mogli napisać odpowiednie wrappery, skonfigurować środowisko i zbudować paczkę. Projekt zakłada, że użytkownik jest w stanie:

- odpowiednio zmodyfikować swój kod w C tak, aby spełniał wymagania systemu
- napisać pliki konfiguracyjne w celu dostosowania działania systemu do swoich potrzeb
- napisać pliki opisujące bibliotekę w formacie obsługiwanym przez system

2.1.2. Użytkownicy korzystający z paczek w R

Zakładamy że użytkownik wygenerowanej paczki nie musi znać języka C ani narzędzi koniecznych do skompilowania kodu. Użytkownik musi jedynie wiedzieć jak zainstalować bibliotekę z wybranego repozytorium paczek R lub z artefaktu udostępnianego na repozytorium biblioteki.

2.2. Współpracujące systemy

Jednym z narzędzi, z których korzysta wykonany projekt, to Github Actions. Jest to narzędzie do automatyzacji procesów wykonywanych na zawartości zadanego repozytorium na Githubie, między innymi do testowania czy budowania kodu. Narzędzie to jest szeroko wykorzystywane w naszym projekcie między innymi w procesie kompilacji i testowanie biblioteki testowej czy też testowania generatora na tej bibliotece. Wykorzystanie tego narzędzie jest szerzej opisane w podsekcji 3.8.

Innym wykorzystywanym mechanizmem w projekcie jest sposób dokumentowania kodu wyspecyfikowany dla narzędzia doxygen, generatora dokumentacji na podstawie komentarzy w kodzie dla różnych języków programowania, w tym dla C. Generator parsuje te komentarze w celu zdobycia informacji na temat parametrów przekazywanych do funkcji oraz wartości przez nie zwracanych. Informacje te są następnie wykorzystywane do stworzenia wrapperów.

2.3. Wymagania funkcjonalne

Wymaganie funkcjonalne stawiane projektowi można podzielić na wymagania dotyczące generowania wrapperów, dotyczące kompilacji paczki oraz inne.

Wymagania funkcjonalne dotyczące generowania wrapperów do funkcji C są następujące:

- obsługa wbudowanych typów języka C
- obsługa tablic
- rozpoznawanie parametrów wejściowych, wyjściowych oraz zarówno wejściowych jak i wyjściowych na podstawie dokumentacji
- tłumaczenie dokumentacji z formatu doxygen do formatu roxygen
- rozpoznawanie rozmiaru tablic na podstawie dokumentacji
- zwracanie wielu parametrów wyjściowych jako lista (struktura klucz-wartość)
- możliwość podawania tablic (wektorów) bez podawania ich długości

Wymagania funkcjonalne dotyczące sposobu kompilacji paczki są następujące:

- automatyczna kompilacja oryginalnych i wygenerowanych plików źródłowych
- możliwość wykorzystywania zewnętrznych bibliotek, np. GSL
- łączenie skompilowanych źródeł i bibliotek w jedną bibliotekę dynamiczną

Inne wymaganie funkcjonalne:

- generacja metadanych paczki na podstawie niewielkich plików z podstawowymi informacjami o bibliotece
- udostępnienie paczki na repozytorium paczek R lub wygenerowanie skryptu pozwalającego na łatwe jej zainstalowanie

W trakcie rozwijania projektu pojawiło się dodatko wymaganie, aby napisać kod testowy, na którym można by było testować generator w trakcie jego rozwoju. Wymagania funkcjonalne z nim związane są następujące:

- każdy typ zmiennych języka C obsługiwany przez generator powinien wystepować przynajmniej raz jako typ wartości zwracanej, parametru wejściowego i parametru wyjsciowego w funkcjach testowych
- kod testowy powinien zawierać osobne funkcje testowe dla typów tablicowych
- funkcje powinny być różnorodne pod względem liczby parametrów wejściowych i wyjściowych
- parametry wyjściowe i wartości zwracane powinny być w jakiś sposób zależne od parametrów wejściowych, o ile funkcja takie posiada

Wszystkie powyższe wymagania są podane w kolejności od najbardziej do najmniej istotnych.

2.4. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne postawione systemowi przedstawiają się następująco:

- wykonanie testów paczki w R - wygenerowana paczka powinna zostać przetestowana pod względem działania
- możliwość korzystania z generatora przez każdego, kto posiada konto na Githubie, w tym przez osoby biorące udział w projekcie AmTrack - generator powinien być szeroko dostępny, zarówno w wersji zautomatyzowaanej (poprzez dodanie jako submoduł do repozytorium) jak i wersji offline (poprzez sklonowanie lub sforkowanie repozytorium)
- możliwość działania powstałej paczki na systemach obsługiwanych przez platformę R (w momencie pisania: Windows NT oraz systemy uniksopodobne: Linux, iOS, itp.)
- możliwość działania generatora na systemach operacyjnych wymienionych w powyższym punkcie
- powiadamianie o problemach w procesie działania - zarówno sam generator, jak i cały proces automatycznej publikacji paczki powinny informować o ewentualnych zaistniałych błędach oraz co je spowodowało, jeśli to możliwe
- interfejs pozwalający na łatwiejsze używanie generatora - preferowanym sposobem jest wyświetlenie opisu dostępnych funkcji po zastosowaniu opcji `--help`
- względnie prosty sposób instalacji opublikowanej paczki - instalacja opublikowanej paczki powinna być prosta i wymagać jak najmniej wiedzy od jej użytkownika na ten temat. Funkcjonalność istotna w przypadku zmiany docelowego repozytorium paczek
- automatyczne udostępnianie paczki na repozytorium paczek przy każdym wydaniu nowej wersji biblioteki w języku C - wygenerowana i przetestowana paczka powinna zostać opublikowana na repozytorium paczej. Preferowanym jest CRAN.
- skalowalność - gotowy projekt powinien bez większych problemów móc być używany również przez inne repozytoria zawierające kod źródłowy bibliotek napisanych w języku C. Preferowanym sposobem jest możliwość dodania repzytorium generatora jako submoduł do repozytorium, na którym chcemy go wywołać. Możliwe powinno również być użycie generatora bez publikowania paczki lub użycie go na kodzie źródłowym, który nie posiada swojego repozytorium na Githubie, poprzez podanie jako argument ścieżki do folderu, w którym się mieści (wersja offline).

W trakcie rozwijania projektu pod wpływem uzyskania nowych informacji na temat jego wykonalności, część wymagań się zmieniła. W związku z tym doszły następujące wymagania niefunkcjonalne:

- automatyczne testowanie kodu testowego C - kod testowy powinien zostać przetestowany poprzez uruchomienie krótkiego programu sprawdzającego, czy jego funkcje działają tak, jak powinny. Proces powinien być zautomatyzowany, to znaczy wykonywać się dla każdej nowej wersji dodanej do repozytorium na Githubie.
- automatyczne testowanie generatora - każda nowa wersja powinna generatora powinna zostać automatycznie przetestowana na kodzie testowym. Podobnie, jak w przypadku testowania kodu testowego C, testowanie generatora powinno odbywać się automatycznie, to znaczy po każdym dodaniu nowej jego wersji na repozytorium na Githubie

Natomiast zrezygnowano z wymagania dotyczącego publikacji wygenerowanej paczki na CRAN lub inne repozytorium. Powody takiego posunięcia opisano w podsekcji 4.5.3. Nowoprzyjęte wymagania dostały taki sam priorytet, jak wymaganie, które zostało nimi zastąpione.

Tak samo, jak w przypadku wymagań funkcjonalnych wszystkie powyższe wymagania są podane w kolejności od najbardziej do najmniej istotnych.

3. Wybrane aspekty realizacji

3.1. Wykorzystywane narzędzia i programy

Jednym z narzędzi wykorzystywanych w projekcie jest Github Actions. Jest ono używane do kompilacji i budowania kodu testowego, testowania kodu testowego za pomocą krótkiego programu, testowania generatora na kodzie testowym, generowania paczki, testowania paczki oraz jej publikacji.

Do generacji wrapperów wykorzystywany jest skrypt napisany w języku Python, korzystający z zewnętrznej biblioteki robotpy-cppheaderparser[7]. Natomiast kod testowy został napisany w ANSI C.

3.2. Weryfikacja danych wejściowych

Sprawdzanie poprawności plików wejściowych odbywa się na wielu etapach generacji wrapperów, w zależności od rodzaju błędu który można napotkać.

