POLITECHNIKA WROCŁAWSKA

WYDZIAŁ ELEKTRONIKI

KIERUNEK:

SPECJALNOŚĆ:

PRACA DYPLOMOWA

MAGISTERSKA

Tytuł pracy w języku polskim

Tytuł pracy w języku angielskim

autor : IMIĘ i NAZWISKO

Opiekun pracy:

tytuł, imię i nazwisko, jednostka

OCENA PRACY:

WROCŁAW ……

[Implementacja 4](#__RefHeading___Toc6486_842470764)

[Użytkowanie aplikacji 4](#__RefHeading___Toc6488_842470764)

[Wykorzystanie gui\_manager.py 5](#__RefHeading___Toc6490_842470764)

[Wykorzystanie console\_fetcher.py 18](#__RefHeading___Toc6492_842470764)

[Artefakty 20](#__RefHeading___Toc6494_842470764)

[Biblioteka 25](#__RefHeading___Toc6496_842470764)

[Warstwowa budowa biblioteki 27](#__RefHeading___Toc6498_842470764)

[Moduły pomocnicze 29](#__RefHeading___Toc6500_842470764)

[GUI 31](#__RefHeading___Toc6502_842470764)

[Warstwa wykonawców 73](#__RefHeading___Toc6504_842470764)

[Właściwy algorytm 78](#__RefHeading___Toc6506_842470764)

[Właściwy algorytm – część indukcyjna 105](#__RefHeading___Toc6508_842470764)

[Warstwa rdzenia 127](#__RefHeading___Toc6510_842470764)

[Warstwa danych 131](#__RefHeading___Toc6512_842470764)

1. Implementacja

Aplikacja została zaimplementowana jako biblioteka języka Python w wersji 3.4. Może ona zatem być wykorzystywana w dowolnym środowisku posiadającym dostęp do interpretera języka Python w wersji 3.4 lub nowszej oraz niezbędne biblioteki.

Biblioteki niezbędne do wykorzystania aplikacji zostały wymienione w pliku „requirements.txt” (znajdującym się w głównym katalogu projektu). Są to:

PyHamcrest – wymagany do uruchomienia testów;

Matplotlib – wykorzystywany do rysowania diagramów procesu uczenia gramatyki;

Mock – wykorzystywany do tworzenia atrap obiektów (ang. Mock Object) na potrzeby testów modułowych oraz komponentowych;

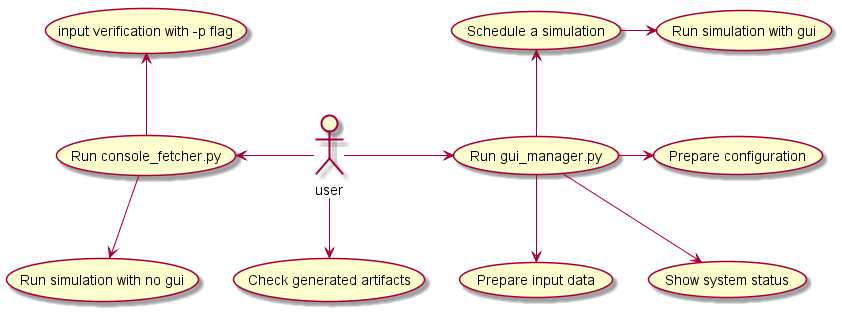
Psutil – wykorzystywany do pobierania informacji o obecnym obciążeniu procesora i zajętej pamięci niezależnie od systemu operacyjnego;

Wszystkie te biblioteki możemy bardzo łatwo zainstalować przy pomocy narzędzia do instalowania modułów pip.

<<opis pip>>

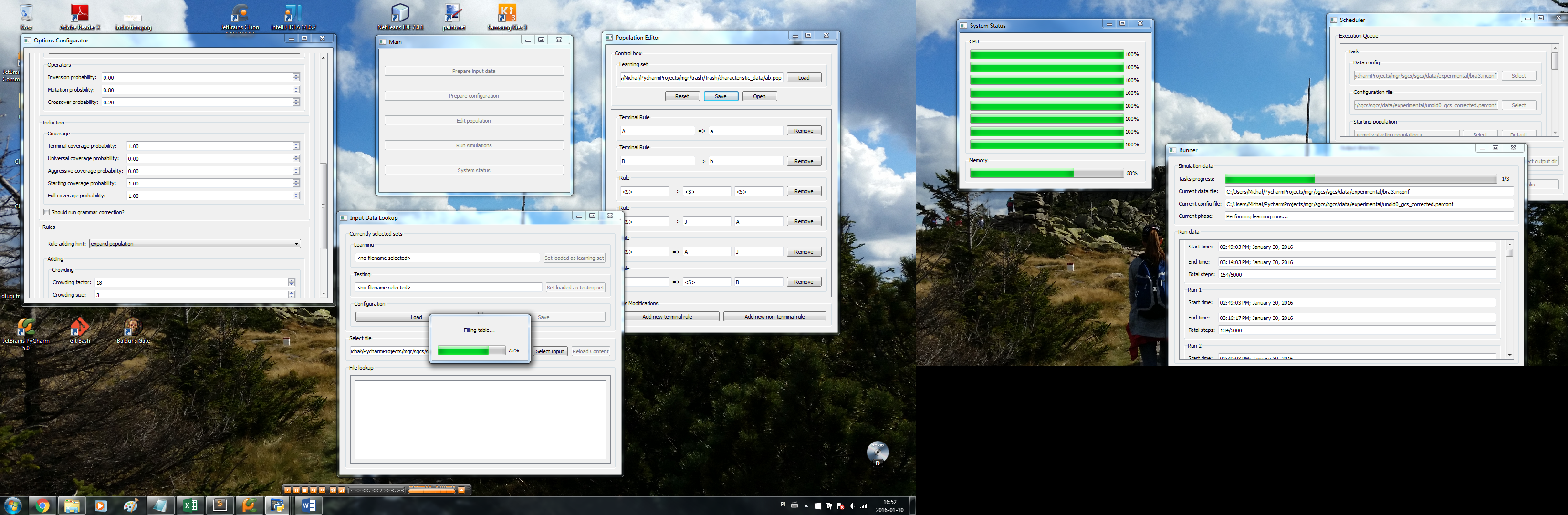
1. Użytkowanie aplikacji

Dodatkowo do biblioteki zostały dołączone skrypty umożliwiające uruchomienie klasycznego modułu GCS/sGCS bez konieczności pisania własnego kodu. Aplikację możemy uruchomić na jeden z dwóch sposobów – z wykorzystaniem skryptu gui\_manager.py albo console\_fetcher.py.



1. Wykorzystanie gui\_manager.py

GCS oraz sGCS to algorytmy o wysokim poziomie złożoności. Posiadają kilkadziesiąt parametrów, mogą przyjmować różne dane wejściowe, zaś procesy generowania gramatyki mogą być żmudne i czasochłonne. Dlatego też w trakcie projektowania aplikacji zdecydowano się na zastosowanie wielookienkowego GUI, mając na celu ułatwienie wykorzystania wielu monitorów w pracy oraz łatwe przygotowywanie zbiorów danych do kolejnych testów, podczas gdy poprzednie są jeszcze w trakcie wykonywania.



Poniżej znajduje się omówienie kolejnych okien dostępnych w stworzonej aplikacji oraz prawidłowy sposób ich wykorzystania.

Menu główne

Po uruchomieniu skryptu powinno pojawić się okno menu głównego. Zawiera ono listę przycisków, które można uruchomić. Są to:

Prepare input data – podgląd plików ze zdaniami gramatyki napisanymi w formacie adabingo oraz przygotowywanie zestawów zbiorów uczących – testowych;

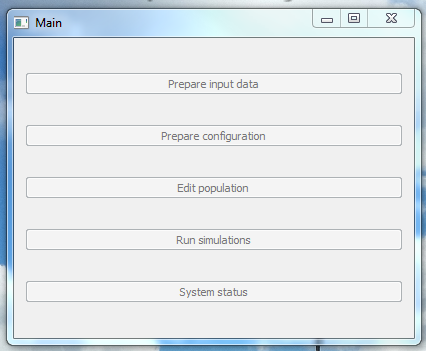
Prepare configuration – umożliwia ustawienie pożądanych parametrów algorytmu i zapisanie tych ustawień w postaci pliku konfiguracyjnego. Dzięki temu jest możliwe późniejsze uruchamianie testów dla różnych kombinacji danych wejściowych i parametrów algorytmu bez konieczności ponownego ich wprowadzania;

Edit population – naciśnięcie tego przycisku spowoduje wyświetlenie edytora populacji. Z jego pomocą możliwy jest wygodny podgląd populacji wygenerowanej przez algorytm, czy też na przykład przygotowanie ręcznie całości lub fragmentu populacji początkowej;

Run simulations – Przycisk ten prowadzi do menu Schedulera, umożliwiającego przygotowanie zestawu testów do uruchomienia;

System status – wynikiem naciśnięcia tego przycisku jest otwarcie okna statusu systemu – okno to pokazuje obciążenie każdego z rdzeni procesora oraz ilość zajętej pamięci RAM. Narzędzie to umożliwia dokonanie oceny czy nasz algorytm w optymalny sposób wykorzystuje dostępne zasoby obliczeniowe.

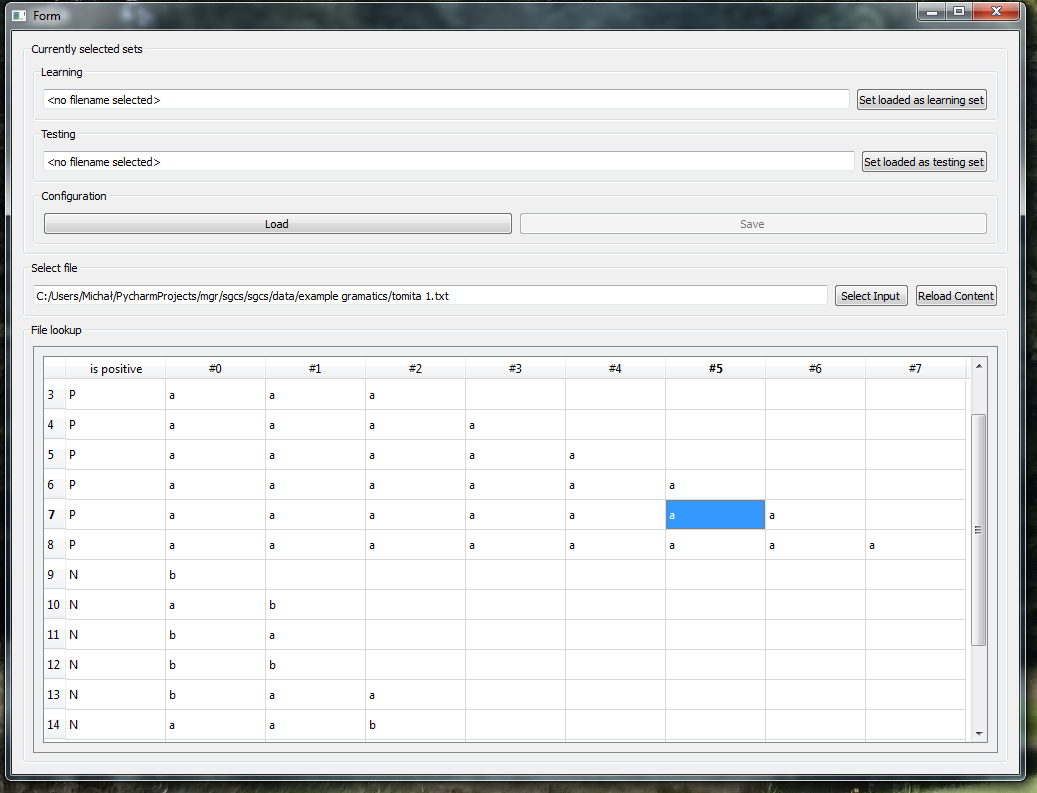
Każde z wymienionych okien jest singletonem – jeżeli jedno gdzieś istnieje, to nie można w ramach tej samej aplikacji otworzyć drugiego takiego samego (co równocześnie nie przeszkadza w otwarciu okna innego typu).



Prepare input data

Okno to umożliwia wczytanie pliku wejściowego z gramatyką. Można tego dokonać przy pomocy przycisku Select Input. Wówczas program załaduje wskazany plik do pamięci, podda go dokładnie takiemu samemu procesowi parsowania, jaki będzie miał miejsce przy uruchomieniu algorytmu, po czym wyświetli wszystkie występujące w nim zdania w formie czytelnej tabeli. Jeżeli plik jest duży, pojawi się okienko dialogowe informujące nas o postępie operacji oraz szacowanym postępem wypełniania tabeli. Aplikacja zakłada, że dane wejściowe są zawsze podane w formacie adabingo, aczkolwiek nie wykorzystuje wszystkich informacji oferowanych przez ten format, w wyniku czego zawartość pierwszego wiersza oraz długość zdania w kolejnych nie są weryfikowane ani wykorzystywane. Niemniej nie są to elementy opcjonalne i muszą w podanym pliku wystąpić; w przeciwnym wypadku mozliwe jest uzyskanie błędnie sparsowanego pliku. Można spowodować odświeżenie zawartości tabeli (na przykład w celu upewnienia się, że po dokonaniu modyfikacji zewnętrznym programem dane są nadal prawidłowo parsowane przez aplikację) naciskając przycisk Reload Content.

Po załadowaniu pliku możemy oznaczyć go jako zbiór uczący lub testowy (przyciski Set loaded as learning set oraz Set loaded as testing set). Po oznaczeniu pliku jego ścieżka bezwzględna pojawia się odpowiednio w polu Learning albo Testing. Kiedy wybierzemy oba zbiory, odblokowaniu ulega przycisk Save – możemy taką konfigurację zapisać jako plik konfiguracji \*.inconf. Plik ten będzie zawierał ścieżki bezwzględne do obu podanych zbiorów (Learning oraz Testing).



Prepare configuration

Okno to umożliwia przygotowanie zestawu parametrów dla algorytmu. W pierwszej kolejności warto wybrać najpierw rodzaj planowanego algorytmu. Pole Template jest odpowiedzialne za określenie czy konfiguracja jest planowana dla sGCS, czy GCS. Ze względu na fakt posiadania innych parametrów drzewo konfiguracji tych algorytmów nieco się różni i zmiana wartości tego pola może spowodować zresetowanie wartości pozostałych pól. Poniżej zostały opisane poszczególne pola.

Reset – przycisk powodujący zresetowanie pól do wartości domyślnych.

Save – przycisk umożliwiający zapisanie obecnego stanu konfiguracji.

Open – przycisk umożliwiający wczytanie wcześniej zapisanego pliku ustawień.

Template – rodzaj stosowanego algorytmu. Akceptowane wartości: sGCS i GCS (neg-sGCS to sGCS z zastosowaniem odpowiedniej funkcji przystosowania).

Should evolution be run? – odznaczenie tego pola spowoduje wyłączenie algorytmu genetycznego – gramatyki będą tworzone w oparciu wyłącznie na samym mechanizmie indukcji.

Max algorithm runs – maksymalna ilość uruchomień cyklu uczącego, jaką ma wykonać aplikacja.

Max algorithm execution time – ograniczenie umożliwiające zakończenie pojedynczego uruchomienia uczącego, jeżeli ten przekroczył podany czas. Jeżeli krokowi uczenia zabraknie czasu, w ten sposób, to jest traktowany jakby przekroczył ograniczenie maksymalnej ilości kroków indukcyjnych.

Max evolution steps – maksymalna ilość kroków ewolucyjnych, jaką może wykonać aplikacja w pojedynczym uruchomieniu uczącym.

Satisfying fitness – przy jakiej wartości parametru fitness proces uczenia uzna gramatykę za wyuczoną (np. jeżeli ustawimy tę wartość na 100%, to uruchomienie uczące zostanie uznane za sukces tylko w przypadku poprawnego sklasyfikowania wszystkich zdań).

Statistics configuration – Tutaj można wybrać rodzaj zastosowanej funkcji fitness. Obecnie w systemie są dostępne dwie funkcje fitness – Classical (przedstawiona w Unold1) oraz Pasieka (przedstawiona w Pasieka). Wybór ten decyduje również o tym, czy będziemy dokonywali parsowania zdań negatywnych w procesie uczenia – funkcja Pasieka wykorzystuje jedynie informacje zdobyte w wyniku parsowania zdań pozytywnych, zatem w przypadku jej wybrania zdania negatywne będą ignorowanie, klasyczna wykorzystuje je w procesie uczenia, więc będą one wykorzystane. W trybie GCS mamy możliwość wykorzystania wyłącznie funkcji klasycznej, w trybie sGCS obu.

Selectors – W bloku opisującym algorytm genetyczny istnieje możliwość wybrania do dziesięciu różnych selektorów. Ilość selektorów wiąże się z ilością reguł branych pod uwagę w procesie działania algorytmu genetycznego oraz wiąże się z arnością operatorów genetycznych (co zostanie dokładniej opisane w sekcji omawiającej moduł sgcs.evolution.evolution\_operators). Z selektorów mamy do wyboru selekcję turniejową, losową oraz metodą ruletki.

Operators – Tutaj można nadać prawdopodobieństwo uruchomienia każdego z operatorów genetycznych. Dostępne operatory to operator krzyżowania, mutacji oraz inwersji

Induction – Ta część okna konfiguracyjnego daje możliwość ustalenia prawdopodobieństw operatorów pokrycia. Zaproponowano ujednolicony model parametrów operatorów pokrycia – wszystkie są wyrażone za pomocą prawdopodobieństwa. Więcej szczegółów na ten temat w omówieniu modułu sgcs.induction.coverage\_operators.

Should run grammar corrections? – zaznaczenie tego pola spowoduje usunięcie produkcji nieosiągalnych oraz nieproduktywnych na końcu każdego etapu indukcji gramatyki (tj. tuż przed uruchomieniem algorytmu genetycznego).

Rules adding hint – W tym miejscu możemy sprecyzować ograniczenia na rozmiar zbioru – jeżeli to zrobimy, to wówczas ilość reguł nieterminalnych od chwili uruchomienia algorytmu nie ulegnie zmianie i będzie utrzymywana na stałym poziomie. Dostępne opcje to None (wówczas ta decyzja jest zależna od wartości domyślnej operatorów pokrycia), expand population (ścisk jest wówczas stosowany tylko podczas działania algorytmu ewolucyjnego, operatory pokrycia rozszerzają populację), control population size (elitism DISABLED) – operatory pokrycia dodają nowe reguły ze ściskiem, control population size (elitism ENABLED) dodatkowo uwzględnia mechanizm elityzmu. Słowo hint w przypadku tego pola jest ważne – mianowicie reguła ta zostanie nadpisana, jeżeli w jej wyniku miałoby nastąpić zwiększenie rozmiaru populacji powyżej wartości maksymalnej (patrz Max non terminal rules population size poniżej).

Crowding – w bloku tym mamy dwie kontrolki – Crowding factor oraz Crowding size. Pierwszy z nich decyduje o rozmiarze populacji ścisku (współczynnik cf), drugi o rozmiarze podpopulacji, z której wybieramy każdego członka populacji ścisku (współczynnik cs).

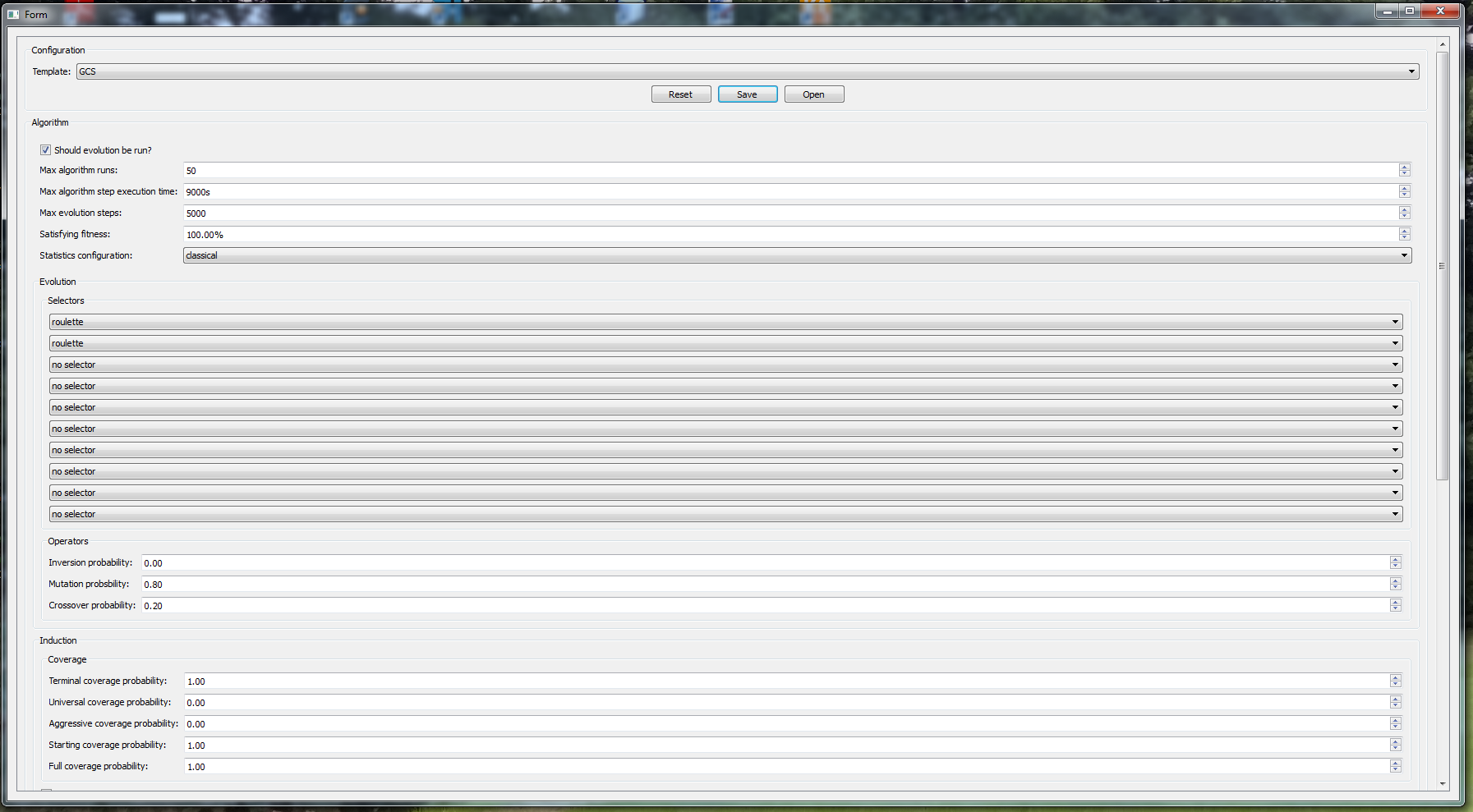
Elitism – na tym etapie możliwe jest podjęcie decyzji na temat elityzmu stosowanego w aplikacji. Możemy go włączyć lub wyłączyć (pole Is elitism used?), można tu też ustalić rozmiar elity.