3.2.1. Problemy z plikami

Pierwszym rodzajem błędów przed którymi chroni się generator są problemy z otwarciem plików. Zachowanie tutaj zależy od rodzaju pliku, lecz jest generalnie takie samo dla każdego możliwego problemu, m. in:

- plik nie istnieje
- plik istnieje, lecz program nie ma uprawnień, aby go otworzyć
- plik wyjściowy istnieje i jest otwarty w innym programie.

Pliki na których operuje generator wrapperów można podzielić na cztery kategorie: pliki wejściowe (pliki nagłówkowe w języku C), plik NAMESPACE (którego struktura i znaczenie są opisane w sekcji Interfejs generatora), plik DESCRIPTION (również opisany w sekcji Interfejs generatora) oraz pliki wyjściowe. Dla plików wejściowych rozwiązaniem jest wypisanie błędu i pominięcie pliku; dla pliku NAMESPACE generator przyjmuje, że zawiera on wszystkie możliwe funkcje; plik DESCRIPTION jest konieczny, lecz jego poprawność nie jest sprawdzna przez generator; natomiast dla plików wyjściowych generator przerywa działanie i informuje użytkownika o problemie (tzn. niemożliwość zapisania wrappera jest traktowana jako błąd krytyczny).

3.2.2. Poprawność dokumentacji

Poprawność dokumentacji (i w ogóle pliku wejściowego) jest najpierw sprawdzana przez bibliotekę robotpy-cppheaderparser, która zwraca błąd jeśli dany plik nie jest poprawnym plikiem C - w takim wypadku plik jest pomijany a użytkownik otrzymuje odpowiednią informację. Ten proces jest wykonywany raz na plik.

Następnie, przy każdej funkcji sprawdzane są następujące aspekty:

- czy funkcja jest udokumentowana, tj. czy posiada komentarz zgodny z formatem doxygen
- czy każdy parametr w dokumentacji posiada odpowiadający mu parametr funkcji i vice versa

```
/**
 * Prints param_one value on console.
 *
 * @param[in] param_one first param
 */
void one_in_param_no_return_function(int param_one);
```

Listing 1: Przykład poprawnej dokumentacji dla której nie zostanie wygenerowany wrapper: funkcja nie daje żadnej informacji zwrotnej

```
/**
 * This functions returns param_one value multiplied by 0.5
 *
 * @return param_one value multiplied by 0.5
 */
double one_in_param_double_return_function(int param_one);
```

Listing 2: Przykład błędnej dokumentacji: parametr param_one nie jest opisany

- czy funkcja w ogóle oddaje jakieś informacje wyjściowe, tj. czy zapisuje do parametrów lub zwraca wartość
- czy parametry tablicowe mają długość możliwą do obliczenia z parametrów skalarnych
- czy parametry i, jeśli jest potrzebna, wartość zwracana używają wyłącznie dozwolonych typów.

Jeśli którykolwiek z tych warunków będzie niespełniony generator zwróci informację o niemożliwości stworzenia wrappera. Listingi 1, 2 i 3 przedstawiają kilka przykładów funkcji, dla których system nie wygeneruje wrapperów.

3.3. Interfejs generatora

Generator wrapperów jest wywoływany z linii poleceń i ma dwa wymagane argumenty.

```
/**
 * Returns difference between two points in time
 *
 * @param end the minuend
 * @param start the subtrahend
 *
 * @return the difference end-start
 */
struct timespec time_diff(struct timespec end, struct timespec start);
```

Listing 3: Przykład błędnej funkcji: `struct timespec` nie należy do dozwolonych typów

```
# only_in_params_sample.h
one_in_param_no_return_function
one_in_param_int_return_function
one_in_param_unsigned_int_return_function
one_in_param_float_return_function
one_in_param_double_return_function
noR:one_in_param_bool_return_function
```

Listing 4: Przykładowy fragment pliku NAMESPACE

3.3.1. Pliki nagłówkowe

Podstawowym argumentem generatora jest lista plików wejściowych, nagłówków języka C, na podstawie których generowane będą wrappery, bądź wzorów typu glob[3] dających listy takich plików.

3.3.2. Plik DESCRIPTION

Drugim wymaganym argumentem jest ścieżka do pliku DESCRIPTION. Plik ten jest konieczny do tworzenia paczek w środowisku R i zawiera metadane na temat biblioteki, do których należą między innymi[13]:

- Nazwa paczki
- Autorzy i ich role w projekcie
- Licencja
- Wersja paczki
- Data zbudowania paczki (nieobowiązkowe)

Generator zezwala użycie w tym pliku dwóch symboli tymczasowych: `##DATE##`, który jest zamieniony na obecną datę w formacie rrrr-mm-dd oraz `##VERSION##`, który jest zamieniony na wersję paczki podaną jako argument (patrz: 3.3.4. Pozostałe argumenty).

3.3.3. Plik NAMESPACE

Dodatkowym, opcjonalnym argumentem jest plik NAMESPACE, który definiuje listę funkcji, jedna na linię które powinny otrzymać wrappery. Znak `#` oznacza komentarz, a przedrostek `noR:` oznacza, że dana funkcja ma otrzymać wrappery, jednak nie powinny być one eksportowane jako interfejs paczki. Białe znaki oraz puste linie są ignorowane.

3.3.4. Pozostałe argumenty

Poza wymienionymi powyżej parametrami, generator posiada kilka argumentów opcjonalnych. Należą do nich:

- flagi `-v` i `-q`, które odpowiednio zwiększają i zmniejszają ilość informacji wyświetlanych na ekran
- ścieżka `logfile`, która pozwala na przekierowanie informacji do pliku

Typ w C	Typ w R	Typ w C dla .C	Typ w C dla .Call (SEXPTYPE)
(unsigned) short	integer	int *	INTSXP
(unsigned) int	integer	int *	INTSXP
(unsigned) long	integer	int *	INTSXP
(unsigned) long long	integer	int *	INTSXP
int16_t	integer	int *	INTSXP
int32_t	integer	int *	INTSXP
int64_t	integer	int *	INTSXP
uint16_t	integer	int *	INTSXP
uint32_t	integer	int *	INTSXP
uint64_t	integer	int *	INTSXP
char	character	char **	STRSXP
int8_t	character	char **	STRSXP
unsigned char	raw	unsigned char *	RAWSXP
uint8_t	raw	unsigned char *	RAWSXP
bool	logical	int *	LGLSXP
float	single	float *	SINGLESXP
double	double	double *	REALSXP
long double	double	double *	REALSXP

Tabela 1: Typy parametrów/zwracanej wartości i ich odpowiedniki dla wrapperów w R i C [2][6][5]

- ścieżka `output_dir` - katalog, do którego chcemy wpisać wygenerowane pliki
- argument `version`, którym podmieniany jest tag `##VERSION##` w pliku DESCRIPTION; jeśli taki tag istnieje, ten argument jest wymagany

3.4. Generacja wrapperów

Generator dla każdej funkcji tworzy dwa wrappery: jeden w C, który konwertuje dane pomiędzy typami przyjmowanymi przez oryginalną funkcję a wykorzystywanymi przez system R na poziomie C; oraz drugi w R, który przygotowuje argumenty dla wrappera w C i wywołuje go. Wynikowe wrappery korzystają jednego z dwóch interfejsów dostępnych dla R: `.C` i `.Call`. Tabela 1 opisuje w jaki sposób konkretne typy są konwertowane na potrzeby danego interfejsu. Ponadto, program generuje jeden plik `init.c` dla całej paczki, który rejestruje wszystkie wrappery w C, aby były łatwo dostępne w środowisku R.

```
int one_in_one_out_param_int_return_function(int param_in,
                                             double* param_out)
{
    *param_out = 1.5 * param_in;
    return -1 * param_in;
}
```

Listing 5: Definicja funkcji testowej *one_in_one_out_param_int_return_function*

3.4.1. Funkcje zapisujące do parametrów

Jeśli funkcja zwraca wyniki przez zapisywanie do parametrów, generator wykorzystuje interfejs `.C`. W takim wypadku powstaje wrapper w C który przyjmuje wskaźniki do zmiennych podstawowych typów z C (3. kolumna w tabeli 1) i odpowiednio przekształca parametry do i z oryginalnych typów parametrów funkcji.