Max non terminal rules population size: Maksymalny, nieprzekraczalny rozmiar nieterminalnej części populacji. Wartość ta nie może być w algorytmie nigdy przekroczona i jeżeli by to miało nastąpić, to taki precedens ma priorytet nad ustawieniami ścisku i nawet jeżeli jest on domyślnie wyłączony, zostanie zastosowany.

Starting population size – rozmiar populacji reguł przed uruchomieniem algorytmu. Na początku działania algorytmu populacja będzie się składać z reguł dostarczonych w pliku populacji początkowej, powiększonej ewentualnie o losowe reguły w ilości brakującej do tej wartości.

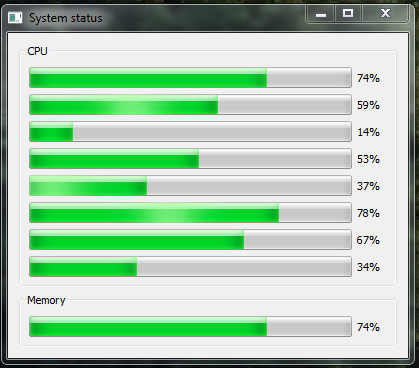
Max non terminal symbols – ilość symboli nieterminalnych, która może wystąpić w algorytmie. Jest to wartość stosunkowo ważna – od rozmiaru tego parametru zależy w końcu przedział symboli, na którym dokonujemy wszelkich operacji losowych. Im mniejszy, tym częściej przy konieczności losowania symbolu będziemy trafiać na te posiadające już jakieś reguły. Z kolei większe wartości pozwolą nam na generowanie bardziej złożonych gramatyk. Ponieważ symbole specjalne (tj. symbol startowy oraz uniwersalny) posiadają specjalne oznaczenia, zatem parametr ten nie posiada górnego ograniczenia.

Pliki konfiguracyjne posiadają rozszerzenie \*.parconf.



System status

Okno to pełni funkcję czysto diagnostyczną, jego zadaniem jest pokazywanie obecnego obciążenia wszystkich rdzeni procesora oraz zużycie pamięci. Informacja ta jest przydatna, gdyż każde uruchomienie cyklu uczącego jest niezależne od pozostałych i możliwe jest puszczenie ich w niezależnych procesach w celu maksymalnego wykorzystania maszyny, na której algorytm został uruchomiony. Dzięki takiemu podglądowi jesteśmy w stanie ocenić czy algorytm w optymalny sposób wykorzystuje zapewnione mu zasoby. Uruchomienie niektórych zbiorów danych może spowodować wielogodzinne stuprocentowe obciążenie procesora, z czym należy się liczyć podczas uruchamiania algorytmu. Jeżeli nasz sprzęt nie jest przygotowany na takie obciążenia (np. nie dysponuje wystarczająco sprawnym układem chłodzącym) uruchamianie algorytmu nie jest wskazane, jako że bez przedsięwzięcia dodatkowych środków ostrożności może ono doprowadzić do wyłączenia komputera, a nawet do trwałego uszkodzenia sprzętu. Biblioteka zawiera jednowątkową wersję algorytmu, aczkolwiek do interfejsu graficznego jest obecnie podpięty jedynie wariant wielowątkowy.



Edit population

Z pomocą przedstawionego poniżej okna możliwe jest zarządzanie populacjami. Można przy jego pomocy otworzyć istniejący plik populacji \*.pop (uzyskany na przykład w wyniku działania algorytmu), czy też utworzyć taką populację od podstaw (na przykład w celu wykorzystania jej w algorytmie jako populacji początkowej). Należy tutaj pamiętać o tym, że wewnątrz aplikacji celem przyspieszenia obliczeń wszystkie symbole gramatyki posiadają wewnętrzne identyfikatory będące liczbami całkowitymi i na potrzeby komunikacji z użytkownikiem są tłumaczone przez warstwę pośrednią (jest to jedna z kluczowych funkcjonalności modułu sgcs.datalayer). W celu prawidłowej obsługi reguł przez poniższe narzędzie niezbędne jest zachowanie kontekstu reguł – umożliwia to plik zbioru uczącego. Jeżeli w jakikolwiek sposób zmodyfikujemy nasz zbiór uczący, dodając, usuwając, modyfikując lub zmieniając kolejność zdań, obiekty populacji mogą przestać być prawidłowo interpretowane. Dlatego też, jeżeli planujemy dokonać jakichkolwiek zmian w zestawie zdań uczących, to po ich dokonaniu należy ponownie wygenerować powiązane pliki \*.pop, lub też przynajmniej upewnić się, że ich zawartość po wczytaniu jest zgodna z naszymi oczekiwaniami.

Ustalenie kontekstu

W celu ustalenia kontekstu populacji należy nacisnąć przycisk Load i wybrać plik, co powinno poskutkować odblokowaniem przycisków Save i Open przy właściwie sformatowanym pliku danych wejściowych, lub komunikat o błędzie w przypadku wystąpienia błędu w trakcie parsowania. Bez ustalenia kontekstu populacji nie będzie możliwe jej wczytanie lub zapisanie, gdyż pliki populacji wykorzystują wewnętrzną reprezentację reguł i podczas odczytu oraz zapisu musi nastąpić tłumaczenie danych na przyjazne człowiekowi i odwrotnie.

Modyfikacje populacji

Populację można zmodyfikować przy pomocy przycisków Remove, Add new terminal rule oraz Add new non-terminal rule. W przypadku reguł nieterminalnych w każdym okienku należy podać identyfikator symbolu. Dla reguł terminalnych po lewej stronie produkcji podajemy identyfikator symbolu, po prawej zaś symbol terminalny. Identyfikator symbolu rządzi się następującymi prawami:

<identyfikator\_symbolu> := <identyfikator\_symbolu\_nieterminalnego> | <identyfikator\_symbolu\_specjalnego>

<identyfikator\_symbolu\_specjalnego> := ‘<S>' | ‘<U>’

<identyfikator\_symbolu\_nieterminalnego := <ciag\_wielkich\_liter>

<ciag\_wielkich\_liter> := <wielka\_litera> | <wielka\_litera> <ciag\_wielkich\_liter>

<wielka\_litera> := ‘A’...’Z’

<symbol\_terminalny> := dowolny ciąg znaków niebędących znakami białymi

Przykłady identyfikatorów symboli nieterminalnych:

A, H, AABC

Należy zauważyć, iż w populacji może istnieć więcej symboli, niż jest znaków alfabetu. W pierwszej kolejności algorytm wykorzystuje do reprezentacji pojedynczą wielką literę alfabetu (A, B i tak dalej), w dalszej kolejności następuje dołożenie kolejnej litery, lub zwiększenie litery na kolejnej pozycji (tj. mamy A, B, …, Y, Z, AA, AB, …, AY, AZ, BA, BB, …). Ludzka reprezentacja zezwala na dowolne symbole z powyższego zbioru, aczkolwiek należy pamiętać, że algorytm wykorzystuje podczas swojego działania jedynie k pierwszych symboli (gdzie k to ilość symboli terminalnych zdefiniowana w danych konfiguracyjnych \*.parconf ). Przy pomocy <S> oznaczamy startowy symbol gramatyki, przy pomocy <U> symbol uniwersalny.

Zapisywanie/otwieranie populacji

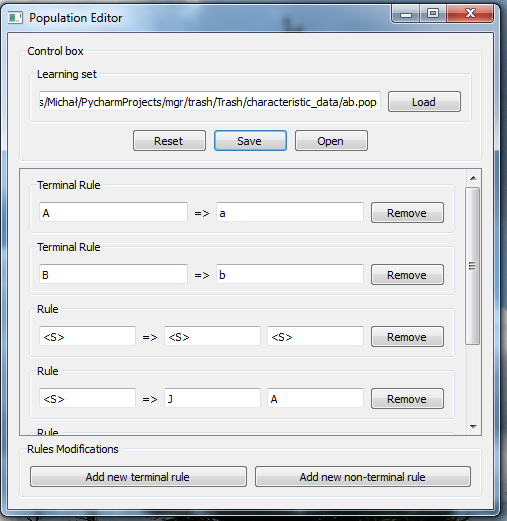
Przy pomocy przycisków Save / Open jest możliwe zapisanie lub wczytanie istniejącej populacji.

Ponowne generowanie plików populacji

Jeżeli dokonujemy zmian w zbiorze uczącym, możemy wygenerować populację ponownie, podejmując następujące kroki:

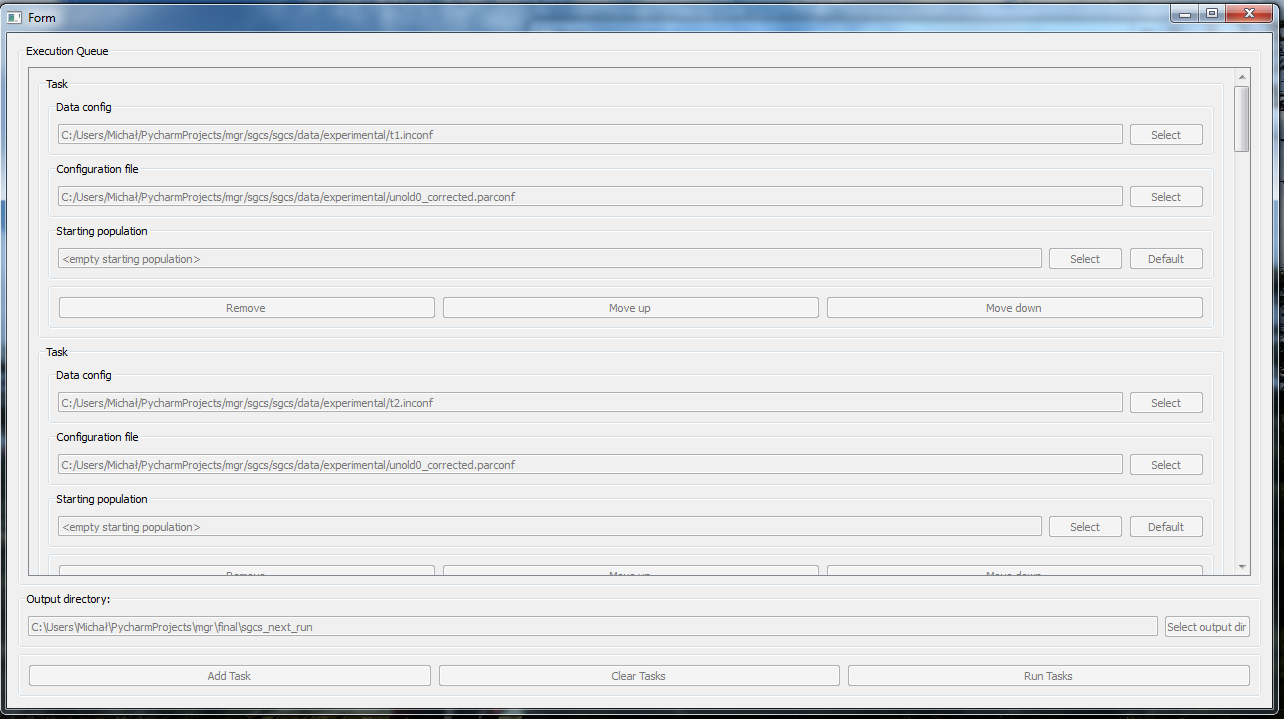
1. Wczytujemy stary zbiór uczący.
2. Otwieramy powiązany z nim plik populacji.
3. Wczytujemy nowy zbiór uczący.
4. Zapisujemy otwartą populację.

Tym sposobem jest możliwe dokonanie konwersji reguł populacji pomiędzy różnymi zbiorami uczącymi.



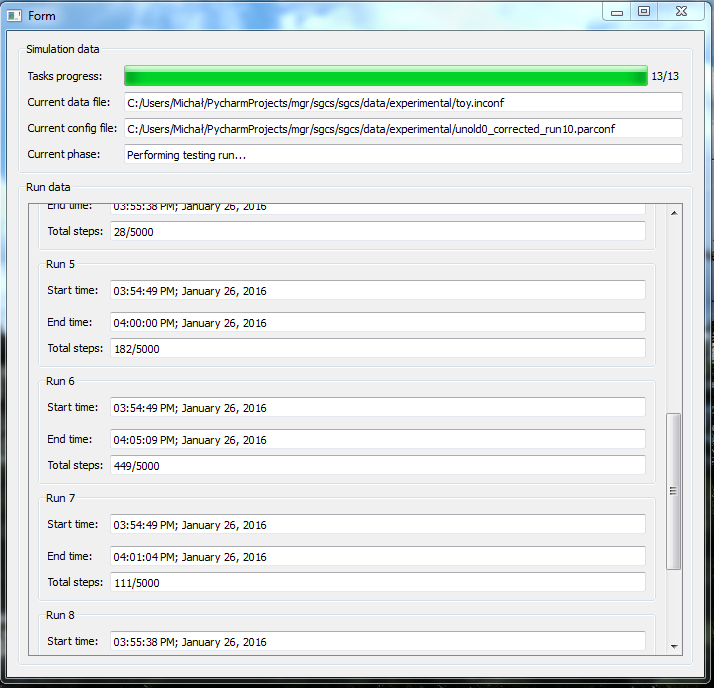
Scheduler

W menu tym można przygotować zestaw badań do przeprowadzenia. Naciskając przycisk Add Task, dodajemy kolejne zadanie obliczeniowe. Przez zadanie rozumiemy pełne odpalenie algorytmu (cykl uczący i testowy) dla zadanego pliku danych wejściowych (skonstruowanego wcześniej przy pomocy Prepare input data), konfiguracji (przygotowanego przy pomocy Prepare configuration) i startowej populacji. Jeżeli w trakcie działania programu trzeba będzie coś zmienić, możliwe jest przerwanie wykonania, usunięcie obliczonych, poprawienie pożądanych rzeczy i wreszcie ponowne wystartowanie procesu obliczeń. Ścieżkę do folderu z danymi wynikowymi ustalamy poprzez naciśnięcie przycisku Select output dir. Folder ten zostanie w całości wyczyszczony na wczesnym etapie działania algorytmu, więc należy wybrać go z rozwagą w celu uniknięcia przypadkowego skasowania ważnych danych! Można również zmieniać kolejność wykonania zadań przy pomocy przycisków Move up, Move down.



Runner

Zadaniem tego okna jest poinformowanie użytkownika o obecnym stanie obliczeń algorytmu. W Simulation data można znaleźć szczegółowe informacje na temat obecnie realizowanego zadania, takie jak nr obecnie wykonywanego zadania, ilość wszystkich zadań, ścieżkę do pliku z danymi wejściowymi, konfiguracją oraz obecnie wykonywany etap prac (patrz omówienie modułu sgcs.gui.runner). W obszarze Run data znajduje się osobna rubryka dla każdego cyklu uczącego, z datą początkową, końcową oraz ilością wykonanych kroków ewolucyjnych. Kiedy wszystkie zadania zostaną wykonane, na ekranie pojawi się informujący nas o tym odpowiedni komunikat. Wszystkie artefakty generowane przez algorytm są zbierane na końcu każdego zadania, a następnie zapisywane do sprecyzowanego w menu Schedulera katalogu wyjściowego. Każde zadanie posiada osobny katalog numerowany w kolejności zgodnej z kolejnością zadań w Schedulerze.



1. Wykorzystanie console\_fetcher.py

Drugi skrypt uruchamiający aplikację został napisany podczas uruchamiania testów w chmurze. Jego powstanie było związane z jednej strony z trudnościami w postawieniu pełnego środowiska graficznego w chmurze oraz z drugiej strony w celu stworzenia prostego konsolowego interfejsu, który można by wykorzystywać w skryptach konsolowych. Jego składnia jest następująca:

console\_fetcher [-p] –o output\_dir [-s] [input\_file config\_file [starting\_population]]

Parametr –p oznacza, że nie algorytm nie uruchomi się, a jedynie wyśle na standardowe wyjście informację o przygotowanej konfiguracji (dzięki czemu możliwe jest wcześniejsze upewnienie się, że skrypt wywołano z właściwymi parametrami). Parametr –o jest obowiązkowy – po nim podajemy ścieżkę do folderu, w którym będą zapisywane artefakty. Następnie należy podać dowolną ilość naprzemiennie występujących plików z danymi wejściowymi i plików konfiguracji. Jeżeli chcielibyśmy dodatkowo podać konfigurację początkową, należy wówczas podać również flagę –s, która sprawi, że algorytm nie będzie konstruował zadań na podstawie dwójki, lecz trójki parametrów. Te trzy parametry to dane wejściowe, konfiguracja i startowa populacja. Jako że całość może wydawać się dosyć skomplikowana, poniżej zostało przedstawione kilka scenariuszy uruchomienia skryptu.

console-fetcher –o ~/data t1.inconf sgcs.parconf t2.inconf gcs.parconf

Powyższe polecenie uruchomi algorytm o konfiguracji sgcs.parconf (czyli najprawdopodobniej sgcs) dla zbioru tomita 1, a następnie gcs dla tomita 2. Artefakty z działania obu algorytmów zostaną zapisane w folderze ~/data.

console-fetcher –p –o ~/data t1.inconf sgcs.parconf t2.inconf gcs.parconf

Flaga –p spowoduje, że program wyświetli tylko informację o tym co by zrobił w przypadku uruchomienia tej samej komendy bez flagi –p.

console-fetcher –s –o ~/data t1.inconf sgcs.parconf t1.pop t2.inconf gcs.parconf t2.pop

Wymienione wcześniej algorytmy rozpoczną działanie odpowiednio z populacją początkową t1.pop oraz t2.pop .

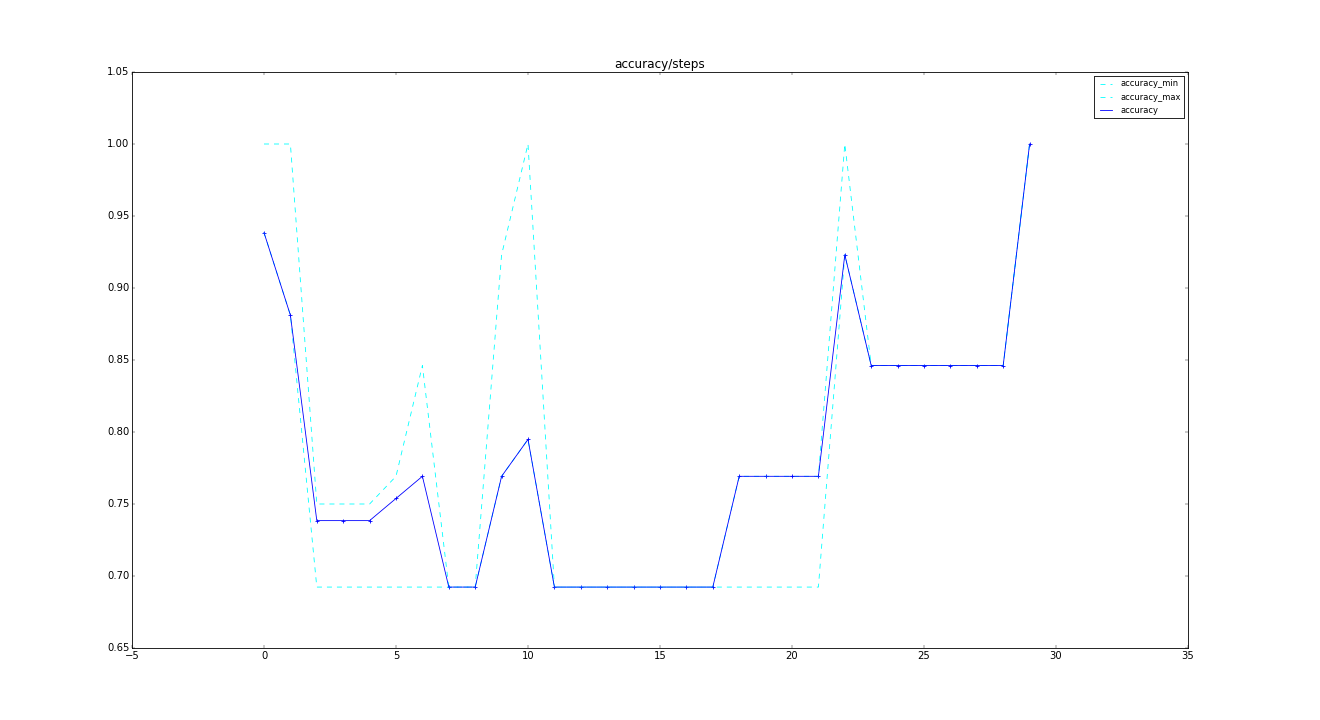
console-fetcher –s –o ~/data t1.inconf sgcs.parconf t1.pop t2.inconf gcs.parconf

Skrypt otrzymał złe argumenty (pomimo flagi –s drugie z planowanych uruchomień nie posiada jawnie podanego pliku populacji. Zostanie wyświetlone ostrzeżenie, po czym skrypt zakończy działanie.

1. Artefakty

Wynikiem działania algorytmu jest wygenerowanie katalogu dla każdego zdefiniowanego zadania. Poniżej została omówiona typowa zawartość takiego katalogu:

pliki \*.png (accurracy.png, fallout,png, falsediscovery.png, falseomisiion.png, missrate.png, negpredictive.png, precision.png, sensivity.png, specifity.png, fitness.png) – są to diagramy średnich dla wszystkich uruchomień cyklu uczącego miar macierzy pomyłek (ang. confussion matrix) od kroku ewolucyjnego. Można na nich znaleźć również wartość minimalną oraz maksymalną (przerywana linia).



final\_population.pop – W tych plikach znajduje się mało czytelny dla ludzkiego oka zapis stanu populacji wykorzystanej w cyklu testowym. Pliki są zapisywane w formacie json. Stosuje się w nich następującą reprezentację wewnętrzną:

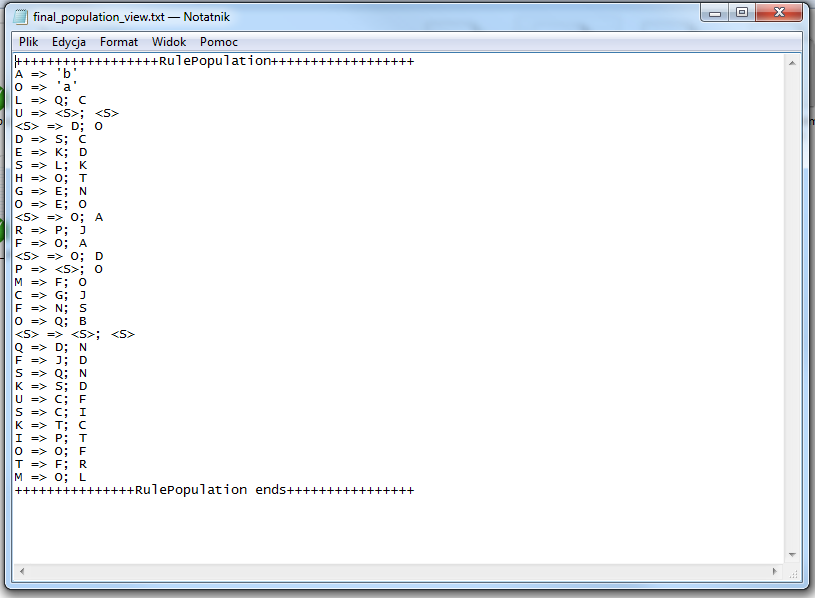
* Całość jest listą;
* Pierwszy element listy to typ populacji reguł (stochastyczna lub nie);
* Pozostałe elementy to wszystkie reguły populacji o nieokreślonej kolejności, na każdą z nich składają się:
  + Nieznormalizowane prawdopodobieństwo (tylko w przypadku populacji stochastycznych);
  + Rodzic, lewe i prawe dziecko – rodzic zawsze jest symbolem nieterminalnym, lewe dziecko może być symbolem terminalnym lub też nie, prawe dziecko jest symbolem nieterminalnym lub wartością null.

Symbol x zostanie zapisany w postaci pojedynczej liczby całkowitej y wg następujących zasad:

* y = -101 – occurence(x) dla symboli terminalnych, gdzie occurence(x) to wartość równa numerowi pierwszego wystąpienia symbolu terminalnego w zbiorze uczącym (np. jeżeli pierwsze zdanie w zbiorze uczącym to „a b a a c”, wówczas symbole a, b, c, otrzymają identyfikatory -101, -102, -103;
* -100 <= y <= 100, jeżeli x to symbol specjalny. W obecnej implementacji biblioteki istnieją tylko dwa symbole specjalne – startowy (y=1) oraz uniwersalny (y=2), reszta może zostać rozbudowana w przyszłości wraz z algorytmem;
* y = 101 + ord(x) jeżeli x to symbol nieterminalny. Przez ord(x) rozumiemy takie odwzorowanie identyfikatora symbolu x na zbiór liczb całkowitych, że A=1, B=2, itd.;

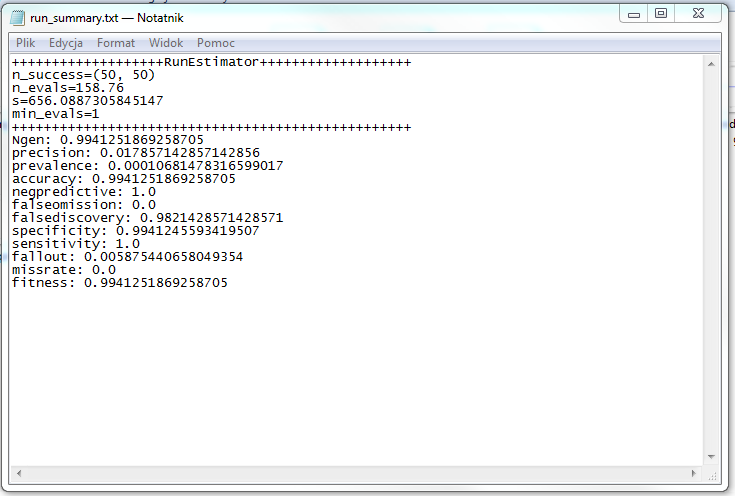


final\_population\_view.txt – Czytelny zapis populacji wykorzystanej w cyklu testowym. Dzięki temu plikowi jest możliwy szybki podgląd populacji wynikowej bez konieczności uruchamiania edytora populacji.



grammar\_estimator.grest – plik .json będący zrzutem obiektu oceny jakości procesu uczenia. Dane te to nic innego jak tekstowa reprezentacja diagramów macierzy omyłek.

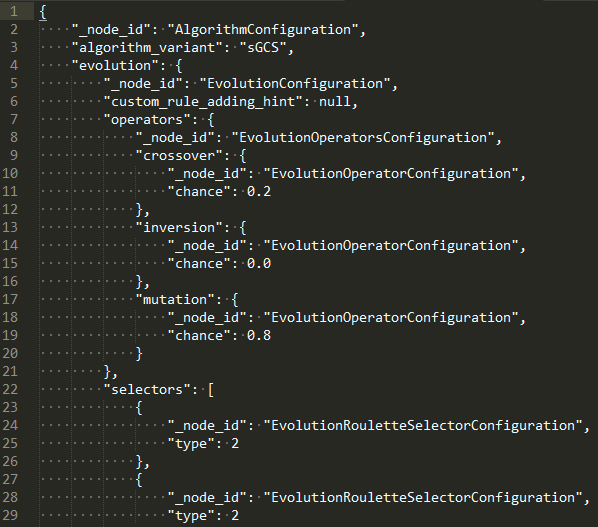
run\_summary.txt – podsumowanie przebiegu algorytmu. Zawiera ilość uruchomień uczących zakończonych sukcesem (n\_success), średnią ilość kroków ewolucyjnych potrzebnych do znalezienia rozwiązania (n\_evals), odchylenie standardowe tejże (s) oraz minimum (min\_evals). Oprócz tego znajduje się tutaj macierz omyłek dla etapu testowego.



\*.inconf – plik danych wejściowych. Zawiera pełną ścieżkę do zbioru testowego oraz uczącego zapisane w formacie json.



\*.parconf – plik konfiguracji. Zawiera wartości wszystkich parametrów algorytmu zapisane w formacie json.



1. Biblioteka

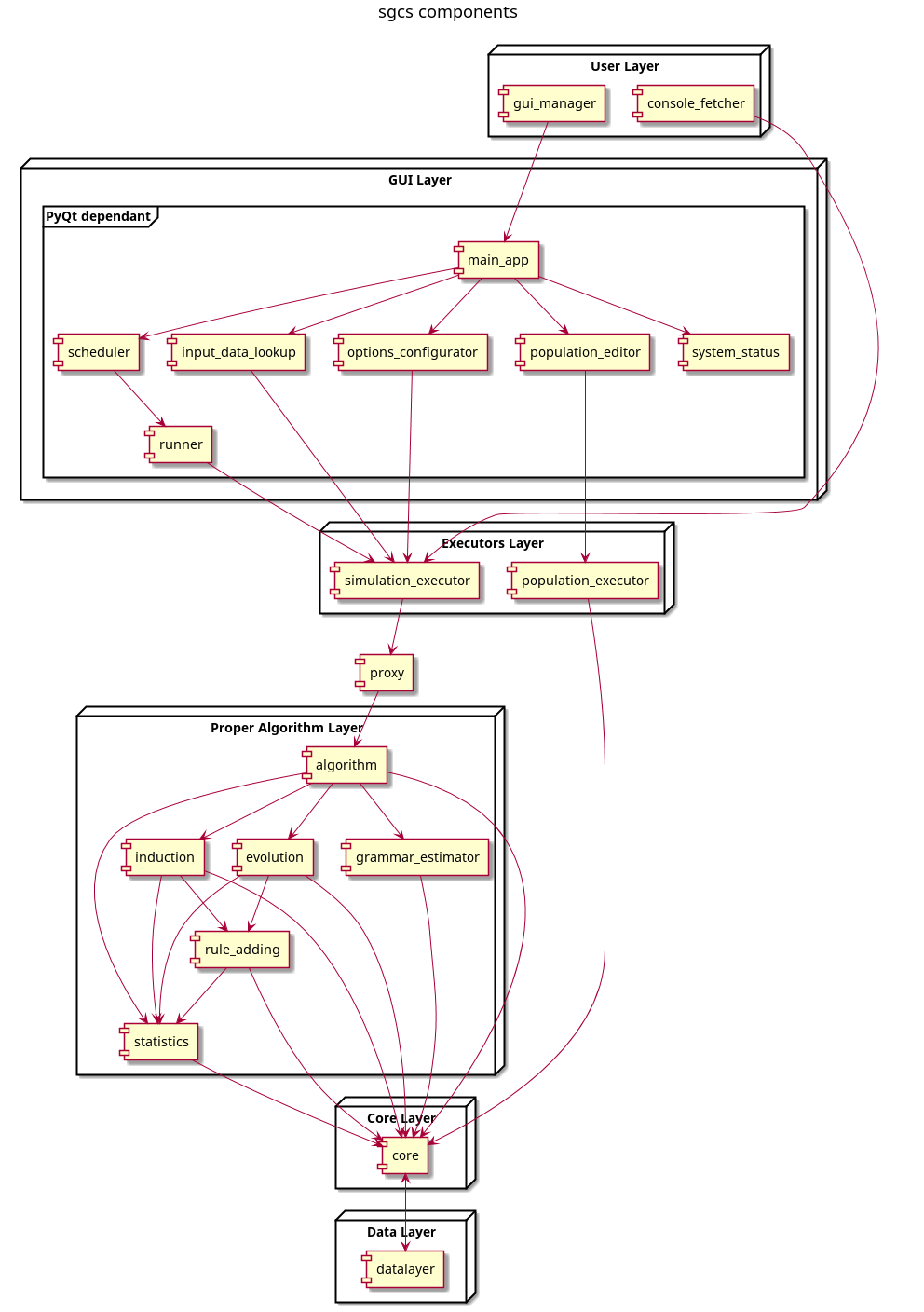
Na najwyższym poziomie projektu znajduje się plik „requirements.txt”. Zawiera on listę wszystkich bibliotek niezbędnych do dziania biblioteki, z wyjątkiem PyQt4, na którym oparto gui i który należy zainstalować osobno, w celu korzystania z elementów interfejsu graficznego.

Instalacji zależności wymienionych w pliku requirements.txt można dokonać następującym poleceniem:

sudo pip install –r requirements.txt

Dodatkowo zależnie od system jedna z wymienionych w tym pliku bibliotek (matplotlib) może wymagać dodatkowych bibliotek, które to zależności należy rozwiązać zgodnie z zaleceniami twórców biblioteki (na przykład na ubuntu polecenie sudo apt-get install python3-matplotlib rozwiązuje wszystkie problemy z zależnościami).

Warstwowa budowa biblioteki

W całej budowie biblioteki można wyróżnić sześć warstw logicznych:

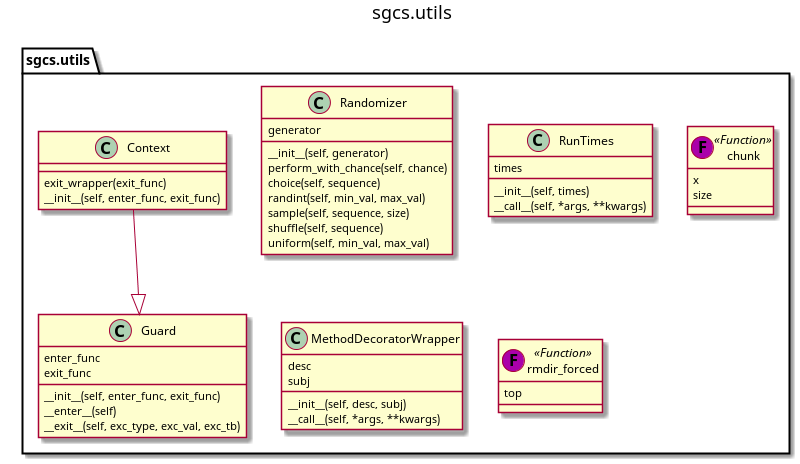
* Warstwa Użytkownika – pierwsza warstwa, na jaką natrafia użytkownik podczas uruchamiania aplikacji. Jest to skrypt inicjujący GUI lub aplikację konsolową;
* Warstwa GUI – warstwa z którą użytkownik aplikacji również ma bezpośrednią styczność (przynajmniej w przypadku aplikacji z interfejsem graficznym, w przypadku aplikacji konsolowej interfejsem użytkownika są parametry wywołania skryptu);
* Warstwa Wykonawców – jest to warstwa realizująca sporą część logiki biznesowej, czyli zbieranie artefaktów, edycję danych wejściowych algorytmu itp.;
* Warstwa Właściwego Algorytmu – warstwa zawierająca „czysty” algorytm GCS/sGCS/neg-sGCS, nieskażona logiką wyższych, ani niższych warstw;
* Warstwa Rdzenia – warstwa odpowiadająca za przechowywanie w pamięci najbardziej podstawowych elementów środowiska algorytmu, ściśle powiązana z warstwą niższą;
* Warstwa Danych – warstwa odpowiadająca za serializację oraz deserializację danych Warstwy Rdzenia.

Dodatkowo mamy jeszcze moduł sgcs.gui.proxy rozszerzający możliwości warstwy algorytmu w celu łatwiejszego wykorzystania go przez GUI. Moduł sgcs.core bywa też używany w niektórych klasach GUI, co trochę przełamuje zaprezentowany diagram zależności, ale chroni przed wprowadzaniem niepotrzebnych warstw abstrakcji. Oprócz tego w bibliotece występują moduły pomocnicze wykorzystywane praktycznie w każdym innym module – z tego powodu nie zostały naniesione na diagram.

Moduły pomocnicze

Biblioteka posiada zbiór kilku klas oraz funkcji wykorzystywanych praktycznie w całej aplikacji. Zostały one zebrane w jeden wygodny w wykorzystaniu moduł – sgcs.utils. Praktycznie każdy moduł wykorzystuje sporą część zawartych w nim funkcjonalności.

sgcs.utils



Randomizer

Klasa zapewniająca warstwę abstrakcji pomiędzy całą biblioteką a standardową klasą biblioteczną random.Random. Dzięki temu jest możliwe wykorzystanie innej klasy zapewniającej dostęp do lepszej maszyny losowej (na przykład klasę wykorzystującą specyficzną architekturę maszyny, na której uruchamiamy naszą aplikację) albo łatwe uruchomienie całej aplikacji z dokładnie tym samym seedem, co może być niezwykle przydatne w przypadku rozwijania oraz debugowania algorytmu w znacznym stopniu opartego w końcu na losowości.

generator – pole przechowujący prawdziwy obiekt generatora (np. instancję klasy random.Random)

\_\_init\_\_(self, generator)\_\_ - konstruktor pobierający instancję obiektu prawdziwego generatora, dla którego klasa ma być fasadą.

perform\_with\_chance(self, chance) – metoda zwraca True, jeżeli liczba wylosowana z przedziału [0,1] <= chance, False w pp.

choice(self, sequence) – metoda zwracająca losowy obiekt z sekwencji sequence (wszystkie mają równą szansę na bycie wylosowanym).

randint(self, min\_val, max\_val) – zwraca losową liczbę całkowitą z przedziału [min\_val, max\_val].

sample(self, sequence, size) – zwraca size obiektów z sekwencji sequence bez powtórzeń.

shuffle(self, sentence) – zwraca kopię sekwencji sequence, nadając jej losowy porządek.

uniform(self, min\_val, max\_val) – zwraca losową rzeczywistą liczbę zmiennoprzecinkową z przedziału [min\_val, max\_val].

RunTimes

Klasa udostępniająca cukier syntaktyczny na wywołania typu:

run\_one\_more\_time = RunTimes(1)

while remainder > self.ALPHABET\_SIZE or run\_one\_more\_time():

chunk(x, size)

Funkcja dzieląca x na listy długości size, a następnie zwracająca taką listę list. Jeżeli x % size != 0 wówczas ostatni chunk zostanie dopełniony elementami z początku x.

Guard

Cukier syntaktyczny umożliwiający łatwe tworzenie zasobów wykorzystywanych w popularnej w języku Python bloku kontekstu with/as.

enter\_func – delegat wywołany przy wejściu w blok with/as.

exit\_func – delegat wywoływany przy opuszczeniu bloku with/as.

Context

Klasa dziedzicząca po klasie Guard, zapewniającą nieco cukru syntaktycznego na funkcję wyjścia (o ile nie ma potrzeby korzystania z zaawansowanych właściwości bloku with/as, i potrzebny jest prosty strażnik, który po prostu wywoła funkcję na początku i końcu bloku, wówczas powinno korzystać się z tej klasy zamiast klasy Guard.

MethodDecoratorWrapper

Wrapper przydatny przy tworzeniu dekoratorów metod potrzebujących dostępu do instancji obiektu, którego metodę dekorują.

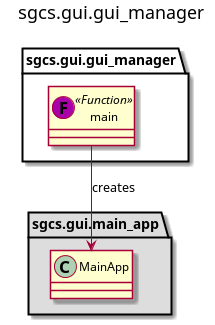
rmdir\_forced(top)

Funkcja usuwająca rekurencyjnie cały folder wraz z jego zawartością.

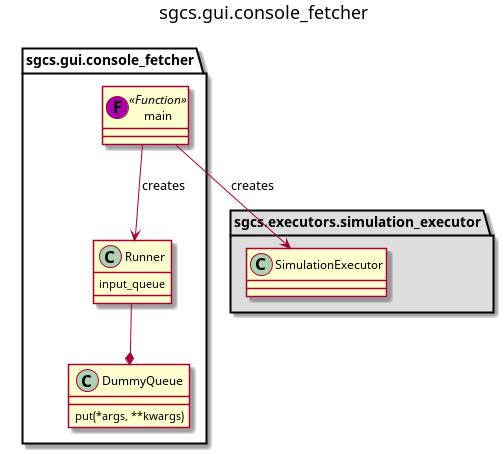
GUI

Ponad całą biblioteką znajduje się dwa proste interfejsy umożliwiające użytkownikowi przetestowanie możliwości biblioteki zaraz po dostarczeniu i wykorzystanie w pełni jej możliwości. Jest to wspominany już w tej pracy konsolowy skrypt console\_fetcher.py oraz utworzone w PyQt GUI uruchamiane przy pomocy skryptu gui\_manager.py. Poniżej znajdują się diagramy zależności obu skryptów.

sgcs.gui.gui\_manager

Skrypt gui\_manager.py powoduje inicjalizację środowiska PyQt, a następnie uruchomienie widgetu MainApp, odpowiadającego za obsługę opisanego wcześniej menu głównego aplikacji. Następuje oddelegowanie kontroli do MainApp, powraca ona do gui\_manager.py dopiero podczas czyszczenia środowiska PyQt podczas finalizowania działania aplikacji.

sgcs.gui.console\_fetcher.py

W przypadku skryptu console\_fetcher.py postawiono na minimum funkcjonalności, jego uruchomienie powoduje natychmiastowe uruchomienie algorytmu. Z tego też powodu jest potrzeba istnienia uproszczonych klas Runner oraz DummyQueue.

Runner

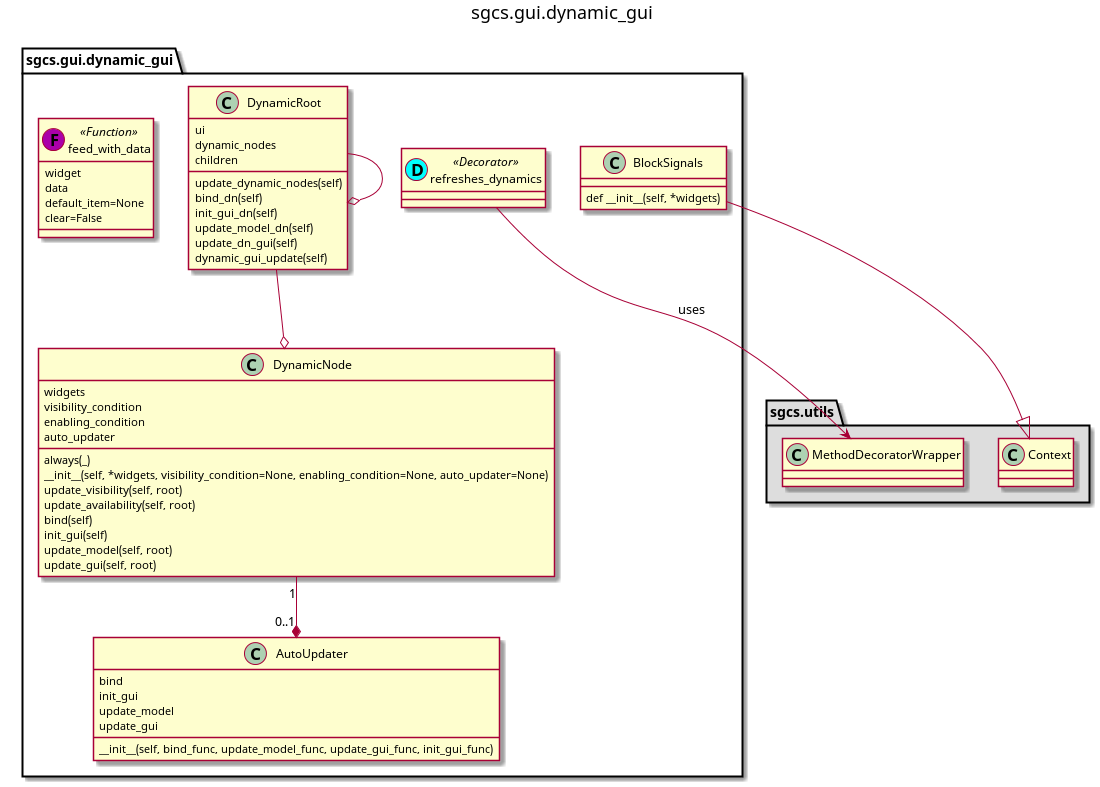
Klasa ta służy do przechowywania oraz ustalenia kolejki zadań, które mają być wykonane przez naszą bibliotekę. W przypadku console\_fetchera nie mamy do czynienia z interaktywną konfiguracją listy zadań (która to jest generowana automatycznie na podstawie podanej listy argumentów wywołania skryptu), toteż klasa ta jest wydmuszką i odpowiedzialność na wygenerowaniu listy zadań spoczywa na funkcji main. Oprócz tego zawiera ona pole input\_queue, również wydmuszki, która w przypadku interaktywnego gui jest wykorzystywana przez podprocesy algorytmu do zwracania szczegółowych danych diagnostycznych, niewykorzystywanych w przypadku uruchomienia aplikacji w trybie chmury.