Zadaniem wrappera w R jest zagwarantowanie przekazania odpowiednich typów parametrów oraz zwrócenie wszystkich parametrów wyjściowych na zewnątrz (jeśli jest więcej niż jeden, dokonuje to przez stworzenie tablicy asocjacyjnej nazwa-wartość).

3.4.2. Funkcje zwracające wartość

Jeśli funkcja nie ma parametrów wyjściowych, a jedynie zwraca wartość przez słowo kluczowe `return`, to program tworzy wrappery korzystające z interfejsu `.Call`. Wrapper w C przyjmuje wtedy parametry typu SEXP i zwraca wartość typu SEXP[6]. Jego zadaniem jest wypakowanie argumentów z tych struktur oraz skonstruowanie wektora z wartości otrzymanej z oryginalnej funkcji, korzystając z jednego z makr wymienionych w 4. kolumnie tabeli 1. Ponadto wrapper musi też wyrejestrować ten wektor z garbage collectora środowiska R na czas kopiowania danych i wywołania funkcji.

Wrapper w R, podoblie jak w poprzednim przypadku, zapewnia zgodność typów oraz zwraca wartość. Ponieważ wywołanie `.Call` zawsze zwraca jeden wektor, ten wariant po prostu podaje tą wartość na zewnątrz.

3.5. Testowanie generatora

W ramach rozwijania produktu napisano również prostą bibliotekę w języku C, na której testowany jest generator w trakcie jego rozwoju. Biblioteka ta została napisana głównie ze względu na niepełną dokumentację biblioteki docelowej Amtrack, zatem testowanie generatora na niej wiązałoby się z problemami.

Funkcje testowe wykonują proste operacje na parametrach wejściowych i zwracają wynik poprzez parametry wyjściowe oraz wartości zwracane, o ile funkcja je posiada. Dla przykładu funkcja *one_in_one_out_param_int_return_function* przedstawiona na listingu 5 do parametru wyjściowego *param_out* zapisuje wartość parametru wejściowego *param_in* przemnożoną przez 1.5, a następnie zwraca liczbę przeciwną do wartości *param_in*.

W przypadku, gdy funkcja nie posiada parametrów wejściowych, do parametrów wyjściowych i wartości zwracanej zapisanywane są pewne wartości stałe. Natomiast dla funkcji, w którym parametrem wejściowym jest tablica, zwracana wartość jest równa sumie jej elementów przeskalowaną przez pewną stałą, czego przykład pokazany jest na listingu 7.

```
/**
 * This functions returns param_in multiplied by -1.
 * Param_out is set to param_in multiplied by 1.5.
 *
 * @param[in] param_in first param
 * @param[out] param_out second param
 *
 * @return status code
 */
int one_in_one_out_param_int_return_function(int param_in,
                                             double* param_out);
```

Listing 6: Przykład dokumentacji dla funkcji *one_in_one_out_param_int_return_function*

```
double one_table_in_no_out_params_double_return_function(
                                        double* table_param_in_one,
                                        unsigned int size_param_in_one)
{
    return table_sum(table_param_in_one, size_param_in_one);
}
```

Listing 7: Przykład funkcji testowej dla parametrów tablicowych.

Każda funkcja testowa opisana jest w pliku nagłówkowym komentarzem w formacie doxygen, z którego później korzysta generator. Listing 6 przedstawia przykład dla funkcji *one_in_one_out_param_int_return_function* wspomnianej powyżej.

Biblioteka testowa zawiera funkcje testujące następujące typy języka C:

- void - typ pusty. Używany tylko w przypadku braku wartości zwracanej przez funkcję
- bool - zmienna typu prawda-fałsz
- char - zmienna typu character (znak)
- unsigned int - zmienna całkowita bez znaku
- int - zmienna całkowita
- float - zmienna zmiennoprzecinkowa
- double - zmienna zmiennoprzecinkowa o podwójnej precyzji

Każdy z tych typów jest co najmniej raz testowany jako parametr wejściowy, parametr wyjściowy oraz wartość zwracana funkcji. Dodatkowo biblioteka zawiera również funkcje testowe dla parametrów tablicowych dla każdego z powyższych typów, za wyjątkiem typu `void*`, zarówno jako parametry wejściowe jak i wyjściowe. Dla parametrów wyjściowych nie będących tablicami do funkcji jest przekazywany wskaźnik na daną zmienną, a nie ona sama.

Biblioteka do testów nie zawiera funkcji testowych dla typów języka C takich jak `uint8_t` czy `long long`, gdyż uznano, że nie jest to konieczne i miałoby małą wartość dodaną.

Proces testowania odbywał się automatycznie przy każdej zmianie na repozytorium generatora. Polegało ono na skompilowaniu i przetestowaniu biblioteki w C, a następnie uruchomieniu na niej generatora. Po tym następowały testy wygenerowanego kodu:

1. czy biblioteka `roxygen2` jest w stanie przeczytać przetłumaczone docstringi,
2. czy środowisko R jest w stanie zbudować paczkę z wygenerowanego kodu,
3. czy wygenerowany kod odwołuje się do kodu skompilowanego,
4. czy wygenerowane funkcje dają identyczne wyniki co ich odpowiedniki w C[1].

Wszelkie zmiany w generatorze były poprawiane dopóki którykolwiek z tych kroków kończył się niepowodzeniem.

Testy były przeprowadzane na systemach operacyjnych Windows, Linux i MacOS, z wykorzystaniem interpretera CPython 3.8 oraz środowiska R w wersji 4.1.

3.6. Testowanie biblioteki testowej

W trakcie rozwoju biblioteki testowej pisano również jej testy, aby móc sprawdzić, czy działa odpowiednio. Testowanie odbywa się poprzez wykonanie programu, który wywoływał wszystkie funkcje i sprawdzał ich wynik. Wyniki były sprawdzane bezpośrednim porównaniem z wyłączeniem liczb zmiennoprzecinkowych, które były porównywane poprzez porównanie ich różnicy do liczby 0.0001. W przypadku, gdy wartość zwracana funkcji lub wartość parametru wyjściowego nie zgadzały się z oczekiwaną wartością, cały proces był przerywany. Ponieważ w trakcie wykonywania się testów wypisują się logi z informacją, które testy zakończyły się powodzeniem, można łatwo odszukać, która funkcja zwróciła błędną wartość. W przypadku wystąpienia błędu kod poprawiano, aż spełniał wszystkie testy.

Testy biblioteki testowej były przeprowadzane na systemach operacyjnych Windows, Linux i MacOS z wykorzystaniem narzędzia CMake. Dla każdego z tych systemów CMake automatycznie wybierał odpowiedni kompilator i były to odpowiednio: MSVC w wersji 19.29, GNU w wersji 9.3 i AppleClang 13.0. Testy były wykonywanie automatycznie po każdej zmianie zmianu testowego lub generatora.

3.7. Prototyp generatora

Jednym z etapów prac nad projektem było utworzenie prostego prototypu generatora. Prototyp ten został opisany w podsekcji 1.4.

3.8. Automatyzacja procesów

Zgodnie z wymaganiami postawionymi projektowi, uruchamianie się następujących po sobie procesów jest zatomatyzowane. Do tych procesów należą:

- kompilacja i przetestowanie biblioteki testowej w C w celu sprawdzenia poprawności jej działania
- wygenerowanie paczki na podstawie kodu testowego
- zbudowanie i zainstalowanie wygenerowanej paczki
- przetestowanie wygenerowanej paczki za pomocą gotowego skryptu

[1] z wyjątkiem wartości zmiennoprzecinkowych, których porównanie polegało na sprawdzeniu, czy rzeczywisty wynik różni się od oczekiwanego o nie więcej niż 0.001%

Wszystkie te czynności wykonywane są jedna po drugiej, w takiej kolejności, w jakiej zostały podane. Wykonywane są na pull requestach w repozytorium generatora przy każdej wprowadzonej do nich zmianie.

W trakcie generowania wrapperów generator przygotowuje również paczke do publikacji na CRAN. W ramach tego wykonywane są następujące działania:

- konwersja dokumentacji kodu z formatu doxygen do formatu roygen2
- kopiowanie pliku DESCRIPTION do folderu wyjściowego generatora

Proces automatyzacji został przeprowadzony tylko dla biblioteki testowej.