DummyQueue

Klasa ta jest wydmuszką klasy multiprocessing.Queue, umożliwiającej asynchroniczną wymianę informacji pomiędzy procesami aplikacji. W tej aplikacji można rozróżnić dwa systemy przesyłania danych z procesów – jako wartość zwracana poleceń imap oraz właśnie przy pomocy tej kolejki. Ponieważ jednak przy pomocy kolejki wysyłamy jedynie informacje diagnostyczne, które mają większą rację bytu przy zastosowaniu gui (nieobecnego w przypadku uruchomienia aplikacji w trybie chmury), więc DummyQueue implementuje wyłącznie pustą metodę put (wszystkie informacje diagnostyczne są ignorowane).

sgcs.gui.dynamic\_gui

Moduł sgcs.gui.dynamic\_gui pełni w zasadzie tą samą rolę dla wszystkich powiązanych z gui modułów bilbioteki co utils dla wszystkich modułów biblioteki – jego zadaniem jest zgromadzenie całego wspólnego kodu dla wszystkich modułów GUI. Większość klas modułu skupia się według idei dynamicznego gui – umożliwiają one tworzenie prostego wiązania danych, co umożliwia programowanie z wykorzystaniem wzorca projektowego Model-View-Viewmodel, domyślnie niewspieranego przez bibliotekę PyQt (istnieje wsparcie dla tegoż wzorca w przypadku kilku bardziej skomplikowanych widgetów, jednakże oddzielenie logiki od implementacji zgodnie z zalecaniami MVVM nie jest możliwe w przypadku prostszych widgetów bez zastosowania dodatkowego kodu.

feed\_with\_data(widget, data, default\_item=None, clear=False)

Funkcja ta spowoduje wypełnienie widgetu dysponującego metodami clear(self), addItems(self, items) oraz setCurrentIndex(self, index) danymi data. Można wykorzystać tą metodę na przykład w celu utworzenia comboboxa na podstawie listy napisów. Pole default\_item umozliwia sprecyzowanie domyślnie zaznaczonej opcji. Podanie opcjonalnego argumentu clear z wartością False spowoduje wyczyszczenie dotychczasowej zawartości widgetu. Funkcja ta zakłada przechowywanie przez widget danych w porządku i może nie działać prawidłowo, jeżeli widget nie zapewnia kolejności pozycji.

refreshes\_dynamics

Dekorator, którym można udekorować metodę klasy dziedziczącej po klasie DynamicRoot. Funkcja w ten sposób udekorowana będzie zawsze pod koniec swojego działania wołać metodę DynamicRoot.update\_dynamic\_nodes(self) w celu odświeżenia dynamicznych elementów gui.

BlockSignals

Obiekt kontekstu (można go wykorzystać w bloku with/as), który powoduje zablokowanie wysyłania sygnałów PyQt przez podane widgety (przydatne przy modyfikowaniu wartości, które mogłyby zostać potraktowane jako akcja użytkownika w przypadku gdy nie jest to pożądane zachowanie. BlockSignals jest bezpieczne ze względu na wielokrotne wywołanie (tj. jeżeli wewnątrz bloku A BlockSignals znajdzie się blok BlockSignals B, oba blokujące ten sam widget x, wówczas x zostanie prawidłowo odblokowany pod koniec najbardziej zewnętrznego bloku A, a nie po opuszczeniu bloku B).

\_\_init\_\_(self, \*widgets) – Konstruktor, wszystkie widgets zostaną zablokowane.

DynamicRoot

Klasa realizująca bindowanie widoków View do obiektów ViewModel.

ui – pole przechowujące obiekt View.

dynamic\_nodes – pole przechowujące listę tzw. dynamicznych węzłów, umożliwiających sprecyzowanie logiki interfejsu bez konieczności modyfikowania jego samego.

children – jest możliwe podczepianie jednego obiektu typu DynamicRoot do drugiego jako dziecko w celu utworzenia struktury drzewiastej. Dzięki temu żądając odświeżenia gui lub modelu korzenia takiej struktury w efekcie nastąpi odświeżenie jej w całości. Ten mechanizm działa tylko w przypadku struktur drzewiastych i nie jest zabezpieczony przed wystąpieniem cyklu!

update\_dynamic\_nodes(self) – powoduje odświeżenie widoczności oraz dostępności gui na podstawie wszystkich dynamicznych węzłów.

bind\_dn(self) – powoduje podpięcie logiki wystawionej przez metody bind(self) wszystkich dynamicznych węzłów.

init\_gui\_dn(self) – powoduje inicjację GUI poprzez wywołanie wszystkich metod init\_gui(self) dynamicznych węzłów.

update\_model\_dn(self) – powoduje odświeżenie modelu na podstawie obecnego stanu gui (czyli dokładnie wywołanie metod update\_model() wszystkich dynamicznych węzłów).

update\_dn\_gui(self) – powoduje odświeżenie gui na podstawie obecnego stanu modelu, blokując przy tym wszystkie sygnały PyQt na czas procesu.

dynamic\_gui\_update(self) – powoduje odświeżenie widoczności, dostępności oraz stanu GUI na podstawie obecnego modelu.

DynamicNode

Klasa umożliwiająca zdefiniowanie części lub całości logiki obsługi widgetu. Może ona przejąć kontrolę przede wszystkim nad widocznością widgetu, jego dostępnością oraz powiązać go z konkretnym viewmodelem. Jeden obiekt DynamicNode reprezentuje jakiś fragment logiki, w efekcie są one powiązane z widgetami relacją wiele do wielu – jeden dynamiczny węzeł może opisywać zachowanie wielu widgetów, ale równocześnie kilka innych węzłów może skupiać się na pozostałych aspektach jednego z tych widgetów itd.

widgets – pole precyzujące widgety, których widoczność i dostępność będzie kontrolowana przez ten obiekt.

visibility\_condition – delegat określający warunki widoczności widgetów.

enabling\_condition – delegat określający warunki dostępności widgetów.

auto\_updater – instancja obiektu AutoUpdater, zajmującego się powiązaniem modelu z widokiem.

always(\_) - pomocniczy delegat zwracający zawsze True.

\_\_init\_\_(self, \*widgets, visibility\_condition=None, enabling\_condition=None, auto\_updater=None) – konstruktor. Jeżeli nie poda się któregoś z parametrów opcjonalnych, wówczas ten aspekt zostanie zignorowany przez ten konkretny węzeł (na przykład niesprecyzowanie delegata visibility\_condition sprawi, że węzeł ten nie będzie zarządzać widocznością widgetów).

AutoUpdater

Klasa grupująca funkcje bind, init\_gui, update\_model oraz update\_gui w jednym miejscu.

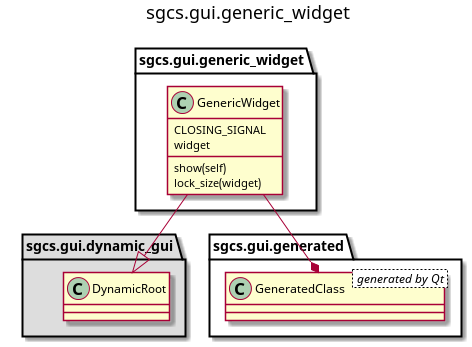
bind – delegat zawierający logikę, która powinna zostać wykonana na późnym etapie tworzenia gui (np. rejestracja na sygnały).

init\_gui – delegat zawierający logikę inicjującą GUI przy tworzeniu interfejsu po raz pierwszy.

update\_model – delegat odświeżający model na podstawie obecnego stanu GUI.

update\_gui – delegat odświeżający GUI na podstawie obecnego stanu modelu.

sgcs.gui.generic\_widget



Klasa GenericWidget stanowi rozszerzenie klasy DynamicRoot o kilka przydatnych mechanizmów. Po pierwsze stanowi pomost pomiędzy klasami wygenerowanymi przy pomocy QtDesignera a zdefiniowanymi przez użytkownika widokami. Po drugie nadpisuje obsługę zamknięcia okna w taki sposób, żeby został wysłany sygnał informujący o tym wydarzeniu (brakujące zachowanie w obecnej wersji PyQt). Po trzecie umożliwia łatwą blokadę rozmiarów okna (kolejna czynność, której nie da się wykonać w PyQt za pomocą jednej metody).

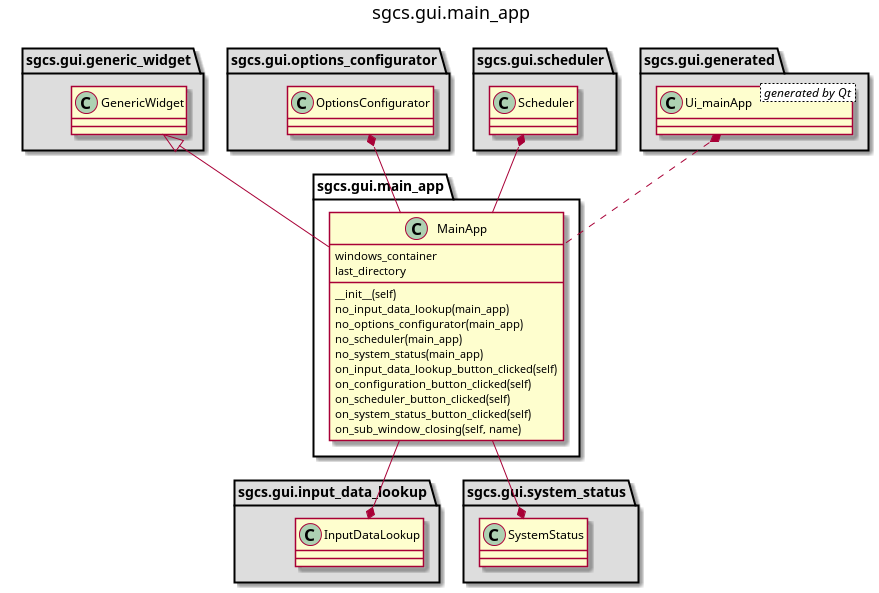
CLOSING\_SIGNAL – identyfikator sygnału wysyłanego podczas zamykania okna. Jedyny argument sygnału to nazwa klasy widgeta, który jest właśnie zamykany.

widget – obiekt QtGui.QtWidget.

show(self) – metoda umożliwiająca pokazanie widgetu

lock\_size(widget) – statyczna metoda umożliwiająca zablokowanie możliwości zmiany rozmiarów okna.

sgcs\_main\_app



Klasa odpowiadająca za wyświetlenie i obsługę menu głównego. Wszystkie otwierane przez nią okna są singletonami – w danej chwili może być otwarte wyłącznie jedno okno danego typu, nie możemy na przykład uruchomić dwóch okien statusu systemu równocześnie. Jest to związane z wykorzystaniem specyficznych zasobów systemowych przez większość okien, co sprawia, że utworzenie ich kilku instancji mogłoby prowadzić do dziwnych problemów (i tylko w kilku przypadkach miałoby jakiś sens).

windows\_container – fabryka okien zajmująca się kontrolą ich czasu życia

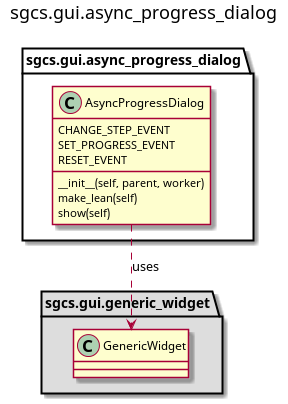
last directory – zmienna przechowująca ostatni otwarty w jakiejkolwiek części aplikacji folder. Ułatwia to powrót do ostatnio używanych danych (nie musimy za każdym razem szukać tego samego folderu).

no\_input\_data\_lookup/no\_options\_configurator/no\_scheduler/no\_system\_status – funkcje sprawdzające czy dane okno zostało już stworzone.

on\_input\_data\_lookup\_button\_clicked/on\_configuration\_button\_clicked/on\_scheduler\_button\_clicked/on\_system\_status\_button\_clicked – funkcje obsługujące otwieranie okien poszczególnych aplikacji.

on\_sub\_window\_closing – funkcja obsługująca proces zamknięcia okna aplikacji, nasłuchuje na GenericWidget.CLOSING\_SIGNAL.

sgcs.gui.async\_progress\_dialog



Klasa umożliwiająca wyświetlenie maksymalnie odchudzonego dialogu z komunikatem oraz paskiem postępu. Po wyświetleniu go można sterować jego zachowaniem przy pomocy odpowiednich sygnałów.

CHANGE\_STEP\_EVENT – identyfikator sygnału wysyłanego w celu zmiany tekstu obecnego komunikatu.

SET\_PROGRESS\_EVENT – identyfikator sygnału wysyłanego w celu zmiany wartości obecnego postępu.

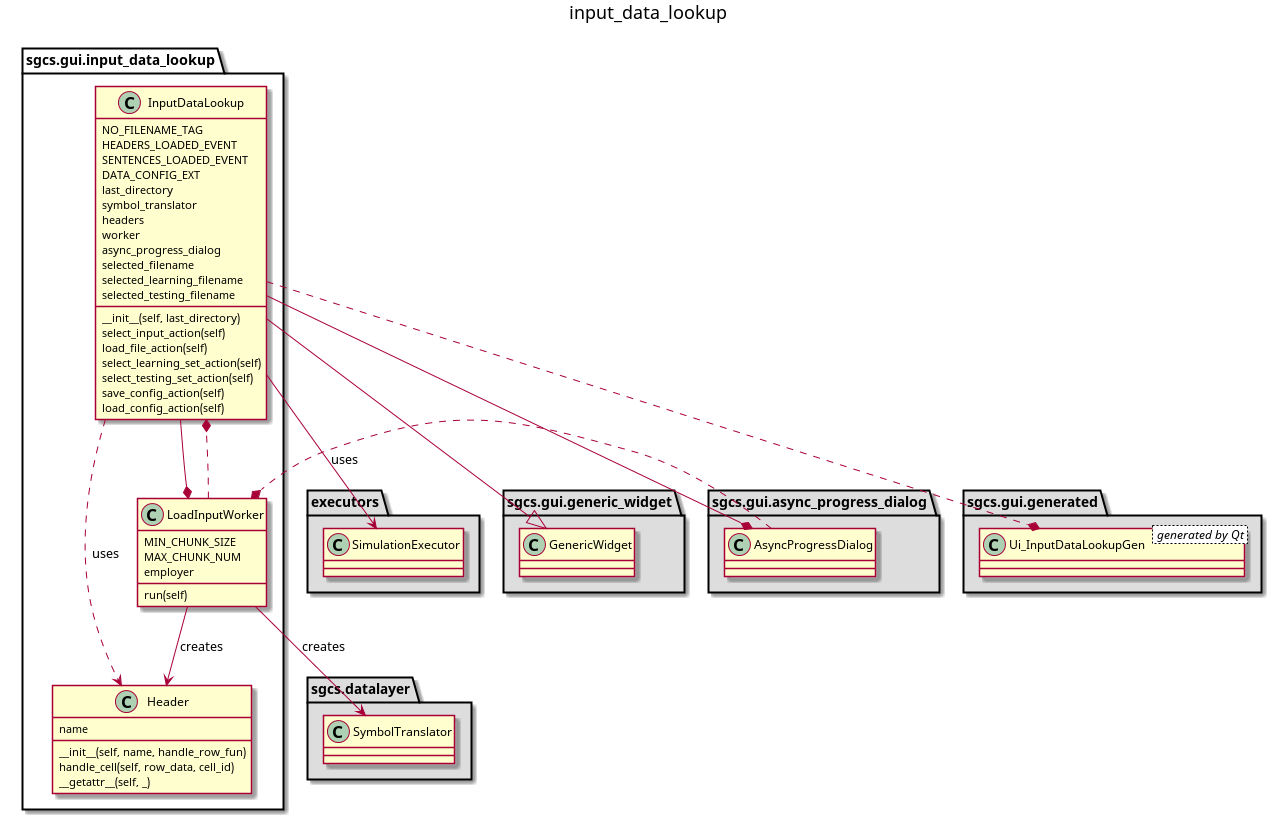
RESET\_EVENT – identyfikator sygnału powodującego schowanie/ponowne wyświetlenie okna dialogowego.

\_\_init\_\_(self, parent, worker) – jako parent należy podać okno – rodzica komunikatu. Worker jest obiektem, który będzie emitować sygnały (na przykład osobny wątek).

make\_lean(self) - „odchudza” komunikat.

show(self) – powoduje wyświetlenie komunikatu.

sgcs.gui.input\_data\_lookup



Moduł ten zawiera klasy niezbędne do obsługi podglądu plików z danymi wejściowymi do gramatyk oraz tworzenie plików konfiguracyjnych danych wejściowych \*inconf.

InputDataLookup

Główna klasa odpowiedzialna za obsługę menu „Prepare input data”.

NO\_FILENAME\_TAG – napis widoczny do czasu wyboru pliku.

HEADERS\_LOADED\_EVENT – identyfikator sygnału wysyłanego, gdy już znamy wszystkie nagłówki kolumn tabeli.

SENTENCES\_LOADED\_EVENT – identyfikator sygnału wysyłanego, gdy tabela została załadowana i jest już gotowa do użytku.

DATA\_CONFIG\_EXT – stała przechowująca rozszerzenie plików konfiguracyjnych danych wejściowych („.inconf”).

last\_directory – ostatni otwarty folder.

symbol\_translator – obiekt wykorzystywany do wczytania danych z dysku (obiekt typu SymbolTranslator).

headers – nagłówki tabeli.

worker – uchwyt wątku, któremu zleca się wczytanie pliku.

async\_progress\_dialog – uchwyt okna z postępem wczytywania wyświetlanego w trakcie ładowania danych wejściowych.

selected\_filename – wybrany plik danych wejściowych.

selected\_learning\_filename – ścieżka do wybranego zestawu uczącego.

selected\_testing\_filename – ścieżka do wybranego zestawu testowego.

\*\_action(self) – funkcje obsługujące naciśnięcie odpowiedniego przycisku.

LoadInputWorker

Wątek zajmujący się wczytaniem danych do tabeli. Wbrew pozorom nie jest to zadanie aż tak trywialne, jak mogłoby się wydawać – mimo wykonywania najbardziej czasochłonnej operacji, czyli ładowania danych z dysku do pamięci w osobnym wątku to w przypadku dużych danych wejściowych samo dodawanie danych do widgetu tabeli może spowodować zawieszenie się głównego okna. Dlatego też worker ten dokonuje podziału danych wejściowych na chunki odpowiedniego rozmiaru, po czym wczytuje je jeden po drugim. Chunk ma rozmiar co najmniej równy MIN\_CHUNK\_SIZE i równocześnie taki, żeby ilość chunków nie przekroczyła MAX\_CHUNK\_NUM. Worker na czas swojej pracy powoduje również wyświetlenie komunikatu ze statusem prac.

MIN\_CHUNK\_SIZE – minimalny rozmiar chunka.

MAX\_CHUNK\_NUM – maksymalna ilość chunków, jakie mogą powstać.

employer – uchwyt zleceniodawcy zadania, czyli InputDataLookup.

run(self) – główna metoda workera, w celu wykonania asynchronicznego nie należy wołać jej bezpośrednio, lecz zamiast tego QtCore.Qthread.start.

Header

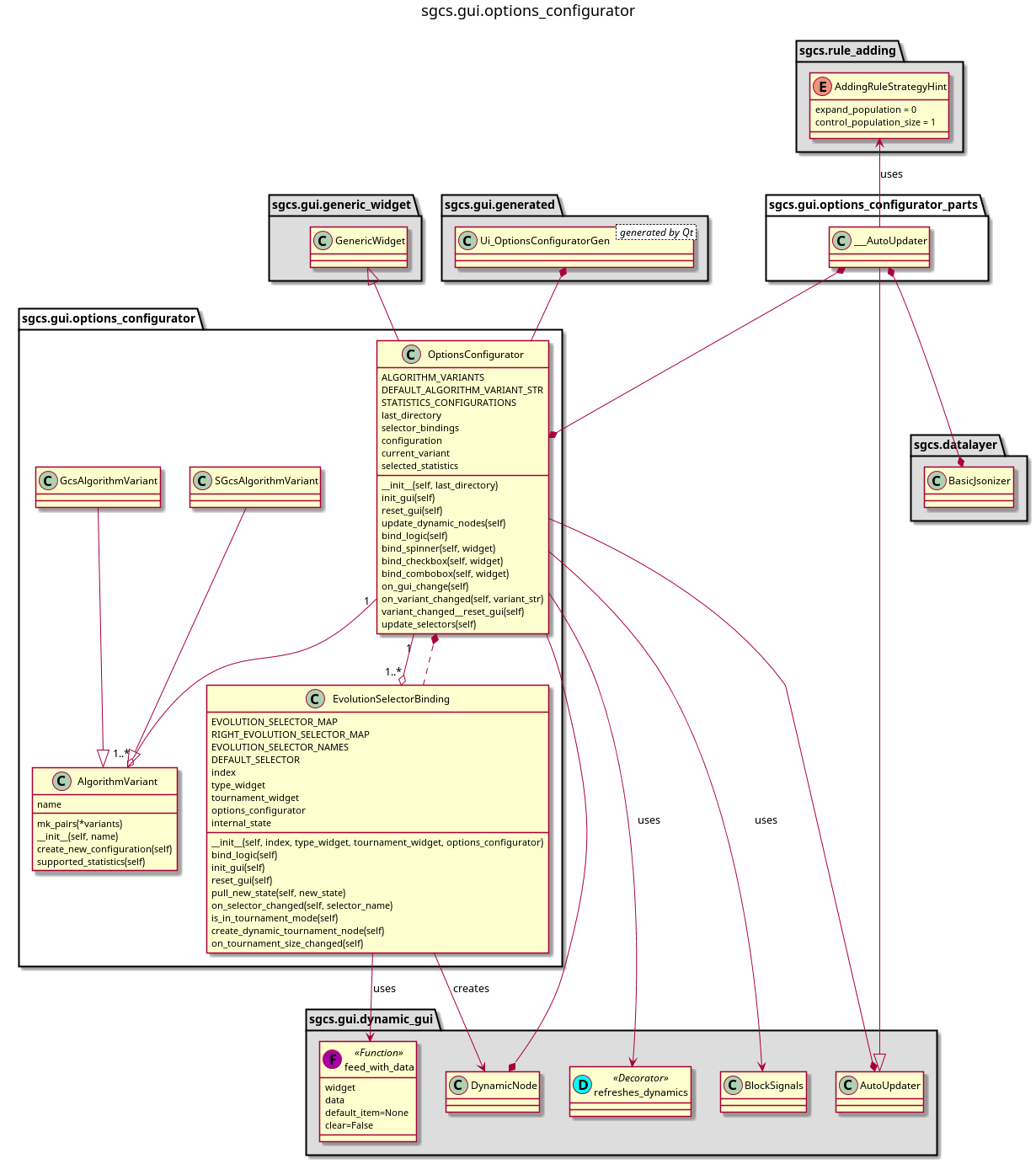
Klasa zajmująca się obsługą prawidłowego wyświetlania zawartości danej kolumny tabeli.

name – nagłówek danej kolumny.

\_\_init\_\_(self, name, handle\_row\_fun) - konstruktor, należy przekazać mu nagłówek kolumny name oraz delegata zdolnego do odpowiedniego wyświetlenia zawartości danej kolumny.

handle\_cell(self, row\_data, cell\_id) – funkcja zajmująca się prawidłowym wypełnieniem danej komórki tabeli.

sgcs.options\_configurator i sgcs.options\_configurator\_parts

Widok „Prepare configuration” jest najbardziej skomplikowaną częścią GUI, co widać, chociażby po jego diagramie klas. Diagram ten byłby jeszcze bardziej rozbudowany, gdyby nie zwijać wszystkich składowych obiektów AutoUpdater z przestrzeni nazw sgcs.gui.options\_configurator\_parts w jedną klasę \_\_\_AutoUpdater. Jest to spowodowane niezwykłym rozbudowaniem tego menu (ma za zadanie obsłużyć dobór około 30 parametrów algorytmu), dynamiczną budową (nie wszystkie pola są zawsze dostępne, istnieją zależności pomiędzy niektórymi parametrami) oraz dosyć wczesnym napisaniem go, kiedy jeszcze struktura GUI nie była aż tak uporządkowana.

AlgorithmVariant

Abstrakcyjna klasa stosowana do rozróżnienia pomiędzy dostępnymi algorytmami na poziomie GUI. Obecnie posiada dwie konkretne implementacje: *GcsAlgorithmVariant* oraz *SgcsAlgorithmVariant*.

name – wyświetlana nazwa algorytmu

mk\_pairs(\*variants) – statyczna metoda tworząca mapę variant.name: variant.

create\_new\_configuration – abstrakcyjna metoda, której wywołanie powinno spowodować utworzenie domyślnego obiektu konfiguracji dla danego wariantu algorytmu.

supported\_statistics – lista nazw funkcji fitness, które są wspierane przez dany algorytm. Od doboru funkcji fitness (wykorzystującej lub nie zdania negatywne) zależy również czy proces uczenia będzie przebiegał również na zdaniach negatywnych.

\_\_\_AutUpdater

Grupa klas dziedziczących po klasie AutoUpdater umiejscowionych w przestrzeni nazw sgcs.gui.options\_configurator\_parts, jak sama nazwa przestrzeni wskazuje, są klasami przechowującymi fragmenty logiki menu „Prerare Configuration”. W przestrzeni tej możemy odnaleźć następujące klasy:

* ClassicalStatisticsAutoUpdater – klasa obsługująca część menu odpowiedzialną za klasyczną funkcję płodności;
* CrowdingAutoUpdater – klasa obsługująca część menu odpowiedzialną za funkcjonowanie ścisku;
* ElitismAutoUpdater – klasa obsługująca część menu odpowiedzialną za funkcjonowanie elityzmu;
* EvolutionAutoUpdater – klasa obsługująca część menu odpowiedzialną za parametry operatorów ewolucyjnych;
* InductionAutoUpdater – klasa obsługująca część menu odpowiedzialną za parametry operatorów pokrycia oraz korekcję gramatyki;
* MainBoxAutoUpdater – klasa obsługująca część menu odpowiedzialną za wczytywanie i zapisywanie konfiguracji oraz wybór wariantu algorytmu;
* RootAutoUpdater – klasa obsługująca część menu odpowiedzialną za ogólne parametry algorytmu (ilość uruchomień cyklu uczacego, satysfakcjonujący poziom przystosowania, rodzaj funkcji przystosowania);
* RulesAutoUpdater – klasa obsługująca część menu odpowiedzialną za ogólny stan populacji reguł (maksymalny rozmiar populacji, ilość symboli nieterminalnych itd.).

Każda z tych klas pilnuje, aby parametry, za które jest odpowiedzialna, były zgodne z tym, co jest widoczne w danym momencie w menu. Dbają również o widoczność i dostępność części menu (na przykład chowając część dotyczącą parametryzacji klasycznej funkcji fitness, jeżeli zdecydujemy się na inną niż klasyczna funkcję fitness).

EvolutionSelectorBinding

Jest to klasa zapewniająca obsługę części menu związanej z doborem selektorów algorytmu genetycznego. Jest to jeden z protoplastów architektury DynamicRoot, który nie został ze względu na swoją skomplikowaną logikę oraz brak czasu w pełni przetłumaczony na nowy mechanizm. Zadaniem tej klasy jest obsługa wyboru selektorów algorytmu genetycznego oraz udostępnienie pól dodatkowych parametrów, jeżeli dany selektor takie parametry posiada.

EVOLUTION\_SELECTOR\_MAP – mapa odwzorowująca nazwę na węzeł konfiguracyjny selektora.

RIGHT\_EVOLUTION\_SELECTOR\_MAP – mapa odwzorowująca węzeł konfiguracyjny węzeł selektora na nazwę.

EVOLUTION\_SELECTOR\_NAMES – lista kluczy EVOLUTION\_SELECTOR\_MAP.

DEFAULT\_SELECTOR – nazwa selektora domyślnego.

index – nr selektora, za który odpowiada instancja.

type\_widget – uchwyt do pola combobox odpowiedzialnego za wybór typu selekcji.

tournament\_widget – uchwyt do pola spinner odpowiedzialnego za określenie rozmiaru turnieju.

options\_configurator – uchwyt do głównej klasy okna konfiguracji.

internal\_state – przechowywana węzeł konfiguracyjny selektora.

bind\_logic(self) – metoda odpowiedzialna za rejestrację funkcji obsługujących wydarzenia oraz późną inicjalizację niektórych wartości pól.

init\_gui(self) – metoda odpowiedzialna za inicjalizację menu (dostępne pola oraz zakres wartości).

reset\_gui(self) – funkcja odświeżania GUI.

pull\_new\_state(self, new\_state) – metoda obsługująca zmianę selektora z równoczesnym odświeżeniem menu (tj. zmianę stanu selektora na przykład poprzez wczytanie istniejącej konfiguracji).

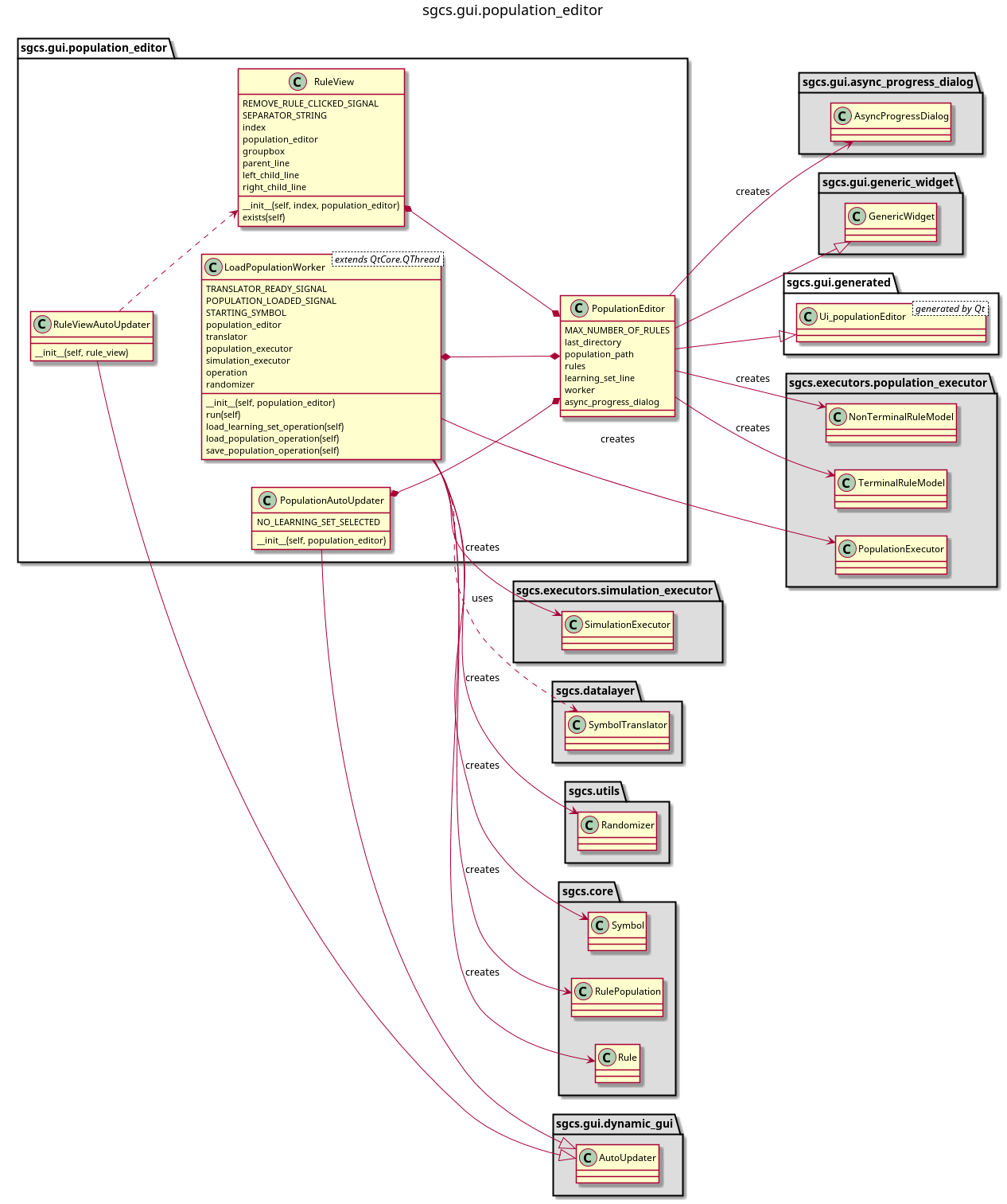
on\_selector\_change(self) – metoda obsługująca zmianę selektora przez bezpośrednią interakcję użytkownika z kontrolką (nie poprzez wczytanie istniejącej konfiguracji).

is\_in\_tournament\_mode(self) – metoda zwracająca True, jeżeli obecną selekcją jest selekcja turniejowa, False w przeciwnym przypadku.

create\_dynamic\_tournament\_node(self) – metoda tworząca dynamiczny węzeł odpowiedzialny za widoczność kontrolki rozmiaru turnieju.

on\_tournament\_size\_changed(self) – metoda obsługująca zmianę rozmiaru turnieju.

sgcs.gui.population\_editor



Okno „Edit population jest tworzone i sterowane przy pomocy klas z modułu sgcs.gui.population\_editor.

RuleViewAutoUpdater

Klasa ta zapewnia synchronizację danych oraz GUI pojedynczej reguły.

PopulationAutoUpdater

Klasa ta zapewnia synchronizację ścieżki do zbioru uczącego z odpowiednim widgetem.

NO\_LEARNING\_SET\_SELECTED – napis wykorzystywany w kontrolce w przypadku braku ścieżki.

RuleView

Klasa ta zarządza widokiem oraz edycją reguły. Informuje również klasę RuleEditor o potrzebie usunięcia reguły.

REMOVE\_RULE\_CLICKED\_SIGNAL – identyfikator sygnału usunięcia reguły.

SEPARATOR\_STRING – napis rozdzielający lewą oraz prawą stronę produkcji.

index – identyfikator reguły.

population\_editor – uchwyt obiektu PopulationEditor.

groupbox – groupbox, w którym klasa umieszcza wszystkie swoje elementy GUI (musi być widoczna na zewnątrz w celu podłączenia do GUI PopulationEditor).

parent\_line, left\_child\_line, right\_child\_line – uchwyty linii tekstu, które składają się na regułę.

exists(self) – Zwraca True, jeżeli istnieje reguła odpowiadająca temu obiektu (tj. w tabieli reguł na pozycji index coś jest).

LoadPopulationWorker

Zadaniem tej klasy jest odciążenie głównego wątku z kilku czasochłonnych procesów. Każdemu uruchomieniu takiego procesu będzie towarzyszyć wyświetlenie odpowiedniego okna, apelującego do użytkownika o cierpliwość. Obecnie Worker może w danej chwili realizować jedno z trzech zadań:

* Wczytywanie zbioru uczącego;
* Wczytywanie pliku populacji;
* Zapisywanie populacji do pliku.

TRANSLATOR\_READY\_SIGNAL – identyfikator sygnału wysyłanego zaraz po wczytaniu pliku ze zbiorem uczącym.

POPULATION\_LOADED\_SIGNAL – identyfikator sygnału wysyłanego po wczytaniu pliku populacji.

STARTING\_SYMBOL – stała, symbol startowy gramatyki.

population\_editor – uchwyt do instancji PopulationEditor.

translator – zmienna przechowująca obiekt SymbolTranslator. Bez tego obiektu nie jest możliwe tłumaczenie z wewnętrznej reprezentacji symboli na przyjazną człowiekowi oraz odwrotnie. Zmienna zostaje zainicjowana po wczytaniu zbioru uczącego.

population\_executor – zmienna przechowująca instancję obiektu PopulationExecutor, zajmującą się logiką biznesową obsługi populacji.

simulation\_executor – zmienna przechowująca instancję obiektu SimulationExecutor, zajmującego się logiką biznesową uruchamiania algorytmu oraz zbierania artefaktów.

operation – obecnie realizowane przez workera zadanie.

randomizer – instancja randomizera niezbędna dla prawidłowego utworzenia niektórych części systemu (na przykład populacji reguł).

run(self) – główna metoda wątku. W celu asynchronicznego wywołania należy wykorzystać metodę QtCore.Qthread.start(). Przed wywołaniem którejkolwiek z tych dwóch metod należy ustawić odpowiednią wartość pola operation (na przykład load\_learning\_set\_operation w celu załadowania zbioru uczącego).

load\_learning\_set\_operation(self) – operacja wczytująca zbiór uczący. Wczytanie zbioru uczącego powoduje zainicjowanie zmiennej translator odpowiednim instancją klasy SymbolTranslator, co umożliwia (i odblokowuje na poziomie GUI) wczytywanie oraz zapisywanie populacji. Należy pamiętać, że GUI chroni jedynie przed wczytaniem lub zapisaniem populacji w przypadku braku istnienia instancji SymbolTranslator. Nie chroni jednak przed wczytywaniem lub zapisywaniem populacji z innym zestawem uczącym niż ten, przy pomocy którego wygenerowano populację reguł. Nie może przed tym chronić, gdyż jest to często zachowanie pożądane (tworzymy zestaw reguł wspólnych dla kilku gramatyk, po czym zapisujemy je dla każdej z nich). Brak ostrożności może jednak spowodować powstanie źle wygenerowanych zbiorów.

load\_population\_operation – operacja umożliwiająca wczytanie istniejącego pliku populacji. Przed wczytaniem należy upewnić się, że wybrano pożądany zbiór uczący.

save\_population\_operation – operacja umożliwiająca zapisanie obecnej populacji do pliku. Przed zapisem należy upewnić się, że wybrano pożądany zbiór uczący.

PopulationEditor

Klasa ta niejako łączy wszystkie powyższe, zapewniając komunikację pomiędzy nimi, dodatkowo zajmuje się aktualizacją modelu populacji (dodawanie i usuwanie reguł w odpowiedzi na wysyłane przez siebie oraz workera sygnały).

MAX\_NUMBER\_OF\_RULES – stała definiująca jak wiele reguł może znaleźć się w populacji.

last\_directory – pole przechowujące ścieżkę ostatnio otwieranego przez nas folderu, dzięki czemu szukanie plików rozpoczynamy od miejsca, gdzie je ostatnio skończyliśmy.

population\_path – ścieżka do pliku ze zbiorem uczącym.

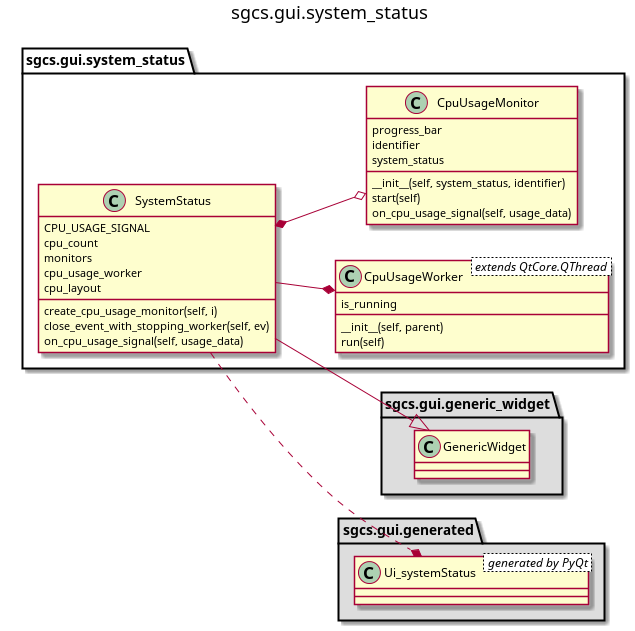
rules – lista obiektów RuleModel, będących reprezentacją reguł na potrzeby tego okna.

learning\_set\_line – linia odpowiadająca za wyświetlanie ścieżki do zbioru uczącego.

worker – instancja klasy LoadPopulationWorker.

async\_progress\_dialog – instancja widgetu asynchronicznego okna postępu (komunikat wyświetlany w trakcie trwania długich operacji).

sgcs.gui.system\_status



Zadaniem tego modułu jest monitorowanie stanu systemu (dokładnie pamięci oraz procesora) i informowanie o jego użyciu przez użytkownika. Nie jest to moduł bezpośrednio związany z samym algorytmem sGCS, jednakże jest niezwykle przydatny. Dzięki niemu można w łatwy sposób śledzić wykorzystanie wszystkich rdzeni oraz ilość zarezerwowanej pamięci. Domyślnie algorytm stara się optymalnie wykorzystywać przydzielone mu zasoby, toteż takie narzędzie jest przydatne w ustalaniu czy system rzeczywiście optymalnie wykorzystuje przydzielone mu zasoby i czy nie da się przyspieszyć całego procesu poprzez modernizację maszyny uruchamiającej aplikację.

CpuUsageWorker

Klasa ta w regularnych odstępach (1s) odpytuje system przy pomocy biblioteki psutil o jego obciążenie procesorów i ilość wolnej pamięci. Następnie wysyła tę informację przy pomocy sygnału SystemStatus.CPU\_USAGE\_SIGNAL.

is\_running – zmienna boolowska umożliwiają prawidłowe zakończenie działania wątku (po zmianie jej wartości na False nie dalej jak sekundę później nastąpi zakończenie działania wątku).

\_\_init\_\_(self, parent) – konstruktor. W tym przypadku pole parent to widget, do którego CpuUsageWorker ma wysyłać sygnały informujące o użyciu procesora (domyślnie widget powiązany z instancją SystemStatus).

run(self) – główna funkcja wątku. Zamiast niej powinno wywoływać się metodę QtCore.Qthread.start w celu asynchronicznego wykonania.

CpuUsageMonitor

Klasa zajmująca się nasłuchiwaniem na sygnały wykorzystania procesora. Jej zadaniem jest obsługa tylko jednego rdzenia.

progress\_bar – widget pasek postępu powiązany z obserwowanym rdzeniem procesora. Przyjmuje wartości od 0% do 100% zużycia.

identifier – Liczba naturalna jednoznacznie identyfikująca obserwowany rdzeń.

start(self) – Wywołanie tej metody powoduje rejestrację klasy na sygnały użycia procesora.

on\_cpu\_usage\_signal(self, usage\_data) – funkcja zajmująca się obsługą przybyłego sygnału zużycia procesora (aktualizuje progress\_bar).

SystemStatus

Klasa tworząca i koordynująca działanie pojedynczej instancji CpuUsageWorker oraz co najmniej jednej instancji CpuUsageMonitor. Oprócz tego zajmuje się obserwowaniem i informowaniem o zużyciu pamięci.

CPU\_USAGE\_SIGNAL – identyfikator sygnału zużycia procesora.

cpu\_counts – zmienna przechowująca całkowitą ilość rdzeni w systemie.

monitors – instancje klasy CpuUsageMonitor (tyle ile wynosi cpu\_counts).

cpu\_usage\_worker – instancja obiektu CpuUsageWorker.

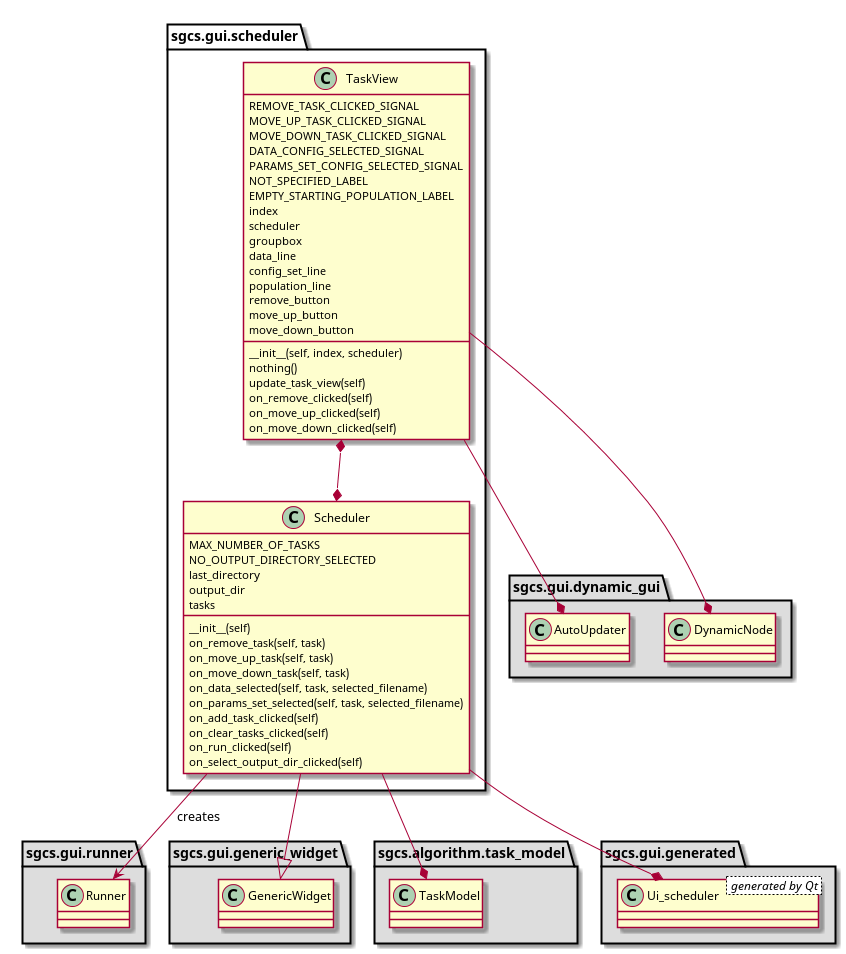
cpu\_layout – część interfejsu, do którego dodają się paski postępu monitorów.

create\_cpu\_usage\_monitor(self, i) – metoda nadzorująca utworzenie oraz rejestrację pojedynczego monitora. Parametr i to numer rdzenia, którego obserwacją ma zająć się monitor.

close\_event\_with\_stopping\_worker(self, ev) – metoda upewniająca się, że w przypadku zamknięcia okna CpuUsageWorker również prawidłowo zakońćzy działanie (wywoływane automatycznie tuż przed oryginalną obsługą wydarzenia zamknięcia).

on\_cpu\_usage\_signal(self, usage\_data) – metoda rejestrowana na sygnały zużycia procesora. Wyciąga z nich informację o zużyciu pamięci i dokonuje odpowiedniej aktualizacji GUI.

sgcs.gui.scheduler



Moduł ten odpowiada za obsługę menu „Scheduler”. Znajduje się tutaj zatem logika umożliwiająca przygotowanie zestawu zadań (tj. odpaleń konkretnego wariantu algorytmu z konkretnymi parametrami dla konkretnych danych wejściowych), zakolejkowanie ich i ustalenie ścieżki folderu na artefakty z przebiegów algorytmu. Odpowiedzialność za uruchomienie i monitoring przebiegu spoczywa na module sgcs.gui.runner, do którego ten moduł wysyła jedynie listę zaplanowanych zadań oraz ścieżkę folderu z artefaktami. Po uruchomieniu algorytmu modyfikacja tych danych nie będzie już możliwa.