4. Organizacja pracy

W tym rozdziale opisano sposób organizacji pracy nad projektem, podział obowiązków między członków zespołu oraz związane z tymi obszarami aspekty. Dodatkowo w ostatniej sekcji opisano również główne problemy w trakcie rozwijania projektu oraz ich rozwiązania.

4.1. Charakterystyka projektu i sposób realizacji

Rozwijany projekt był badawczo-rozwojowy, większość wymagań była określona w momencie rozpoczęcia prac, jednakże częśc wymagań została zmieniona lub dodana w trakcie prac z powodu większej wiedzy w tym zakresie.

Częścią badawczą projektu było zbadanie, na ile możliwe jest spełnienie narzuconych wymagań, a także sprawdzenie, czy nie istnieją łatwiejsze sposoby realizacji niektórych z nich. Przykładem może tutaj być zbadanie alternatywnych repozytoriów paczek R do repozytorium CRAN czy też sprawdzenie procedury publikacji paczki na repozytorium CRAN.

Natomiast częścią rozwojową projektu było utworzenie produktu spełniającego jak najwięcej z zadanych wymagań.

Proces rozwoju projektu był przyrostowo-iteracyjny. Wynika to z faktu, że projekt można podzielić na następujące niezależne lub słabo zależne części:

- wrappery funkcji C do R
- kod testowy, na którym można testować generator
- tworzenie paczki na podstawie wrapperów i kodu w C
- przygotowanie paczki do publikacji na repozytorium CRAN

Powyższa lista została uporządkowana pod względem kolejności rozpoczęcia prac nad tymi modułami.

4.2. Osoby biorące udział w projekcie

W projekcie udział brały następujące osoby: Magdalena Pastuła oraz Paweł Tyszko jako zespół rozwijający projekt oraz dr Leszek Grzanka jako opiekun projektu oraz klient.

Do obowiązków Magdaleny Pastuły jako członka zespołu należały organizacja oraz pilnowanie terminów spotkań z promotorem, napisanie kodu testowego w C, a także jego testów i kompilacji, integracja projektu z Githubem oraz częściowo przygotowanie paczki do publikacji na repozytorium CRAN.

Do obowiązków Pawła Tyszko należały szukanie rozwiązań adresujących problematykę produktu, wstępne ustalenie wykonalności automatyzacji udostępniania paczki na repozytorium CRAN, rozwijanie generatora wrapperów oraz ustalenie formatu danych wejściowych dla niego.

Dr Grzanka w ramach projektu rozpisywał zadania w formie issues na Githubie, a także poddawał prace zespołu procesowi code review.

4.3. Organizacja prac

W zespole odbywały się cotygodniowe spotkania mające na celu wyjaśnienie pewnych kwestii i problemów, podsumowanie prac od ostatniego spotkania i organizację pracy na następny tydzień. Spotkania odbywały się przy pomocy aplikacji Discord.

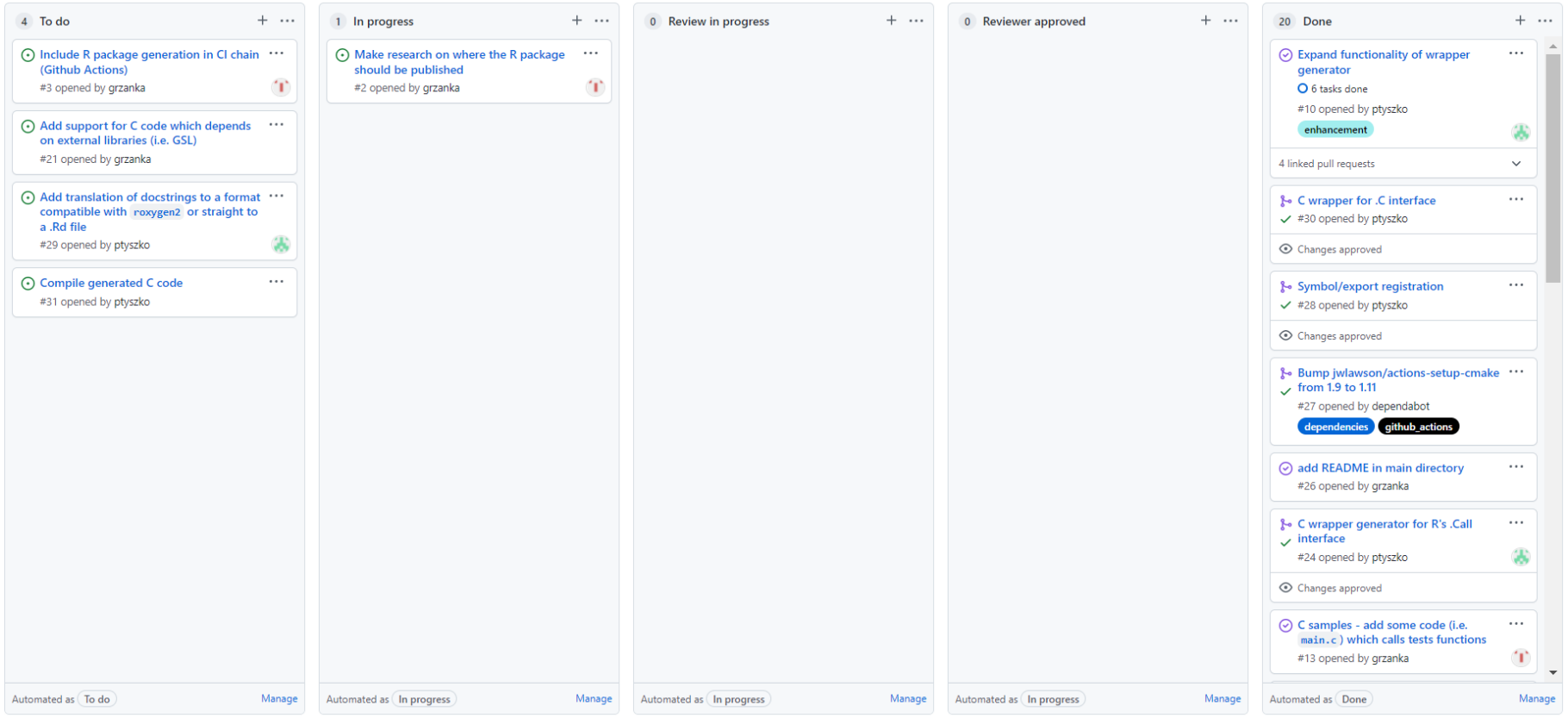
Zastosowany sposób organizacji w zespole odpowiada metodyce Scrum z wyłączeniem takich spotkań jak retrospektywa, czyli oceny sprintu, oraz daily, czyli dziennego składania raportów na temat poczynionych postępów.

W ramach uporządkowania zadań oraz w celu łatwego śledzenia postępu utworzono a następnie używano tablicy Kanban na Githubie na repozytorium generatora. Obraz 2 przedstawia stan tablicy z dnia 22.11.2021. Tablica ta zawiera zadania jako issue na Githubie podzielona na pięć kategorii: do zrobienia (To do), w trakcie (In progress), w trakcie przeglądu (Review in progress), zatwierdzone po przeglądzie (Reviewer approved) oraz Zrobione(Done). Co więcej, zadania do zrobienia były sortowane pod względem priorytetu - najważniejsze zadania znajdowały się na początku, a najmniej ważne na końcu listy.