TaskView

ViewModel obiektu zadania. Wyświetla jego dane (ścieżkę konfiguracji danych wejściowych \*inconf, konfiguracji parametrów \*.parconf oraz populacji początkowej \*.pop) i zajmuje się ich obsługą.

REMOVE\_TASK\_CLICKED – identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien zostać usunięty.

MOVE\_UP\_TASK\_CLICKED\_SIGNAL – identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien być przesunięty w górę kolejki.

MOVE\_DOWN\_TASK\_CLICKED\_SIGNAL - identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien być przesunięty w dół kolejki.

DATA\_CONFIG\_SELECTED\_SIGNAL - identyfikator sygnału informujący o podaniu nowej ścieżki do danych wejściowych \*.inconf.

PARAMS\_SET\_CONFIG\_SELECTED\_SIGNAL - identyfikator sygnału informujący o podaniu nowej ścieżki do parametrów algorytmu \*.parconf.

NOT\_SPECIFIED\_LABEL – napis wyświetlany w przypadku niesprecyzowanych danych wejściowych lub parametrów algorytmu.

EMPTY\_STARTING\_POPULATION\_LABEL – napis wyświetlany, gdy nie podano ścieżki do populacji początkowej (przyjmowana jest wówczas domyślnie populacja pusta, rozszerzona ewentualnie o losowe reguły początkowe, oczywiście o ile tak stwierdzą parametry algorytmu).

index – indeks powiązanego modelu,

scheduler – uchwyt do obiektu klasy Scheduler.

groupbox – GUI generowane przez viewmodel, wystawione na potrzeby dołączenia do GUI Schedulera.

data\_line – element GUI wyświetlający obecną ścieżkę do pliku \*.inconf.

config\_set\_line – element GUI wyświetlający obecną ścieżkę do pliku \*.parconf.

population\_line – element GUI wyświetlający obecną ścieżkę do pliku \*.pop.

remove\_button – przycisk umożliwiający usunięcie powiązanego modelu.

move\_up\_button – przycisk umożliwiający przesunięcie zadania w górę kolejki.

move\_down\_button – przycisk umożliwiający przesunięcie zadania w dół kolejki.

nothing() - pusty delegat, przydatny w miejscach gdzie prawdziwy delegat nie jest potrzebny.

update\_task\_view(self) – metoda zajmująca się aktualizacją GUI na podstawie modelu.

on\_remove\_clicked(self) – metoda obsługująca naciśnięcie przycisku usunięcia zadania.

on\_move\_up\_clicked(self) – metoda obsługująca naciśnięcie przycisku przesunięcia zadania w górę kolejki.

on\_move\_down\_clicked(self) – metoda obsługująca naciśnięcie przycisku przesunięcia zadania w dół kolejki.

Scheduler

Klasa zarządzająca modelem kolejki zadań. Nasłuchuje na sygnały wysyłane przez obiekty TaskView i dokonuje odpowiednich modyfikacji kolejki.

MAX\_NUMBER\_OF\_TASKS – maksymalna ilość zadań w kolejce. GUI jest dynamicznie generowane i jest w stanie obsłużyć maksymalnie taką liczbę zaplanowanych zadań.

NO\_OUTPUT\_DIRECTORY\_SELECTED – napis wyświetlany zamiast ścieżki folderu na artefakty, jeżeli ten nie został jeszcze sprecyzowany.

last\_directory – zmienna pamiętająca ostatnio otwarty folder.

output\_dir – zmienna pamiętająca ścieżkę folderu na artefakty.

tasks – model kolejki zadań.

on\_remove\_task(self, task) – metoda zarejestrowana na sygnał REMOVE\_TASK\_CLICKED\_SIGNAL, usuwająca zadanie task z kolejki.

on\_move\_up\_task(self, task) – metoda zarejestrowana na sygnał MOVE\_UP\_TASK\_CLICKED\_SIGNAL, przenosząca zadanie task w górę kolejki.

on\_move\_down\_task(self, task) – metoda zarejestrowana na sygnał MOVE\_DOWN\_TASK\_CLICKED\_SIGNAL, przenosząca zadanie task w dół kolejki.

on\_data\_selected(self, task, selected\_filename) – metoda zarejestrowana na sygnał DATA\_CONFIG\_SELECTED\_SIGNAL, aktualizująca ścieżkę \*.inconf danego zadania task.

on\_params\_set\_selected(self, task, selected\_filename) – metoda zarejestrowana na sygnał PARAMS\_SET\_CONFIG\_SELECTED\_SIGNAL, aktualizująca ścieżkę \*.parconf danego zadania task.

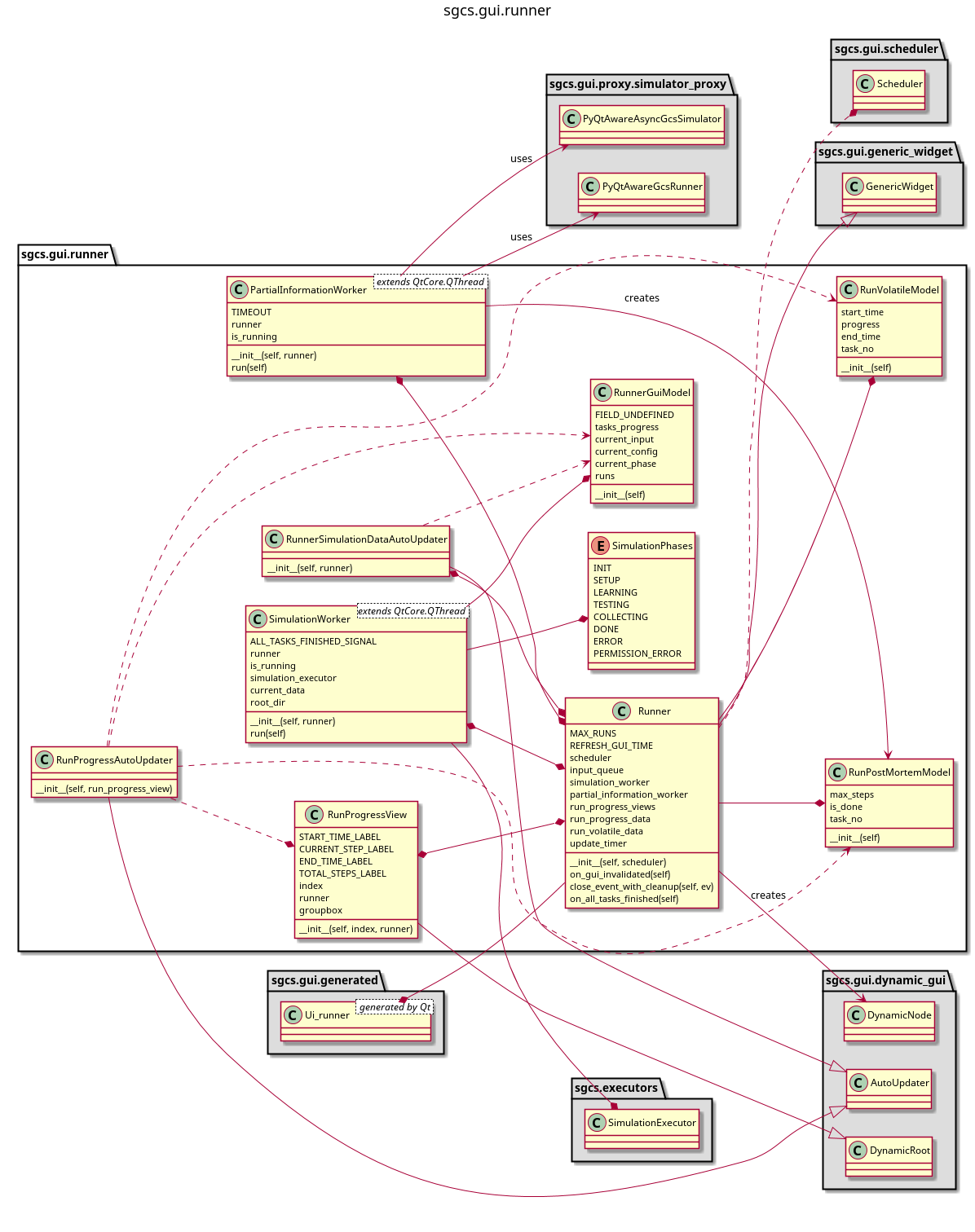
on\_add\_task\_clicked(self) – metoda zarejestrowana na naciśnięcie przycisku dodania nowego zadania. Powoduje ona dodanie nowego pustego (tj. z niesprecyzowaną ścieżką konfiguracji wejściowej i parametrów oraz pustą populacją) na końcu kolejki.

on\_clear\_tasks\_clicked(self) – metoda zarejestrowana na naciśnięcie przycisku wyczyszczenia kolejki zadań. Usuwa wszystkie zakolejkowane zadania.

on\_run\_clicked(self) – przekierowuje obecną kolejkę zadań i sterowanie do modułu sgcs.gui.runner. Oczywiście pod warunkiem poprawności danych (ścieżka artefaktów musi być podana, każde zadanie musi posiadać sprecyzowane ścieżki do pliku \*.inconf i \*.parconf, plik \*.pop jest opcjonalny).

on\_select\_output\_dir\_clicked(self) – medosa obsługująca wybór ścieżki folderu na artefakty.

sgcs.gui.runner

Moduł ten odpowiada za uruchomienie algorytmu i zapewnienie możliwości obserwacji jego działania przy pomocy GUI.

RunPostMortemModel

Model przechowujący niezmienne lub rzadko zmieniające się (na przykład pod koniec przebiegu uczącego). Nie zawiera danych przesyłanych przy pomocy kolejki Runner.input\_queue. W kodzie modułu ma czasami miejsce zamienne stosowanie terminologii „task” w odniesieniu do zadania (czyli całego uruchomienia algorytmu dla danej populacji startowej, parametrów i danych wejściowych) oraz w odniesieniu do przebiegu uczącego (pojedyncze uruchomienie etapu uczącego dla algorytmu). Należy więc zachować szczególną ostrożność w analizowaniu kodu modułu pod tym względem.

max\_steps – maksymalna ilość kroków ewolucyjnych, jaką może podjąć algorytm.

is\_done – czy dany przebieg uczący został zakończony.

task\_no – numer przebiegu uczącego.

RunVolatileModel

Model przeznaczony do przechowywania danych ulotnych. Są to dane ulegające dosyć częstym zmianom, przesyłane z algorytmu przy pomocy kolejki Runner.input\_queue. Mogą czasami nie być aktualne z obecnym stanem algorytmu, w szczególności jeżeli jest ich generowanych w danym momencie dużo, aczkolwiek są to nadal dane diagnostyczne i nie muszą być w 100% aktualne, tak długo jak pozostają w miarę aktualne.

start\_time – data rozpoczęcia przebiegu uczącego.

progress – numer obecnego kroku ewolucyjnego.

end\_time – data zakończenia przebiegu ewolucyjnego.

task\_no – numer przebiegu uczącego.

RunProgressAutoUpdater

Klasa odpowiedzialna za aktualizowanie obecnego stanu przebiegu uczącego.

RunProgressView

Klasa odpowiedzialna za prezentację obecnego stanu przebiegu uczącego. Zajmuje się początkowym ustawieniem stylu całego bloku oraz zapewnieniem widoczności odpowiednich jego elementów, podczas gdy samo zadanie odświeżania wartości zostało oddelegowane do instancji RunProgressAutoUpdater. Wyświetla GUI tylko dla przebiegów rzeczywiście istniejących (to jest RunProgressView o warotści pola index = 60 nic nie wyświetli jeżeli jest tylko 50 zaplanowanych przebiegów cyklu uczącego).

START\_TIME\_LABEL – stała, część dynamicznego GUI.

CURRENT\_STEP\_LABEL – stała, część dynamicznego GUI.

END\_TIME\_LABEL – stała, część dynamicznego GUI.

TOTAL\_STEPS\_LABEL – stała, część dynamicznego GUI.

index – numer przebiegu uczącego.

runner – uchwyt instancji klasy Runner.

groupbox – GUI generowane przez tą klasę, wystawione na potrzeby podpięcia do reszty GUI klasy Runner.

SimulationPhases

Klasa przechowująca fazy, które mają miejsce podczas działania algorytmu, wraz z ich opisami. W trakcie działania algorytmu wyróżniamy następujące fazy:

* INIT – faza początkowa, podczas której ma miejsce głównie przygotowanie interfejsu.
* SETUP – faza podczas której jest brane kolejne zadanie z kolejki, ma miejsce update GUI i przygotowanie środowiska do uruchomienia algorytmu.
* LEARNING – faza podczas której są wykonywane asynchronicznie przebiegu uczące w liczbie sprecyzowanej w odpowiednim pliku konfiguracyjnym \*.parconf.
* TESTING – faza testowa następująca po fazie uczącej, podczas której ma miejsce przebieg testowy algorytmu.
* COLLECTING – faza zbierania artefaktów. Po zakończeniu działania algorytmu wszystkie dane zgromadzone z jego przebiegu, wynikowa populacja, itd. są zapisywane w wyznaczonym wcześniej folderze. Po tym kroku następuje przejście do kroku ENVIRONMENT jeżeli kolejka zadań jest niepusta, przejście do kroku DONE w pp.
* DONE – przejście do tej fazy następuje po opróżnieniu kolejki zadań. Można bezpiecznie zamknąć okno Runner.
* ERROR – pojawia się w momencie, gdy w trakcie działania wystąpił krytyczny błąd i nie jest możliwe prawidłowe kontynuowanie działania algorytmu.
* PERMISSION\_ERROR – błąd pojawiający się, gdy podczas działania algorytm nie ma prawa dostępu do jakiegoś miejsca. Najczęściej wiąże się to z brakiem praw do zapisu w folderze z artefaktami, gdyż na przykład jakiś plik jest otwarty przez zewnętrzną aplikację. Algorytm będzie ponawiać próbę dostępu co sekundę, pozostając w tym stanie do czasu rozwiązania przez użytkownika problemu z dostępem do zasobu, po czym wznowi działanie od miejsca, gdzie je ostatnio skończył.

RunnerSimulationDataAutoUpdater

Klasa odpowiedzialna za aktualizację danych dotyczących całej obsługi kolejki zadań. Jej zadaniem jest wyświetlanie postępu w realizacji zadań z kolejki oraz ścieżki do obecnie wykorzystywanych plików wejściowych, co pozwala łatwo zidentyfikować wykonywane właśnie zadanie.

RunnerGuiModel

Model reprezentujący obecny stan kolejki zadań.

FIELD\_UNDEFINED – napis wyświetlany w przypadku, gdy dana wartość jest niezdefiniowana (na przykład czas startu przebiegu uczącego, który jeszcze się nie zaczął).

tasks\_progress – numer obecnie wykonywanego zadania (lub FIELD\_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current\_input – ścieżka do pliku \*.inconfig obecnie wykonywanego zadania (lub FIELD\_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current\_config – ścieżka do pliku \*.parconfig obecnie wykonywanego zadania (lub FIELD\_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current\_phase – faza SimulationPhase, w której obecnie Runner się znajduje.

runs – kopia kolejki zadań wykonana na podstawie tej w klasie Scheduler.

SimulationWorker

Klasa odpowiedzialna za obsługę kolejki zadań i uruchamianie algorytmu dla kolejnych zadań.

ALL\_TASKS\_FINISHED\_SIGNAL – identyfikator sygnału informującego, że wszystkie zadania z kolejki zostały wykonane.

runner – uchwyt do instancji Runner.

is\_running – zmienna boolowska wykorzystywana do delikatnego wyłączenia workera. Niestety zatrzymanie SimulationWorkera nie jest łatwym zadaniem z całymi pulami procesów zarządzanymi przez niego i może nie nastąpić natychmiast, toteż po zatrzymaniu algorytmu w połowie wykonania zaleca się ponowne uruchomienie aplikacji w celu uzyskania pewności, że wątek ten został ostatecznie zniszczony. Jest to o tyle ważne, że właśnie ten wątek jest odpowiedzialny za uruchomienie całego algorytmu, nieprawidłowo zaś zatrzymany algorytm może pozostawić osierocone procesy nadal zajmujące sporą część zasobów komputera. Ponowne uruchomienie aplikacji po nagłym przerwaniu algorytmu daje 100% pewności, że wszystkie zasoby zostały zwolnione.

simulation\_executor – instancja obiektu SimulationExecutor zajmującego się logiką biznesową przebiegu symulacji oraz zbieraniem artefaktów.

current\_data – instancja RunnerGuiModel.

root\_dir – ścieżka do folderu na artefakty.

run(self) – główna metoda wątku. Zdecydowanie nie zaleca się jej ręcznego wywoływania, jako że jest to metoda zajmująca niezwykle dużo czasu (wykonuje po kolei wszystkie zadania). Zamiast tego dobrze jest wystartować ją w osobnym wątku poprzez wywołanie metody QtCore.Qthread.start(self).

PartialInformationWorker

Klasa zajmująca się odbieraniem wiadomości diagnostycznych przesyłanych z wnętrza algorytmu przy pomocy Runner.input\_queue. Są to czasy rozpoczęcia, zakończenia przebiegu uczącego oraz obecny krok ewolucyjny.

TIMEOUT – maksymalny czas (w sekundach) oczekiwania na wiadomość w Runner.input\_queue. W przypadku przekroczenia tego czasu worker sprawdzi swój warunek zakończenia pracy i jeżeli będzie miał ją kontynuować, to przejdzie do dalszego czekania na wiadomość. Przerwy w oczekiwaniu na wiadomość są konieczne w celu umożliwienia workerowi bezpieczne zakończenie pracy w przypadku problemu z działaniem algorytmu. Wybrano 10 sekund uniknięcia obciążania procesora częstym łapaniem wyjątków i sprawdzaniem warunków działania. Oznacza to też, że w przypadku przedwczesnego zakończenia działania algorytmu wątek zakończy swoje działanie najpóźniej w 10 sekund (czas obsługi pojedynczej wiadomości jest w tym przypadku pomijalnie mały).

runner – uchwyt instancji klasy Runner.

is\_running – zmienna boolowska, której przestawienie na False spowoduje zakończenie działania wątku w co najwyżej 10 sekund (patrz opis pola TIMEOUT powyżej).

run(self) – główna metoda wątku. W celu asynchronicznego wywołania należy stosować QtCore.Qthread.start(self).

Runner

Klasa odpowiedzialna za obsługę GUI podczas działania algorytmu. W zasadzie oddelegowuje ona całą logikę do swoich klas składowych, sama przechowując kilka współdzielonych przez nie obiektów Zapewnia też prawidłowe zwalnianie zasobów w przypadku przedwczesnego zakończenia działania algorytmu, wyświetlenie odpowiedniego powiadomienia po planowanym zakończeniu działania algorytmu oraz okresowe odświeżanie GUI.

MAX\_RUNS – maksymalna ilość przebiegów procesu uczenia. Jeżeli algorytm uruchomi większą ilość cykli uczących, wówczas nie będą one prawidłowo wyświetlane. Wartość tą można zwiększyć, aczkolwiek PyQt zawsze zarezerwuje zasoby niezbędne do stworzenia maksymalnego GUI, co może spowodować wolniejsze działanie programu.

REFRESH\_GUI\_TIME – wartość tego pola definiuje co ile milisekund następi odświeżenie GUI.

scheduler – uchwyt schedulera, w którym zaprojektowano to uruchomienie.

input\_queue – kolejka wiadomości typu multiprocessing.Queue. Wykorzystywana do przesyłania wiadomości diagnostycznych z wnętrza działającego algorytmu.

simulation\_worker – instancja klasy SimulationWorker.

partial\_information\_worker – instancja klasy PartialInformationWorker.

run\_progress\_views – lista (o długości MAX\_RUNS) obiektów typu RunProgressView.

run\_progress\_data – lista obiektów RunVolatileModel.

update\_timer – instancja obiektu QtCore.QTimer, wywołujący odświeżenie GUI co REFRESH\_GUI\_TIME milisekund.

on\_gui\_invalidated(self) – metoda wymuszająca odświeżenie całego GUI.

close\_event\_with\_cleanup(self, ev) – rozszerzenie standardowej obsługi zamknięcia okna, wymuszające zatrzymanie wszystkich podległych wątków w pierwszej kolejności.

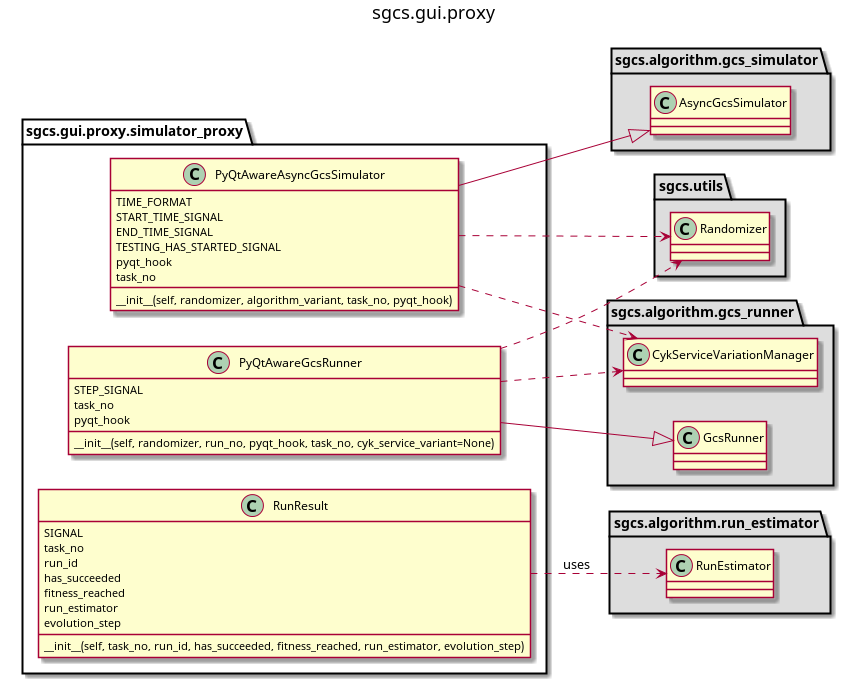
on\_all\_tasks\_finished(self) – metoda wywoływana w chwili planowanego ukończenia wszystkich zadań – powoduje wyświetlenie stosownego komunikatu.

sgcs.gui.proxy

W bibliotece podjęto starania mające na celu uniezależnienie biblioteki od wykorzystywanego GUI. Niestety GUI potrzebuje różnego rodzaju informacji na różnych etapach działania algorytmu i pełne rozdzielenie ich od siebie okazało się nietrywialnym zadaniem. Ostatecznie zdecydowano się na następujące rozwiązanie – cały algorytm został napisany w tak, że będzie on praktycznie nieświadomy istnienia GUI, natomiast moduł sgcs.gui.proxy stanowić będzie rozszerzenie jego najwyższego poziomu abstrakcji w celu udostępnienia niezbędnych GUI danych. Umożliwia to uruchamianie algorytmu w całkowitej separacji od GUI.

Zatem moduł ten zawiera prosty protokół asynchronicznej komunikacji pomiędzy algorytmem a GUI oraz klasy świadome tego nowego mechanizmu.

Wbrew mylnemu nazewnictwu stosowanemu wewnątrz modułu nie zawiera on żadnych zależności względem biblioteki PyQt. Stanowi rozszerzenie interfejsu algorytmu, które może być wykorzystane przez dowolne GUI, nawet przez aplikację konsolową.

RunResult

Klasa odpowiedzialna za wysyłanie pełnego zestawu danych zakończonego przebiegu uczącego algorytmu.

SIGNAL – identyfikator sygnału wysyłanego po zakończeniu przebiegu uczącego algorytmu.

task\_no – numer aktualnie wykonywanego zadania.

run\_id – numer przebiegu uczącego algorytmu.

has\_succeded – zmienna boolowska informująca czy przebieg zakończył się sukcesem.

fitness\_reached – osiągnięty poziom fitness f z przedziału [0;1]

run\_estimator – obiekt run\_estimator zawierający statystyki danego przebiegu.

evolution\_step – ilość wykonanych kroków ewolucyjnych.

PyQtAwareGcsRunner

Klasa rozszerzająca klasę GcsRunner o wysyłanie sygnałów kroku ewolucyjnego po każdym zakończonym kroku ewolucyjnym.

STEP\_SIGNAL – identyfikator sygnału zakończonego kroku ewolucyjnego.

task\_no – numer wykonywanego zadania.

pyqt\_hook – uchwyt do kolejki Runner.input\_queue z pomocą której następuje wysyłanie wiadomości.

PyQtAwareAsyncGcsSimulator

Klasa rozszerzająca klasę AsyncGcsSimulator o wysyłanie sygnałów rozpoczęcia, zakończenia oraz przejścia do cyklu testowego algorytmu.

TIME\_FORMAT – stała, format zapisu wysyłanego czasu.

START\_TIME\_SIGNAL – identyfikator sygnału czasu rozpoczęcia działania danego przebiegu algorytmu (zarówno uczącego, jak i testowego).

END\_TIME\_SIGNAL – identyfikator sygnału czasu zakończenia działania danego przebiegu algorytmu (zarówno uczącego, jak i testowego).

TESTING\_HAS\_STARTED – identyfikator sygnału rozpoczęcia cyklu testowego algorytmu.

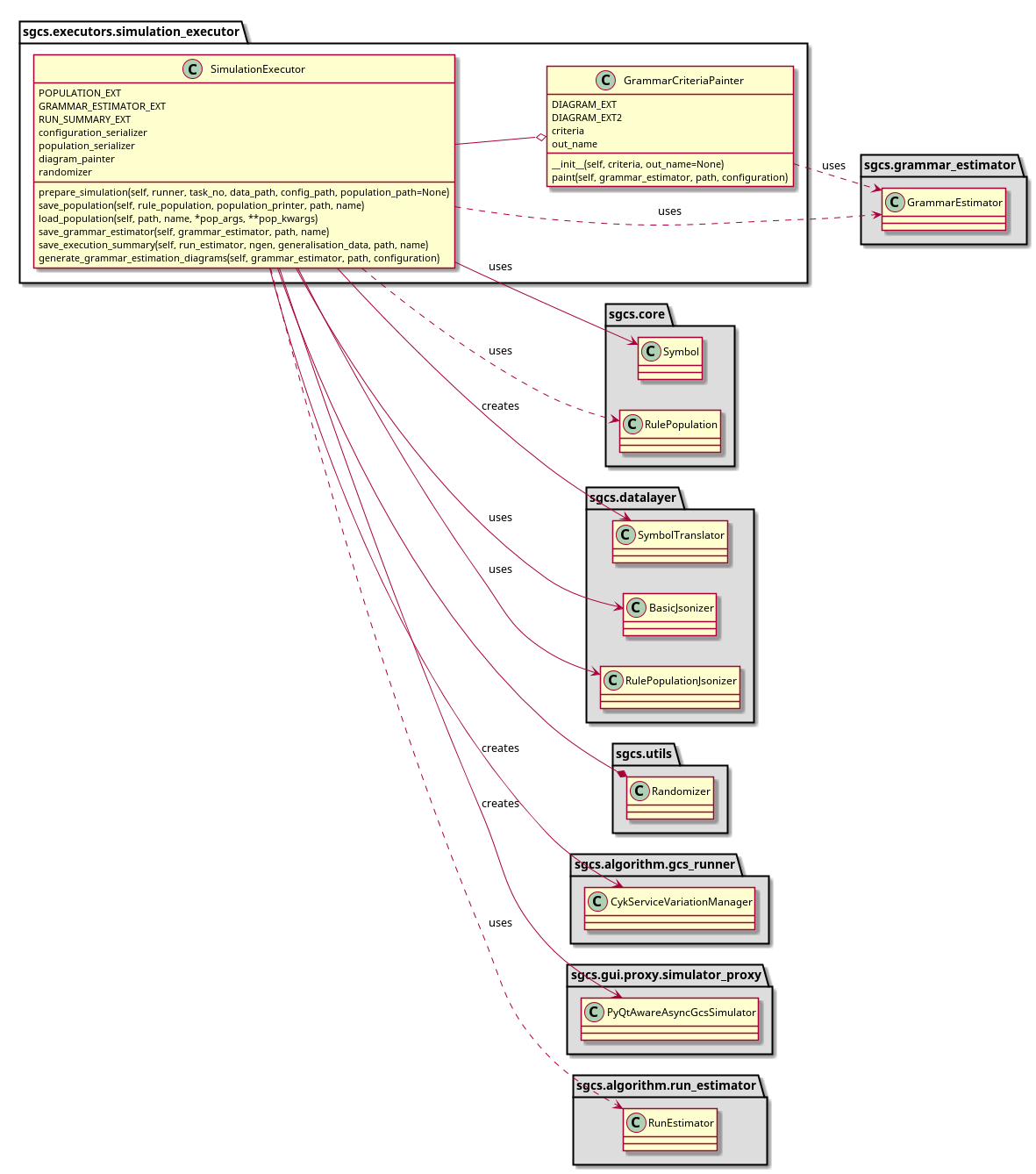
task\_no – numer wykonywanego zadania.

pyqt\_hook – uchwyt do kolejki Runner.input\_queue z pomocą której następuje wysyłanie wiadomości.

Warstwa wykonawców

Cała logika biznesowa biblioteki znajduje się w przestrzeni nazw sgcs.executors. Warstwa ta posiada stosunkowo niewiele funkcjonalności – główne zadania egzekutorów to tłumaczenie danych z reprezentacji wewnętrznej na łatwą do odczytu dla człowieka, generowanie diagramów oraz zbieranie artefaktów z działania algorytmu.

sgcs.executors.simulation\_executor



Moduł ten skupia się na obsłużeniu logiki występującej podczas działania algorytmu. Zajmuje się prawidłowym uruchomieniem algorytmu oraz zapisywaniem artefaktów, w tym rysowaniem diagramów.

SimulationExecutor

Klasa ta łączy większość logiki systemu, zapewniając możliwość załadowania niezbędnych danych z pomocą modułu sgcs.datalayer, zainicjować klasy z modułu sgcs.algorithm odpowiednimi danymi, a po wykonaniu wygenerować artefakty i zapisać je ponownie z pomocą modułu sgcs.datalayer do folderu artefaktów.

POPULATION\_EXT – stała, rozszerzenie plików populacji wykorzystywane w całym systemie („.pop”).

GRAMMAR\_ESTIMATOR\_EXT – stała, rozszerzenie plików oceny gramatyki wykorzystywane w całym systemie („.grest”).

RUN\_SUMMARY\_EXT – stała, rozszerzenie plików podsumowania symulacji wykorzystywane w całym systemie („.txt”).

configuration\_serializer – instancja BasicJsonizer, która umożliwia tłumaczenie węzłów konfiguracji na format \*.json i odwrotnie.

population\_serializer – instancja RulePopulationJsonizer, która umożliwia tłumaczenie obiektów typu RulePopulation na format \*.json i odwrotnie.

diagram\_painter – lista instancji klasy GrammarCriteriaPainter dla wszystkich wspieranych rodzajów diagramów.

randomizer – instancja obiektu Randomizer.

prepare\_simulation(self, runner, task\_no, data\_path, config\_path, population\_path=None) – metoda konstruująca z podanych danych delegata (który otrzymawszy obiekt konfiguracji uruchomi cały algorytm, a po jego zakończeniu zwróci krotkę zawierającą wyniki wykonania algorytmu), obiekt konfiguracji oraz delegata umożliwiającego wypisanie w czytelny dla człowieka sposób reguł populacji.

save\_population(self, rule\_population, population\_printer, path, name) – metoda umożliwiająca zapisanie populacji reguł rule\_population w folderze path w dwóch plikach – szczegółowego i ciężkiego w odczycie przez człowieka pliku name.pop oraz łatwego w odczycie pliku name\_view.txt.

load\_population(self, path, name, \*pop\_args, \*\*pop\_kwargs) – metoda umożliwiająca odczytanie populacji z pliku path/name.pop. Ponieważ populacja reguł wymaga dodatkowych parametrów przy tworzeniu, parametry te muszą być również dostarczone (pop\_args, pop\_kwargs).

save\_grammar\_estimator(self, grammar\_estimator, path, name) – metoda powoduje zapisanie obiektu typu GrammarEstimator w pliku path/name.grest

save\_execution\_summary(self, run\_estimator, ngen, generalization\_data, path, name) – metoda generuje plik podsumowania przebiegu algorytmu, a następnie zapisuje go w path/name.txt

generate\_grammar\_estimation\_diagrams(self, grammar\_estimator, path, configuration) – metoda powoduje wygenerowanie wszystkich wspieranych (lista diagram\_painter) rodzajów diagramów i zapisanie ich w folderze path.

GrammarCriteriaPainter

Klasa zdolna do wygenerowania pojedynczego diagramu. W obecnej postaci klasa brakuje trochę do bycia skończonej – wiele cech diagramów, które mogłyby być parametryzowalne, nie są, oprócz tego brak wsparcia dla generowania diagramów w postaci grafiki wektorowej, co byłoby bardzo wygodne w szczególności właśnie w przypadku diagramów. Większość logiki umożliwiającej to jest już zaimplementowana i niewiele potrzeba, aby rozszerzyć tę klasę o nowe możliwości, tymczasem spełnia ona wszystkie wymagania niezbędne do rysowania wymaganych przez aplikację diagramów.

DIAGRAM\_EXT – domyślne rozszerzenie grafiki rastrowej, do którego są zapisywane diagramy.

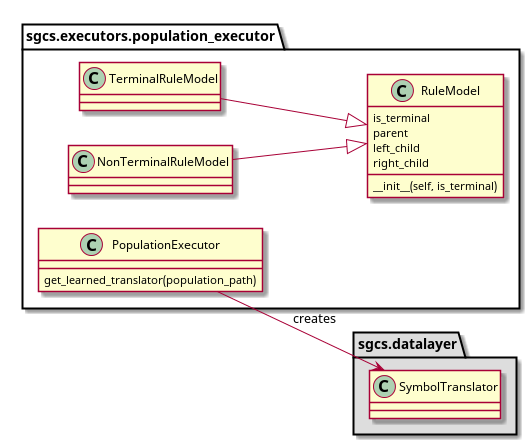
DIAGRAM\_EXT2 – domyślne rozszerzenie grafiki wektorowej, do której mogłyby być zapisywane diagramy.

criteria – nazwa i równocześnie identyfikator kryterium, które ma być przedstawione na diagramie.

out\_name – opcjonalna nazwa, która ma być stosowana na diagramach zamiast criteria (alternatywna reprezentacja).

paint(self, grammar\_estimator, path, configuration) – metoda rysująca, a następnie zapisująca diagram do plików (na razie tylko postać rastrowa, wektorowa może zostać dodana w przyszłości).

sgcs.executors.population\_executor



Moduł skupia się na zapewnieniu obsługi populacji poza działającym algorytmem. Jest to stosunkowo prosty moduł, większość funkcjonalności jest realizowana przez warstwy niższe (sgcs.core oraz sgcs.datalayer).

RuleModel

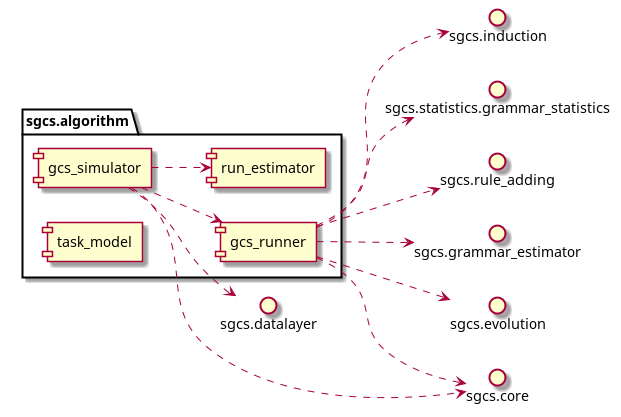
Prosta klasa do przechowywania reguł w czytelnej dla człowieka formie. Posiada dwie klasy dziedziczące po niej (TerminalRuleModel oraz NonTerminalRuleModel).

PopulationExecutor

Klasa umożliwiająca dostęp do obiektów populacji poza działającym algorytmem.

get\_learned\_translator(population\_path) – po podaniu ścieżki do zbioru uczącego population\_path wczytuje go w całości, w efekcie zwracając obiekt SymbolTranslator. Obiekt ten jest zdolny do tłumaczenia symboli w dwie strony – z reprezentacji wewnętrznej do przyjaznej człowiekowi i odwrotnie, jest zatem niezbędny do interaktywnej obsługi populacji poza działającym algorytmem.

Właściwy algorytm



Właściwy algorytm zawiera całą logikę działania algorytmu klasyfikacji gramatycznej, całkowicie odseparowaną od interfejsu graficznego oraz od niskopoziomowych funkcjonalności jak na przykład obsługa systemu plików. Warstwa ta zapewnia spójny interfejs do uruchamiania algorytmu w dowolnym wariancie – domyślnie wspiera GCS, sGCS oraz neg-sGCS wraz z licznymi parametrami i mechanizmami usprawniającymi ich działanie.

sgcs.algorithm.task\_model

Moduł ten zawiera jedynie jedną prostą klasę – TaskModel. Jego pozostałe funkcjonalności zostały przeniesione do innych modułów i powinien zostać usunięty po przeniesieniu klasy TaskModel.

TaskModel

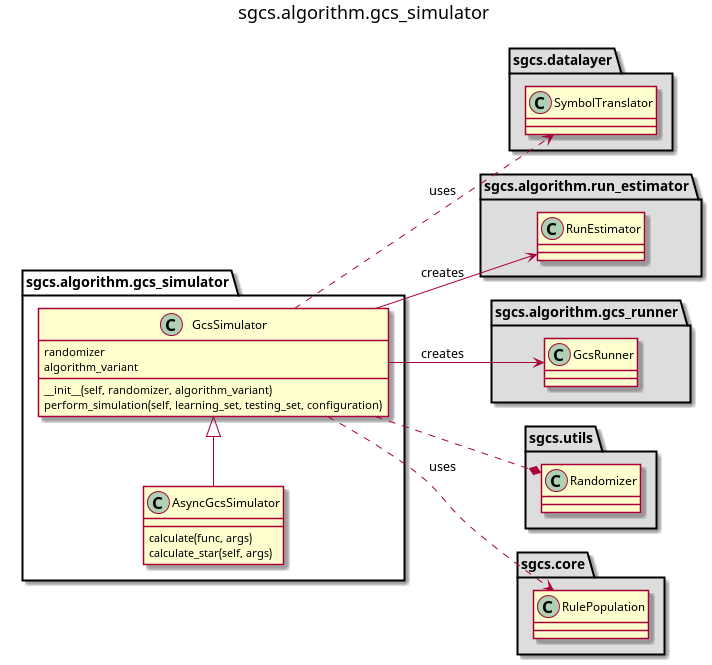
Klasa mająca za zadanie reprezentować pojedyncze zadanie w kolejce zaplanowanych uruchomień algorytmów. Zawiera 3 pola przechowujące ścieżki do związanych z zadaniem zasobów.

data\_configuration – ścieżka do pliku \*.inconf dla zadania.

params\_configuration – ścieżka do pliku \*.parconf dla zadania.

population\_configuration – ścieżka do pliku \*.pop dla zadania.

sgcs.algorithm.gcs\_simulator

Moduł ten zawiera klasy GcsSimulator oraz AsyncGcsSimulator. Są to klasy najwyższego poziomu wykonujące w swoim ciele cały algorytm klasyfikacji gramatyki.

GcsSimulator

Wbrew nazwie klasa ta zajmuje się obsługą dowolnego algorytmu klasyfikacji gramatycznej. Obiektem decydującym o różnym zachowaniu jest algorithm\_variant, aczkolwiek na tym poziomie jest on tylko przekazywany w dół struktury do klasy GcsRunner. GcsSimulator uruchamia configuration.max\_algorithm\_runs przebieg uczący, po czym przechodzi do pojedynczego przebiegu testowego. Całość jest wykonana sekwencyjnie.

randomizer – uchwyt obiektu Randomizer.

algorithm\_variant – obiekt typu CykServiceVariationManager.

perform\_simulation(self, learning\_set, testing\_set, configuration) – funkcja pobiera SymbolTranslator wygenerowany na podstawie zbioru uczącego, SymbolTranslator wygenerowany na podstawie zbioru testowego oraz obiekt konfiguracji. Dokonuje przebiegów uczacych, a następnie testowych, po czym zwraca krotkę zawierającą dane zebrane podczas procesu.

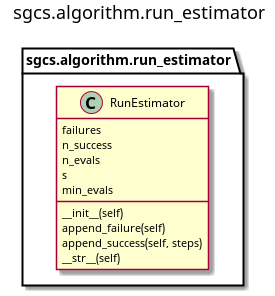
AsyncGcsSimulator

Klasa ta w przeciwieństwie do swojej nadklasy uruchamia cykle uczące równolegle zamiast sekwencyjnie. Na samym początku zostaje utworzona pula procesów o takiej liczności jak ilość rdzeni logicznych dostępnych na danym komputerze (czyli uwzględnia ilość rdzeni fizycznych oraz technologie typu Hyper-Threading). Następnie wszystkie przebiegi cyklu uczącego są wykonywane asynchronicznie przez tą pulę procesów. Główny proces zajmuje się jedynie zbieraniem wyników oraz ewentualnym łączeniem niektórych danych (np. przeliczanie średniej wartości funkcji fitness w danym kroku ewolucyjnym dla wszystkich przebiegów uczących algorytmu). Dzięki temu wykorzystanie asynchronicznego zamiast sekwencyjnego symulatora pozostaje transparentne i daje zauważalne przyspieszenie w przypadku parsowania gramatyk wymagających dużej ilości kroków ewolucyjnych do wyuczenia się gramatyki.

calculate(func, args) – funkcja pomocnicza wywołująca pierwszy parametr z listą parametrów przekazaną w postaci drugiego argumentu.

calculate\_star – funkcja przygotowana do wykorzystania z multiprocessing.imap, w wygodny sposób przekazuje parametry do funkcji, zapewnia inną wartość seeda dla każdego z procesów oraz rozszerza zwracaną wartość funkcji o dodatkowe informacje wymuszone przez asynchroniczność wykonania obliczeń.

sgcs.algorithm.run\_estimator



Klasa ta odpowiada za gromadzenie statystyk wykonania algorytmu. Po każdym ukończeniu przebiegu uczącego jest wołana metoda append\_failure lub append\_success. W efekcie po zakończeniu uczenia instancja tej klasy jest w stanie podać wartości n\_success, n\_evals, s oraz min\_evals umożliwiające ocenę przebiegu procesu uczenia.

failures – ilość przebiegów uczących zakończonych porażką.

n\_success – parametr nSuccess (ilość przebiegów uczących zakończonych znalezieniem satysfakcjonującej gramatyki).

n\_evals – parametr nEvals (średnia ilość kroków potrzebnych do znalezienia satysfakcjonującej gramatyki).