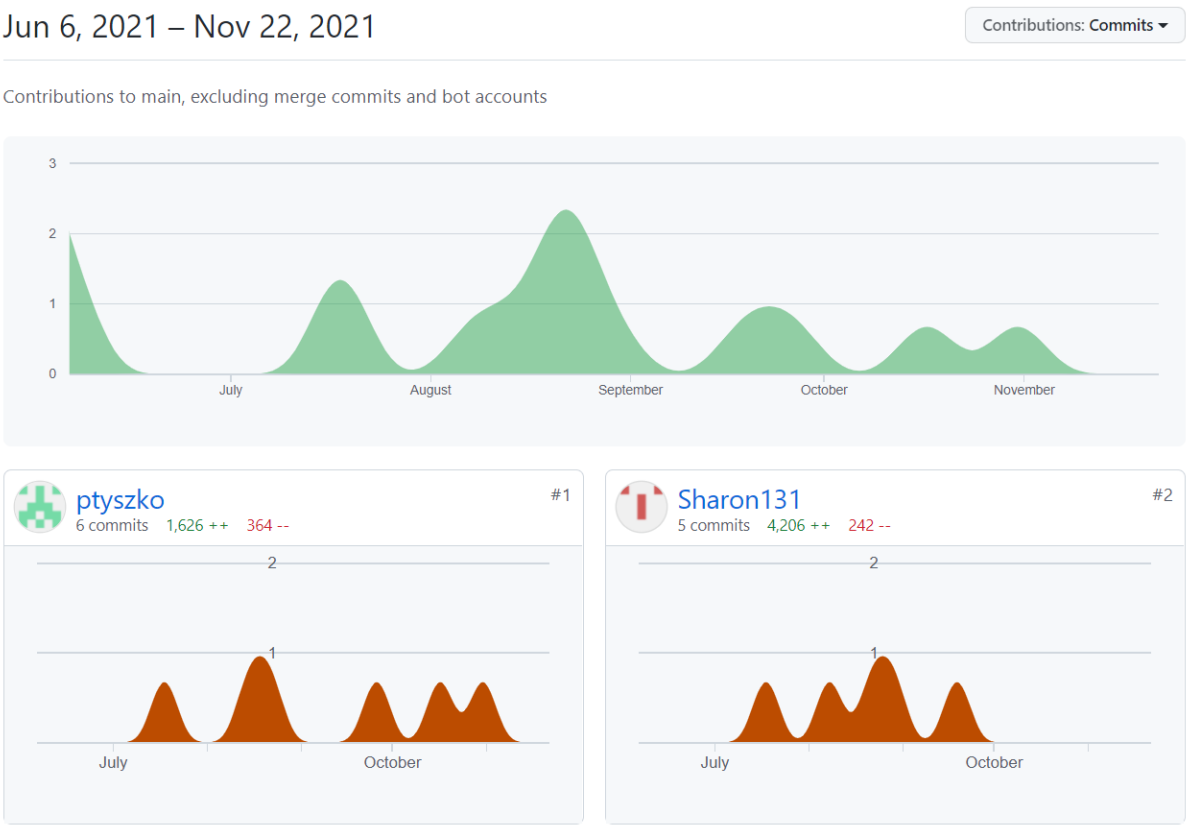
W trakcie prac nad projektem do tablicy dodawane były kolejne zadania do zrobienia odpowiadające kolejnym krokom w rozwoju projektu.

Natomiast obraz 3 przedstawia zdjęcie wykresu statystyk kontrybucji do repozytorium z dnia 22.11.2021 od dnia 6.06.2021, kiedy to na generator zostało utworzone osobne repozytorium. Wcześniej zmiany były wykonywane na samym repozytorium biblioteki Amtrack, gdzie przechowywany był stary generator.

Inną częścia Githuba, która była często wykorzystywana w projekcie, to Continuous Integration. Narzędzie do wykorzystano do automatycznego budowania i testowania kodu testowego oraz testowania generatora i wygenerowania paczki.



Rysunek 2: Stan używanej tablicy Kanban z dnia 21-11-2021.



Rysunek 3: Wykres kontrybucji w repozytorium ReGenerator do dnia 21-11-2021.

4.4. Harmonogram prac

Projekt rozwijany był w okresie od połowy marca 2020 roku do grudnia 2021 roku. Prace nad projektem można podzielić na kilka etapów, które zostały opisane poniżej.

Pierwszy etap prac nad projektem trwał od marca do kwietnia 2021 roku i było to wstępne rozpoznanie problemu i możliwych rozwiązań. W efekcie zebrane zostały informacje na temat istaniejących możliwości oraz nastąpiło zapoznanie się ze starą wersją generatora i decyzja, w jakim stopniu można go wykorzystać.

Drugi etap prac nad projektem trwał od kwietnia do czerwca 2021 roku i polegał głównie na pierwszych próbach rozwiązania problemu. Efektem tych prac było utworzenie prototypu produktu, który parsował zadany plik nagłówkowy i na tej podstawie tworzył plik R z wrapperami funkcji tam udokumentowanych. Działanie prototypu było ograniczone do przypadków opisanych w sekcji 3.7.

Następny etap prac nad projektem trwał od lipca do października 2021 roku i był to etap intensyfikacji prac nad projektem. Utworzony w poprzednim etapie prototyp był dalej rozwijany, a do tego napisano również kod testowy dla generatora.

Ostatni etap prac trwał od października do grudnia 2021 roku i polegał na dokończeniu prac. Efektem tych prac był gotowy produkt oraz jego dokumentacja.

4.5. Główne problemy i ich rozwiązania

W trakcie rozwoju projektu natrafiliśmy na kilka problemów. W następujących sekcjach znajdują się ich opisy oraz rozwiązania.

4.5.1. Wykorzystanie starego generatora

Na wczesnym etapie postanowiono, aby pozbyć się poprzedniego generatora paczek, głównie z powodu trudności w zrozumieniu kodu w R oraz jego nieaktualności (generator został stworzony w roku 2011).

4.5.2. Wykorzystanie istniejących produktów wykonujących podobne zadanie

O ile nie znaleziono rozwiązań generujących paczki w R, to wykorzystano narzędzia spełniające wymagania konkretnych zadań:

- biblioteka robotpy-cppheaderparser została wykorzystana do parsowania plików nagłówkowych C i otrzymania informacji o funkcjach, do których są tworzone wrappery
- narzędzie CMake zostało wykorzystane w celu kompilacji kodu w C

4.5.3. Automatyzacja publikacji na repozytorium CRAN

Jednym z kroków rozowju projektu było określenie, w jaki sposób odbywa się publikacja paczki na preferowanym repozytorium CRAN oraz zdefiniowanie, jak taki proces można zautomatyzować. Okazało się, że publikowanie paczki na repozytorium CRAN odbywa się poprzez wypełnienie trzyczęściowego formularza internetowego, zarówno w przypadku publikacji pierwszej wesji, jak i każdej kolejnej. Nie znaleźliśmy żadnego API, które pomogłoby zautomatyzować ten proces, tak jak na przykład ma miejsce w przypadku repozytorium Maven dla języka Java. W związku z tym zbadano, jak wygląda publikacja na platformach innych niż CRAN.

W ramach poszukiwań dowiedzieliśmy się, że paczki R mogą być publikowane w następujących miejscach:

- repozytorium paczek CRAN
- repozytorium kodu Github
- jako artefakt w Github Actions
- repozytorium paczek R-Forge
- repozytorium paczek Bioconductor

W przypadku repozytorium CRAN, jak wspomniano powyżej, istnieje problem z automatyzacją publikacji paczki. Wprawdzie problem ten można by rozwiązać poprzez wykorzystanie narzędzi do testów automatycznych stron internetowych, jak na przykład Selenium - za jego pomocą wypełniać odpowiednie pola formularza publikacji paczki. Jednakże takie rozwiązanie może kolidować z regulaminem publikacji na repozytorium CRAN, a konkretniej z punktem: Śubmitting updates should be done responsibly and with respect for the volunteers' time. Once a package is established (which may take several rounds), "no more than every 1–2 months" seems appropriate."[1]. Co więcej, repozytorium CRAN posiada własną listę warunków, jakie musi spełniać paczka, żeby mogła zostać opublikowana, co jest dodatkową przeszkodą.

Jeśli chodzi o przechowywwanie paczki na repozytorium na Githubie, to w taki sposób zwykle przechowuje się kod paczki, która jest w trakcie rozwoju. Oznacza to, że na repozytorium może zostać dodana wersja, która nie jest jeszcze sprawdzona, a zatem może być niestablina albo nawet mieć problemy z działaniem. Dodatkową wadą takiego rozwiązania jest fakt, że w przypadku chęci automatyzacji takiego sposobu publikacji, należałoby wcześniej utworzyć repozytorium na Githubie. Co więcej, do instalacji takiej paczki przez użytkownika potrzebna jest paczka devtools, która jest duża i głównie używana przy tworzeniu paczek. Inną możliwością jest samodzielne zbudowanie paczki na swoim komputerze, jednakże taki sposób wymaga dodatkowej wiedzy od jej użytkownika.

Innym, ale powiązanym rozwiązaniem, jest przetrzymywanie wygenerowanej paczki jako artefakt na Githubie, to znaczy jako efekt uboczny procesu CI w Github Actions. Jednakże artefakty są przechowywane w Github Actions tylko przez pewien okres czasu, co uniemożliwiałoby dostęp do starszych wersji paczki. W przypadku, gdy tak publikowane paczki nie są kopiowane i dodatkowo publikowane na innym repozytorium, starsze wersje paczki byłyby tracone. Co więcej, instalacja takiej paczki byłaby mocno utrudniona: należałoby wejść na odpowiednie repozytorium, paczkę pobrać, a następnie za pomocą odpowiedniej biblioteki R zainstalować na swoim komputerze. Taki sposób instalacji nie spełnia jednego z wymagań postawionych projektowi.

R-Forge jest repozytorium paczek bazującym na systemie kontroli wersji SVN. W związku z tym publikacja na tym repozytorium mogła by być problematyczna, ponieważ projekt biblioteki Libamtrack korzysta z systemu Git. Dodatkowo przy publikacji nowej paczki konieczne jest dodanie jej przez stronę repozytorium. Nie znaleziono na stronie informacji, jak publikować kolejne wersje paczki już kiedyś opublikowanej.

Bioconductor jest konkurencyjnym repozytorium dla CRAN-a, na któym publikowane są paczki z zakresu biologii lub pokrewnych. Ponieważ jest to rozwiązanie konkurencyjne, paczka nie może być jednoczesnie publikowana na tych dwóch repozytoriach, co jest problematyczne w przypadku biblioteki Libamtrack, która kiedyś została opublikowana już na repozytorium CRAN. Co więcej, dodanie paczki na to repozytorium odbywa się poprzez tworzenie issue na Githubie. W tym celu konieczne jest, by paczka posiadała swoje repozytorium na Githubie. Natomiast, podobnie, jak w przypadku przechowywania paczki na repozytorium na Githubie, do instalacji paczki z Bioconductora potrzebna jest wcześniejsza instalacja paczki BiocManager.

Ostatecznie łącznie z Opiekunem projektu postanowiliśmy pozostać przy wyborze repozytorium CRAN z racji posiadania przez niego najłatwiejszego interfejsu do instalacji paczek, do którego nie trzeba instalować nic dodatkowego. Zdecydowaliśmy się również na wycofanie wymagania dotyczącego automatyzacji publikacji paczki na CRAN. Zamiast tego pojawiło się nowe wymaganie, by generowane paczki przygotowywać do publikacji na repozytorium CRAN jak najbardziej, jak to możliwe.

4.5.4. Sprawdzanie poprawności kodu testowego dla typów zmiennoprzecinkowych

W trakcie pisania krótkiego programu sprawdzającego działanie kodu testowego natrafiono na następujący problem: część wartości zwracanych przez funkcje testowe nie zgadzała się z wartością oczekiwaną. Po przestudiowaniu problemu okazało się, że

Podjęto trzy próby rozwiązania tego problemu,.

Pierwszą próbą była zamiana operacji mnożenia na dodawanie w przypadku funkcji przyjmujących parametry tablicowe. Ponieważ wartość zwracana była równa sumie elementów tablicy, w przypadku tablicy wypełnionej jedną stałą wartością oczekiwano, że wartość zwrócona będzie równa iloczynowi tej stałej przez liczbę parametrów. Jednakże w przypadku, gdy ele-

menty tablicy były równe na przykład 1.2, odpowiedni iloczyn i suma nie były sobie równe, co wynika ze skończonej dokładności reprezentacji liczb przez komputer. W związku z tym problem spróbowano rozwiązać poprzez zamianę mnożenia stałej, na faktyczne jej dodawanie w pętli. Mimo że rozwiązanie to poprawiło sytuację, bo mniejsza część testów zwracała błąd, to jednak problem się utrzymywał.

Drugą próbą rozwiązania tego problemu było dopasowanie typów float i double. Stałe zmiennoprzecinkowe są inaczej kodowane w C w przypadku typu float i double. Stałe liczbowe typu float posiadają literę f na końcu zapisów, natomiast stałe liczbowe typu double nie posiadają żadnego dodatkowego znacznika. Dla przykładu, stała 0.4 zapisana w formacie float będzie miała postać 0.4f, a w formacie double postać 0.4. W testach początkowo stosowano tylko format double liczb, skąd też brała się część błędów. Jednakże, po poprawieniu formatów część testów kodu testowego nadal nie przechodziła.

W ramach trzeciej próby rozwiązania problemu zamieniono porównywanie dwóch liczb zmienno przecinkowych na porównywanie ich różnicy do epsilon równego 10^{-4}. Ostatecznie to podejście rozwiązało problem i sprawiło, że się już więcej nie pojawiał.

5. Wyniki projektu

Poniższy rozdział podsumowuje efekty prac nad projektem, w tym opisuje zrealizowane funkcjonalności, możliwe sposoby użycia produktu, ocenę produktu przez zespół i zdobytą wiedzę, a także możliwe obszary dalszego rozwoju i usprawnień.

5.1. Zrealizowane funkcjonalności

W ramach rozwijania projektu zrealizowano następujące funkcjonalności:

- generacja wrapperów funkcji C w R dla podstawowych typów języka C oraz tablic tych typów
- generacja wrapperów - obsługa tablic podstawowych typów języka C
- generacja wrapperów - odróżnianie parametrów wejściowych, wyjściowych oraz zarówno wejściowych jak i wyjściowych na podstawie dokumentacji kodu
- generacja dokumentacji kodu w roxygen2 na podstawie dokumentacji kodu w C w formacie doxygen
- generacja wrapperów - zwracanie wielu parametrów wejściowych jako lista (struktura klucz-wartość)
- kompilacja paczki - automatyczne sprawdzanie wygenerowanego kodu w R przez kompilację i skrypty testowe
- kompilacja paczki - wsparcie dla kodu C korzystającego z zewnętrznych bibliotek, na przykład GSL czy expat (biblioteka do parsowania plików XML) - wymaganie częściowo spełnione. Przy kompilacji paczki można ustawić odpowiednią zmienną kompilacji (dla GSL jest to `PKG_LIBS = -lgsl -lgslcblas -lm`) w pliku makevars (makevars.win na systemie Windows) i paczka skompilowana w taki sposób będzie działać. Jednakże plik ten trzeba utworzyć ręcznie, ponieważ generator nie sprawdza, czy kod posiada zewnętrzne zależności
- przygotowanie wygenerowanej paczki do publikacji na repozytorium CRAN - funkcjonalność spełniona częściowo
- napisanie kodu testowego, na którym można testować generator
- kod testowy - każdy typ zmiennych języka C obsługiwany przez generator powinien wystepować przynajmniej raz jako typ wartości zwracanej, parametru wejściowego i parametru wyjsciowego
- kod testowy - funkcje testowe dla typów tablicowych
- kod testowy - funkcje powinny być różnorodne pod względem liczby parametrów wejściowych i wyjściowych
- kod testowy - parametry wyjściowe i wartości zwracane powinny być w jakiś sposób zależne od parametrów wejściowych, o ile funkcja takie posiada

- sprawdzenie kodu testowego poprzez kompilację i uruchomienie prostego programu testującego
- automatyzacja dwóch powyższych punktów z użyciem Github Actions
- możliwość korzystania z generatora przez każdego, kto posiada konto na Githubie, w tym przez osoby biorące udział w projekcie AmTrack - generator posiada swoje własne repozytorium na Githubie
- możliwość działania powstałej paczki na systemach obsługiwanych przez platformę R
- możliwość działania generatora na systemach operacyjnych wymienionych w powyższym punkcie
- powiadamianie o problemach w procesie działania - w razie wystąpienia błędu generator wypisuje informację o nim. Natomiast jeśli chodzi o automatyzację procesów, to w przypadku wystąpienia błędu całość jest przerywana i dostajemy informację, na którym etapie wystąpił błąd, łącznie z logami, jeśli tym etapem było generowanie wrapperów
- interfejs pozwalający na łatwiejsze używanie generatora - opis w podsekcji 5.2

W związku z tym nie udało się zrealizować następujących funkcjonalności:

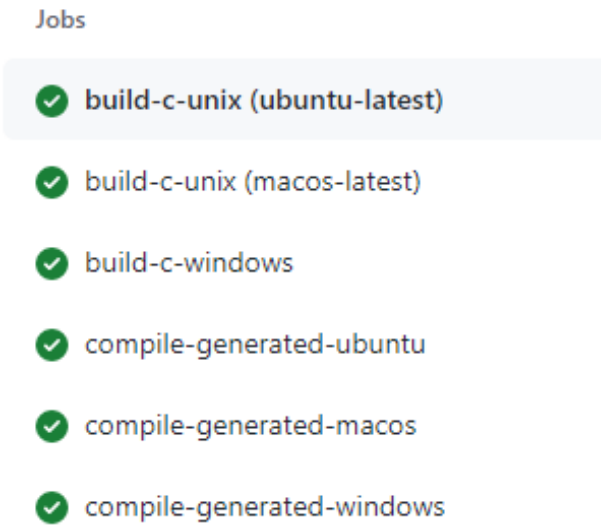
- generacja wrapperów - rozpoznawanie rozmiaru tablic na podstawie dokumentacji
- geracja wrapperów - możliwość podawania tablic (wektorów) bez podawania ich długości
- kompilacja paczki - łączenie skompilowanych źródeł i bibliotek w jedną bibliotekę dynamiczną - niespełnione ze względu na brak spełnienia powyższego wymagania
- automatyzacja publikacji wygenerowanej paczki na repozytorium CRAN - wycofane w trakcie rozwoju projektu
- względnie prosty sposób instalacji opublikowanej paczki - funkcjonalność niespełniona ze względu na wycofanie wymagania dotyczącego publikowania paczki
- skalowalność projektu na inne biblioteki niż testowa

Podsumowując, udało nam się spełnić większość zadanych nam na początku rozwoju wymagań. W przypadku wymagań z powyższej listy, które nie posiadają uzasadnienia za ich niespełnieniem, powodem była zbyt mała ilość czasu.

Rysunki 4, 5, 6, 7, 8 pokazują efekty automatyzacji procesów projektu w Github Actions i są to odpowiednio: lista prac wykonywanych w CI wraz z informacją o ich sukcesie, lista wykonanych procesów składających się na pracę zbudowania i przetestowania kodu testowego wraz z informacją o ich sukcesie, część logów wypisywanych podczas testowania kodu testowego, lista wykonanych procesów składających na pracę wygenerowania, zbudowania i przetestowania paczki wraz z informacją o sukcesie oraz część logów wypisywanych podczas działania generatora na bibliotece testowej.

5.2. Sposób użycia generatora

Sposób użycia generatora został opisany w sekcji 3.3. Szczegółowa instrukcja użycia jest dostępna przez wywołanie skryptu z argumentem `-h`. Listing 8 jest kopią zwracanej wiadomości.



Rysunek 4: Lista prac wykonana w CI po dodaniu nowej wersji generatora z informacją o sukcesie

5.3. Ocena projektu przez zespół

Pomimo że projekt nie pokrywa wszystkich oryginalnych wymagań uważamy, że stanowi on dobry początek i krok w stronę dalszego rozwoju automatyzacji generacji paczek dla języka R na podstawie bibliotek napisanych w języku C. Projekt stanowił również dobrą okazję do poszerzenie naszej wiedzy

Rozwijanie projektu pozwoliło na szersze poznanie języka programowania R, w szczególności jego interfejsu dla kodu napisanego w języku C, na przykład:

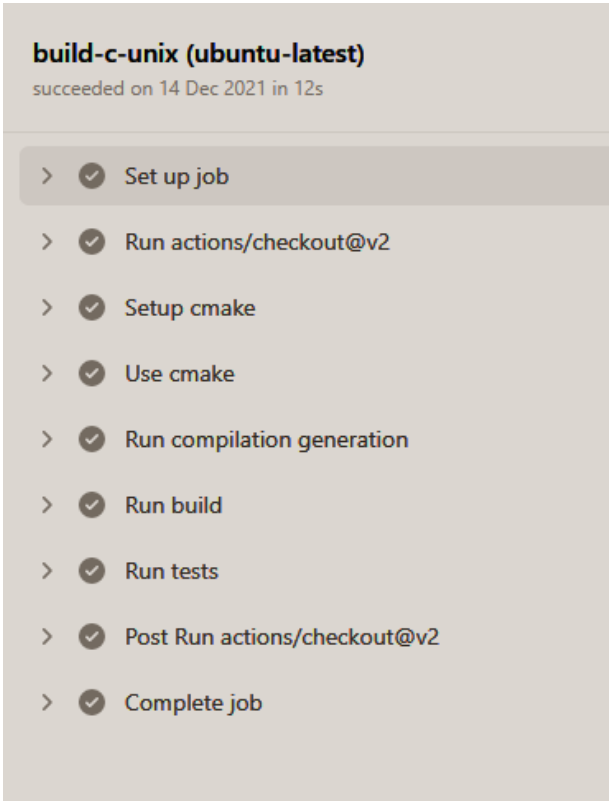
- wewnętrzny sposób przechowywania zmiennych przez środowisko R
- sposoby wywoływania funkcji pochodzących ze skompilowanych bibliotek
- istnienie wielu interfejsów do wywołania zewnętrznych funkcji (w tym niewymieniony we wcześniejszych rozdziałach interfejs Rcpp)

Rozwijanie projektu pozwoliło również na zdobycie wiedzy na temat mechanizmu paczek w R, sposobu ich użycia, tworzenia oraz publikacji. Ponadto praca nad projektem dała nam okazję do zapoznania się z jednym z podstawowych narzędzi Github Actions, jakim jest CI/CD.

5.4. Możliwe dalsze obszary rozwoju

Projekt może zostać rozwinięty i ulepszony o następujące opcje:

- rozszerzenie mechanizmu przygotowywania paczki do opublikowania na repozytorium CRAN - na ten moment mechanizm ten nie jest kompletny. Nie jest na przykład sprawdzane, czy kod paczki narusza ustawienia globalne użytkownika lub czy nie zawiera przypadkiem plików ywkonywalnych i binarnych
- mechanizm generacji paczki w ramach procesu CI na Githubie dla zewnętrznych bibliotek, poprzez dołączenie repozytorium generatora jako submoduł



Rysunek 5: Lista wykonanych procesów składających na pracę zbudowania i przetestowania kodu testowego wraz z informacją o sukcesie

- mechanizm automatycznej publikacji paczki, na przykład na repozytorium CRAN lub R-Forge - jak wspomniano w sekcji 4.5.3, w trakcie rozwijania projektu zrezygnowano z tego wymagania z powodu większego skomplikowania problemu, niż zakładano na początku. Dodanie takiego mechanizmu ułatwiłoby proces publikacji nowej wersji biblioteki

- poprawa wsparcia dla kodu źródłowego C korzystającego z innych bibliotek - na ten moment generator nie zadziała dla kodu źródłowego C, który korzysta z jakiejś zewnętrznej biblioteki bez ręcznego dodania pliku makevars (makevars.win dla Windowsa) z odpowiendio ustawionymi z miennymi kompilajci. Dobrze by było, gdyby generator sam potrafił rozpoznać, czy kod posiada jakieś zewnętrzne zależności i generował taki plik

- wprowadzienie wsparcia dla 64-bitowych liczb całkowitych, np. przez wykorzystanie zewnętrznej paczki bit64[4]

- wprowadzienie wsparcia dla typów innych niż wymienione w tabeli 1 oraz dla typów złożonych, na przykład struktur (patrz listing 3)

- wprowadzienie wsparcia dla tablic o rozmiarach wyrażanych działaniami arytmetycznymi (patrz listing 9)

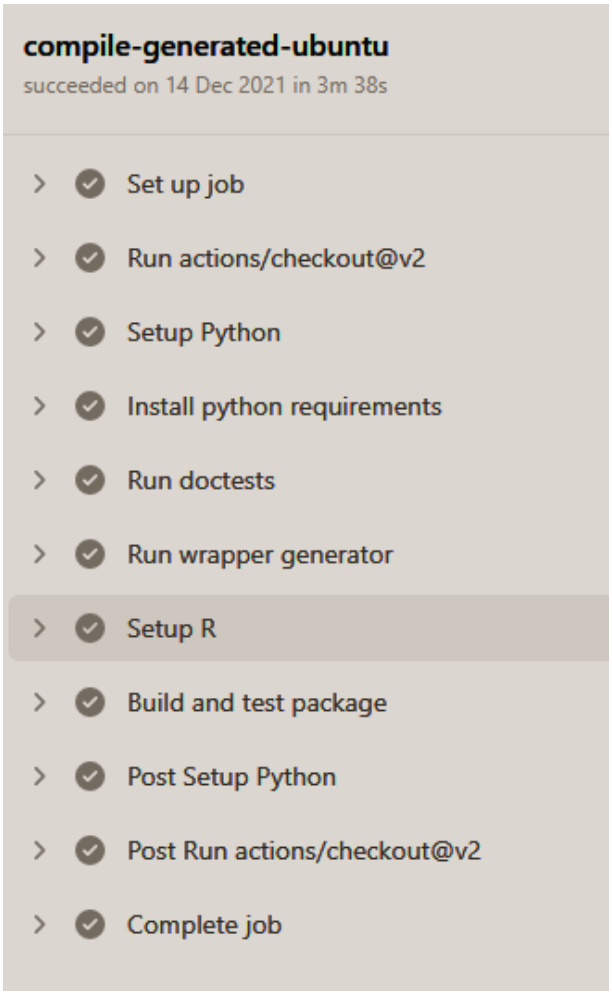
- wsparcie dla kodu używającego napisów kodowanych innym kodem niż ASCII (patrz [12] aneks D, [9] sekcja 7.7) - język ANSI C zapewnia wsparcie dla napisów kodowanych na przykład za pomocą Unicode'u. Dla takich napisów istnieje osobny typ wchar_t

```
▶ Run ./C_samples/build/C_samples
Starting testing...
Testing functions with no parameters...
Printing empty line:

Testing functions with no parameters ended succesfully.
------------------------------------------------------------------------
Proceeding with testing functions with only in parameters...
Printing number 10:
10
Testing functions with only in parameters ended succesfully.
------------------------------------------------------------------------
Proceeding with testing functions with only out parameters...
Printing number 5:
5
Testing functions with only out parameters ended succesfully.
------------------------------------------------------------------------
Proceeding with testing functions with one in and one out parameter...
Printing numbers 5 and 4.5:
5, 4.500000
Testing functions with one in and one out parameter ended succesfully.
------------------------------------------------------------------------
Proceeding with testing functions with one in and two out parameters...
Printing numbers 5, 30 and 0:
5, 30.000000, 0.000000
Testing functions with one in and two out parameters ended succesfully.
```

Rysunek 6: Część logów procesu wykonywania testów kodu testowego

Powyższa lista jest uporządkowana pod względem priorytetu od najwyższego do najmniejszego. Pierwsze cztery podpunkty to niespełnione przez nas wymagania, pozostałe to możliwe rozszerzenia, które odkryliśmy podczas rozwijania projektu.



Rysunek 7: Lista wykonanych procesów składających na pracę wygenerowania, zbudowania i przetestowania paczki wraz z informacją o sukcesie

```
▶ Run python scripts/header_parser.py ReGeneratorTest C_samples/*.h -vv --out-dir out
DEBUG:root:Creating R function one.in.two.out.params.no.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_no_return_function_wrapper (.C)
DEBUG:root:Creating R function one.in.two.out.params.int.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_int_return_function_wrapper (.C)
DEBUG:root:Creating R function one.in.two.out.params.unsigned.int.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_unsigned_int_return_function_wrapper (.C)
DEBUG:root:Creating R function one.in.two.out.params.float.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_float_return_function_wrapper (.C)
DEBUG:root:Creating R function one.in.two.out.params.double.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_double_return_function_wrapper (.C)
DEBUG:root:Creating R function one.in.two.out.params.bool.return.function (.C)
DEBUG:root:Creating C function one_in_two_out_params_bool_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.no.return.function (.C)
DEBUG:root:Creating C function two_in_two_out_params_no_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.int.return.function (.C)
DEBUG:root:Creating C function two_in_two_out_params_int_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.unsigned.int.return.function (.C)
DEBUG:root:Creating C function two_in_two_out_params_unsigned_int_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.float.return.function (.C)
DEBUG:root:Creating C function two_in_two_out_params_float_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.double.return.function (.C)
DEBUG:root:Creating C function two_in_two_out_params_double_return_function_wrapper (.C)
DEBUG:root:Creating R function two.in.two.out.params.bool.return.function (.C)
```

Rysunek 8: Część logów procesu generowania wrapperów

```
usage: header_parser.py [-h] [-o output_dir] [-n namespace] [-l
 ↪  logfile] [-v] [-q] infile [infile ...]

Create wrappers necessary to create an R package for functions
 ↪  declared in a header file(s)

positional arguments:
  infile                C header files (or glob patterns) to be
   ↪  parsed

options:
  -h, --help            show this help message and exit
  -o output_dir, --out-dir output_dir
                        directory to write wrappers to,
                         ↪  (default ./out)
  -n namespace, --namespace namespace
                        path to project namespace file (default
                         ↪  ./NAMESPACE)
  -l logfile, --log-file logfile
                        Redirect logging information to logfile
                         ↪  instead of stderr
  -v, --verbose         Log more information, can be stacked up
   ↪  to 2 times. Concels out with -q
  -q, --quiet           Log less information, can be stacked up
   ↪  to 2 times. Cancels out with -v

Glob patterns supported by this script are:
    ? - matches any single character
    * - matches any string
    ** - matches any path recursively

For example, `?oo.txt` can match foo.txt and boo.txt, `*oo.txt`
 ↪  will additionally match kazoo.txt, and **oo.txt will match
 ↪  all of the above, as well as a/oo.txt and a/b/zoo.txt
```

Listing 8: Instrukcja użytku generatora, w języku angelskim ze względu na szerokie wykorzystanie tego języka branży IT (w tym repozytorium biblioteki Amtrack)

```
/**
 * Calculates the determinant of a square matrix
 * @param n size of the matrix
 * @param A the matrix (array of size n*n)
 * @return the determinant of A
 */
double det(int n, double *A);
```

Listing 9: Przykładowa funkcja, która nie otrzymałaby poprawnie działającego wrappera

Spis tabel

Spis rysunków

Spis kodów źródłowych

Materiały źródłowe

[1] *CRAN Repository Policy*. https://cran.r-project.org/web/packages/policies.html, dostęp z dnia [2021-12-07].

[2] Dokumentacja funkcji .c [ang]. https://stat.ethz.ch/R-manual/R-devel/library/base/html/Foreign.html, dostęp z dnia [2021-11-09].

[3] Dokumentacja modułu glob [ang]. https://docs.python.org/3.8/library/glob.html, dostęp z dnia [2021-11-09].

[4] Dokumentacja paczki bit64 [ang.]. https://cran.r-project.org/web/packages/bit64/index.html, dostęp z dnia [2021-12-07].

[5] Dokumentacja pliku nagłówkowego r_ext/rdynload.h z api środowiska r [ang]. https://lukasstadler.github.io/RAPI/html/_rdynload_8h.html, dostęp z dnia [2021-11-09].

[6] Dokumentacja pliku nagłówkowego rinternals.h z api środowiska r [ang]. https://lukasstadler.github.io/RAPI/html/_rinternals_8h.html, dostęp z dnia [2021-11-09].

[7] Strona biblioteki robotpy-cppheaderparser [ang]. https://pypi.org/project/robotpy-cppheaderparser/, dostęp z dnia [2021-11-09].

[8] *Uncover the R Applications – Why Top Companies are using R Programming*. https://data-flair.training/blogs/r-applications/, dostęp z dnia [2022-01-04].

[9] *Writing R Extensions*. https://cran.r-project.org/doc/manuals/R-exts.html, dostęp z dnia [2021-12-07].

[10] S. Greilich, L. Grzanka, N. Bassler, C. E. Andersen, and O. Jäkel. Amorphous track models: A numerical comparison study. *Radiation Measurements*, 45(10):1406–1409, Dec 2010.

[11] R. Herrmann, S. Greilich, L. Grzanka, and N. Bassler. Amorphous track predictions in 'libamtrack' for alanine relative effectiveness in ion beams. *Radiation Measurements*, 46(12):1551–1553, Dec 2011.

[12] ISO. ISO/IEC C standard. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf, dostęp z dnia [2021-12-07].

[13] H. Wickham and J. Bryan. *R packages*, chapter 8. Package metadata. O'Riley, 2015. Dostęp online: https://r-pkgs.org/description.html, dostęp z dnia [2022-02-16].