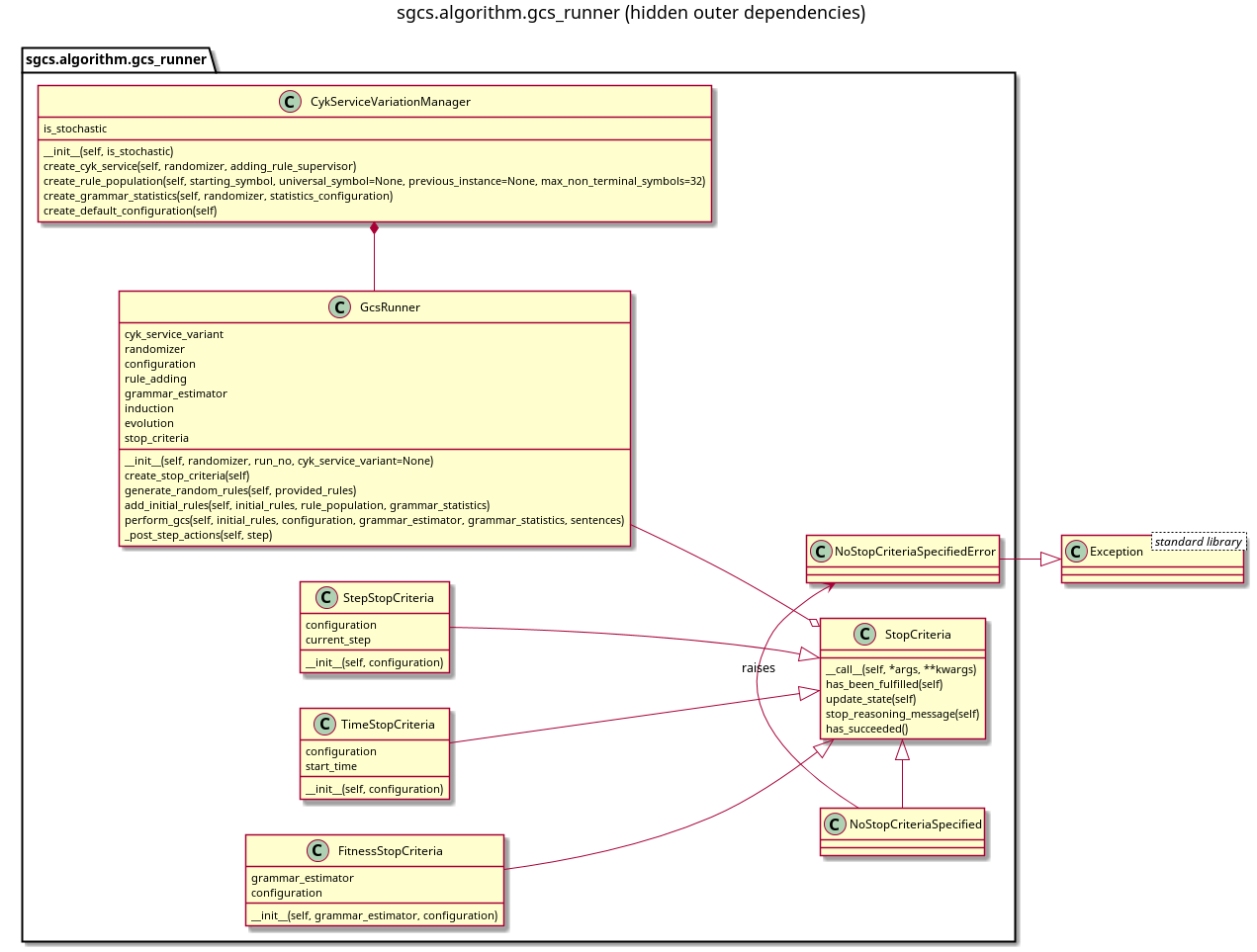
s – parametr s (odchylenie standardowe wartości nEvals).

min\_evals – parametr minEvals (minimalna ilość kroków potrzebnych do znalezienia satysfakcjonującej gramatyki).

append\_failure(self) – zapisuje przebieg jako zakończony porażką.

append\_success(self, steps) – zapisuje przebieg jako zakończony sukcesem (wraz z ilością kroków).

sgcs.algorithm.gcs\_runner

Moduł sgcs.algorithm.gcs\_runner gromadzi całą logikę pojedynczego przebiegu algorytmu w jedną metodę. Poszczególne klasy tego modułu posiadają wiele zależności od innych modułów i przedstawienie ich w czytelny sposób na diagramie jest praktycznie niemożliwe, dlatego też zależności międzymodułowe nie zostały przedstawione. Jest on zależny od wielu klas należących do następujących modułów:

* sgcs.utils;
* sgcs.rule\_adding;
* sgcs.core;
* sgcs.statistics,grammar\_statistics;
* sgcs.grammar\_estimator;
* sgcs.evolution;
* sgcs.induction;

CykServiceVariationManager

Klasa ta zarządza różnicami w zachowaniu pomiędzy algorytmami stochastycznymi (sGCS, neg-sGCS) a klasycznymi (GCS). Zamysłem jest tutaj wystawienie wspólnego API dla obu grup algorytmów, ukrywając w ciałach metod wszelkie różnice pomiędzy nimi. Dzięki temu klient tej klasy (GcsRunner) może pozostać w całkowitej nieświadomości czy wykorzystywany przez niego algorytm jest stochastyczny, czy też nie.

is\_stochastic – przechowuje informację czy obecnie wykorzystywany algorytm jest stochastyczny, czy też nie.

create\_cyk\_service(self, randomizer, adding\_rule\_supervisor) – metoda tworzy instancję typu CykService w przypadku algorytmu klasycznego lub StochasticCykService w przypadku algorytmu stochastycznego.

create\_rule\_population(self, starting\_symbol, universal\_symbol=None, previous\_instance=None, max\_non\_terminal\_symbols=32) – metoda tworząca instancję RulePopulation lub StochasticRulePopulation.

create\_rule\_statistics(self, randomizer, statistics\_configuration) – metoda tworząca odpowiednią instancję liczącą statystyki gramatyki (klasyczną lub pasieki). Należy zwrócić uwagę, że decyzja jest w tym przypadku zależna od tego czy dana konfiguracja akceptuje uczenie na zdaniach negatywnych (Pasieka nie, klasyczna tak).

create\_default\_configuration(self) – metoda tworząca domyślny obiekt konfiguracji dla danego algorytmu.

StopCriteria

Abstrakcyjna klasa realizująca pojedynczy warunek stopu dla głównej pętli algorytmu.

Przykład użycia:

criterias = [StopCriteria1(), StopCriteria2()]

while not any(cr() for cr in criterias):

…

message = „Success!” if any(cr.has\_succeded() for cr in criterias) else „Failure!”

print(message)

has\_been\_fullfilled(self) – metoda sprawdzająca spełnienie warunku.

update\_state(self) – metoda aktualizująca wewnętrzny stan instancji kryterium. Kiedy wywołujemy metodę \_\_call\_\_ kryterium, wówczas najpierw zostanie wywołana metoda update\_state, a następnie dopiero zwrócony rezultat has\_been\_fullfilled.

stop\_reasoning\_message(self) – metoda wypisująca przyczynę zatrzymania (przydatne przy kilku warunkach stopu).

has\_succeded() – metoda informująca czy nastąpienie tego warunku stopu powoduje sukces, czy też nie. Dzięki temu możemy mieszać różne kryteria razem i na podstawie ich stanu po opuszczeniu pętli ustalić, czy nastąpił sukces, czy też porażka. Domyślnie zwraca wartość False.

NoStopCriteriaSpecified

Klasa dziedzicząca po StopCriteria. Przy próbie wywołania metody has\_been\_fullfilled podnosi wyjątek NoStopCriteriaSpecifiedError. Jest to przydatne gdy dysponuje się klasą nieposiadającą domyślnych kryteriów stopu i chce się mieć pewność, że użytkownik zapewni własne.

FitnessStopCriteria

Warunek stopu spełniony, kiedy przejście gramatyki zakończy się uzyskaniem fitness na poziomie configuration.satisfying\_fitness. Jego metoda has\_succeded() zwraca wartość True.

grammar\_estimator – uchwyt do instancji GrammarEstimator, która pozwala sprawdzić obecną wartość fitness gramatyki.

configuration – uchwyt do węzła konfiguracyjnego pozwalający odczytać wartość configuration.satisfying\_fitness.

StepStopCriteria

Warunek stopu spełniony, kiedy current\_step przekroczy configuration.max\_algorithm\_steps. current\_step, jest preinkrementowany przy każdym wywołaniu metody update\_state (czyli również przy \_\_call\_\_).

configuration – uchwyt do węzła konfiguracyjnego pozwalający odczytać wartość configuration.max\_algorithm\_steps.

current\_step – obecny krok (zaczyna liczenie od 0).

TimeStopCriteria

Warunek stopu spełniony, kiedy obecny czas jest większy od start\_time + configuration.max\_execution\_time.

configuration – uchwyt do węzła konfiguracyjnego pozwalający na odczytanie wartości configuration.max\_execution\_time

start\_time – czas w chwili utworzenia tego obiektu.

GcsRunner

Klasa ta realizuje główną pętlę algorytmu w sposób niezależny od tego, czy jest stochastyczny i czy wykorzystuje zdania negatywne w procesie uczenia. Wykorzystuje wszystkie warunki stopu (FitnessStopCriteria, TimeStopCriteria oraz StepStopCriteria).

cyk\_service\_variant – klasa zajmująca się obsługą różnic pomiędzy różnymi wariantami agorytmu.

randomizer – uchwyt instancji Randomizer.

configuration – uchwyt węzła konfiguracyjnego.

rule\_adding – instancja klasy RuleAddingSupervisor.

grammar\_estimator – obiekt klasy GrammarEstimator.

induction – instancja klasy CykService.

evolution – instancja klasy EvolutionService.

stop\_criteria – lista obiektów StopCriteria wykorzystywanych jako warunek stopu głównej pętli algorytmu.

create\_stop\_criteria(self) – Funkcja inicjująca listę stop\_criteria nowymi instancjami klas FitnessStopCriteria, StepStopCriteria oraz TimeStopCriteria.

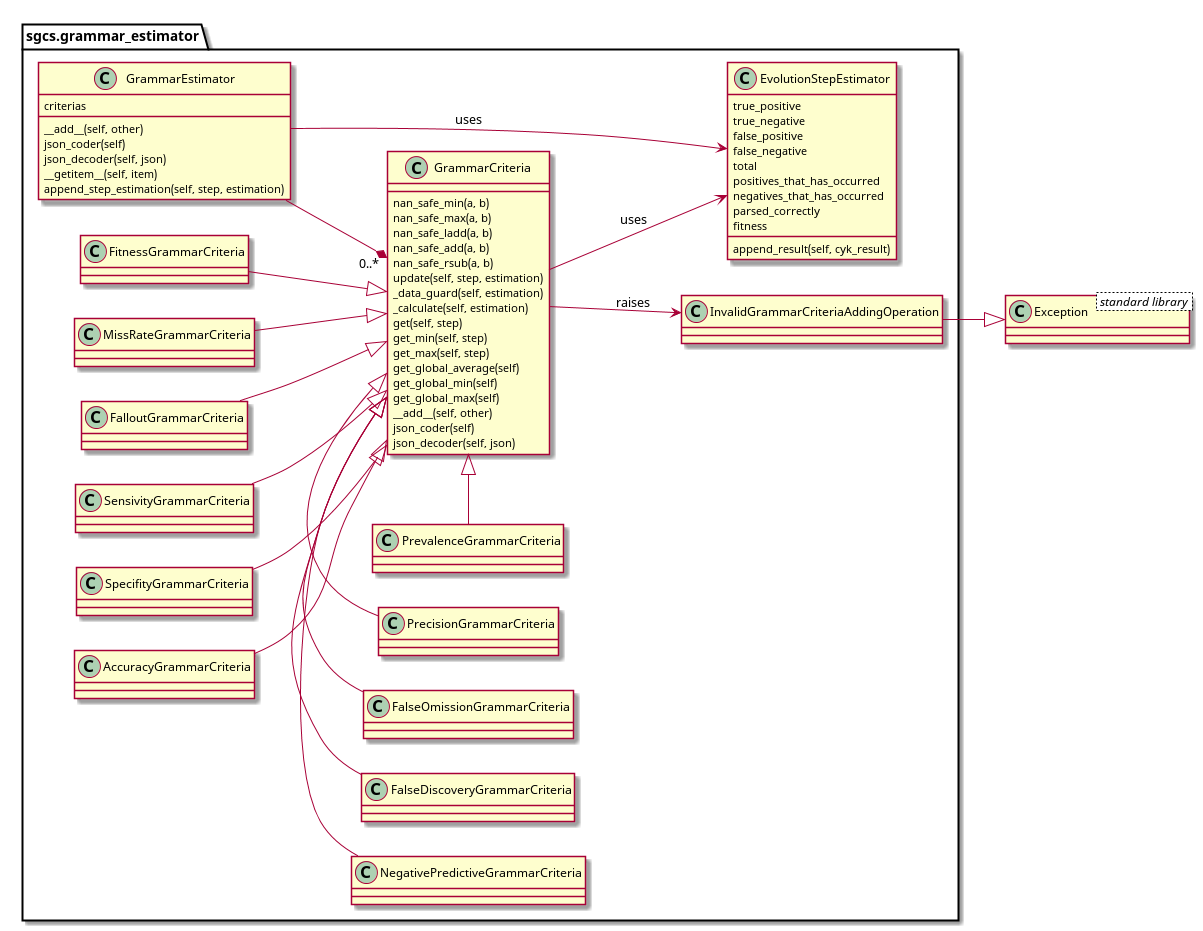
generate\_random\_rules(self, provided\_rules) – metoda zwiększająca zbiór reguł początkowych o configuration.rule.random\_starting\_population\_size – len(provided\_rules).

add\_initial\_rules(self, initial\_rules, rule\_population, grammar\_statistics) – metoda dodająca listę reguł startowych initial\_rules do populacji rule\_population z uwzględnieniem aktualizacji grammar\_statistics.

perform\_gcs(self, initial\_rules, configuration, grammar\_estimator, grammar\_statistics, sentences) – główna metoda algorytmu. Inicjuje zestaw reguł początkowych i wykonuje kolejne kroki ewolucyjne aż do momentu nastąpienia któregoś z warunków stopu.

\_post\_step\_actions(self, step) – metoda zawierająca zestaw działań wykonywanych po zakończeniu pojedynczego kroku ewolucyjnego. Istniejąca implementacja jest pusta, daje to jednak możliwości rozwoju klasom dziedziczącym (jak na przykład AsyncGcsRunner).

sgcs.grammar\_estimator

Moduł ten wykorzystuje się do gromadzenia informacji o kondycji algorytmu. Na podstawie ilości należących sparsowanych, nienależących sparsowanych, nienależących niesparsowanych oraz należących niesparsowanych zdań są obliczane wartości macierzy omyłek dla każdego kroku ewolucyjnego (oraz samego testu generalizacji).

EvolutionStepEstimator

Klasa przechowująca ilość zdań pozytywnych sparsowanych (True Positive), ilość zdań negatywnych niesparsowanych (True Negative), ilość zdań pozytywnych niesparsowanych (False Negative) oraz ilość zdań negatywnych sparsowanych (False Positive). Zawiera jeszcze kilka pomocniczych miar i sama zajmuje się kontrolą tych wszystkich wartości – wystarczy wywołać jej metodę append\_result wraz z obiektem CykResult zwróconym przez instancję CykService.

true\_positive – pole służące do odczytu ilości sparsowanych zdań prawdziwych.

true\_negative – pole służące do odczytu ilości niesparsowanych zdań negatywnych.

false\_positive – pole służące do odczytu ilości sparsowanych zdań negatywnych.

false\_negative – pole służące do odczytu ilości niesparsowanych zdań pozytywnych.

total – ilość wszystkich sparsowanych i niesparsowanych zdań

positives\_that\_has\_occurred – zmienna opisująca ilość pozytywnych zdań, które wystąpiły.

negatives\_that\_has\_occurred – zmienna opisująca ilość negatywnych zdań, które wystąpiły.

parsed\_correctly – ilość prawidłowo sparsowanych zdań.

fitness – osiągnięty w kroku poziom fitness.

GrammarCriteria

Klasa odpowiedzialna za gromadzenie analizę wielu kroków ewolucyjnych algorytmów. Przyjmuje kolejne obiekty EvolutionStepEstimator ze wszystkich uruchomień i na ich podstawie generuje daną statystykę na przestrzeni wielu kroków, w każdym z nich obliczając ją na podstawie wszystkich dostępnych przebiegów algorytmu. Wiele innych klas dziedziczy w tym module po tej klasie, nie ma jednak sensu poświęcać im osobnego miejsca na omówienie, jako że implementują one po prostu metody \_calculate oraz \_data\_guard zgodnie ze wzorem na statystykę, którą reprezentują (co jest zawarte w ich nazwie).

nan\_safe\_min(a, b) – funkcja min odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza nie jest liczbą).

nan\_safe\_max(a, b) – funkcja max odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza nie jest liczbą).

nan\_safe\_ladd(a, b) – operacja dodawania lewostronnie odporna na wartość 'nan' (zwraca b w przypadku gdy a nie jest liczbą).

nan\_safe\_add(a, b) – operacja dodawania odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza z nich nie jest liczbą).

nan\_safe\_rsub(a, b) – prawostronnie odporna na wartość 'nan' operacja odejmowania (zwraca a w przypadku gdy b nie jest liczbą).

update(self, step\_estimation) – aktualizuje przebieg statystyki na podstawie kolejnego obiektu step\_estimation.

\_data\_guard(self, estimation) – abstrakcyjna metoda, której konkretyzacje powinny zwracać True, jeżeli dla tego estimation dana statystyka istnieje (na przykład statystyka Precision nie jest zdefiniowana jeżeli żadne zdanie nie zostało sparsowane). Zazwyczaj wystarczy, aby funkcja ta pilnowała czy mianownik wzoru danej statystyki jest niezerowy.

\_calculate(self, estimation) – abstrakcyjna metoda, której konkretyzacje zajmują się liczeniem danej statystyki (dla poprawnych danych, funkcja ta wywołuje się tylko wtedy, kiedy metoda \_data\_guard zwróciła True).

get(self, step) – pobranie średniej wartości statystyki w danym kroku.

get\_min(self, step) – pobranie wartości minimalnej statystyki w danym kroku.

get\_max(self, step) – pobranie wartości maksymalnej statystyki w danym kroku.

get\_global\_average(self) – pobranie globalnej średniej statystyki na podstawie wszystkich kroków.

get\_global\_min(self) – pobranie globalnej wartości minimalnej statystyki na podstawie wszystkich kroków.

get\_global\_max(self) – pobranie globalnej wartości maksymalnej statystyki na podstawie wszystkich kroków.

\_\_add\_\_(self, other) – umożliwia dodawanie do siebie obiektów GrammarCriteria. W wyniku dodania dwóch obiektów tworzony jest trzeci, który posiada połączoną wiedzę dwóch pozostałych na temat gramatyki. Funkcja przydatna w przypadku asynchronicznego wykonania przebiegów uczących, kiedy to trzeba zgromadzić wiedzę zdobytą przez osobne procesy.

json\_coder(self) – metoda umożliwiająca serializację do napisu w formacie json.

json\_decoder(self, json) – metoda umożliwiająca wczytanie wewnętrznego stanu obiektu z obiektu json. Metoda nie sprawdza poprawności obiektu json.

GrammarEstimator

Klasa gromadząca wszystkie statystyki gramatyki. Przechowuje je w postaci mapy nazwa – obiekt GrammarCriteria w polu criterias. Udostępnia dostęp do tych statystyk przez nazwę, interfejs do dodawania kolejnych obiektów EvolutionStepEstimation oraz operację dodawania obiektów GrammarEstimator.

criterias – pole mapujące nazwę statystyki do obiektu GrammarCriteria. Domyślnie mapa ta jest zainicjowana statystykami z macierzy pomyłek, można ją dowolnie zmodyfikować.

\_\_add\_\_(self, other) – funkcja umożliwiająca dodawanie obiektów GrammarEstimator, szczególnie przydatna przy asynchronicznym wykonywaniu przebiegów uczących.

json\_coder(self) – metoda umożliwiająca serializację do napisu w formacie json.

json\_decoder(self, json) - metoda umożliwiająca wczytanie wewnętrznego stanu obiektu z obiektu json. Metoda nie sprawdza poprawności obiektu json.

\_\_get\_item\_\_(self, item) – metoda wbudowana wprowadzająca poniższy cukier syntaktyczny:

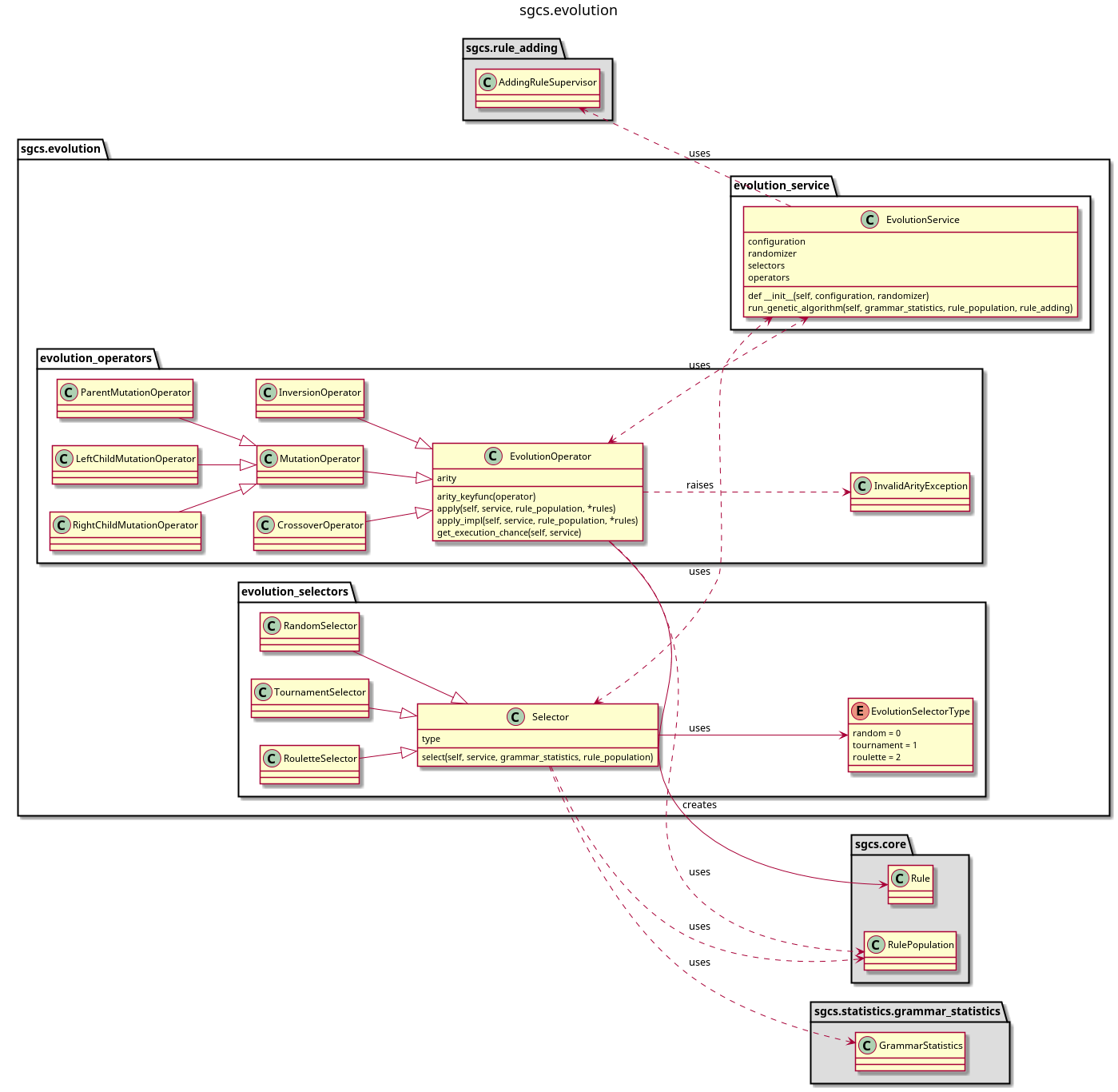
fitness\_criteria = grammar\_criteria['fitness']

Który tak naprawdę oznacza:

fitness\_criteria = grammar\_criterias.criterias['fitness']

append\_step\_estimation(self, step\_estimation) – metoda dodająca obiekt EvolutionStepEstimator step\_estimation do wszystkich statystyk w criterias (poprzez zawołanie na nich metody update).

sgcs.evolution



Moduł ewolucji zawiera całą logikę algorytmu genetycznego wykorzystywanego w algorytmie. Algorytm genetyczny można by w łatwy sposób podmienić na jakiś inny algorytm ewolucyjny, obecnie jednak moduł posiada jedynie implementację algorytmu genetycznego. Moduł ten ma na celu wygenerować nowe reguły na podstawie istniejącej populacji i jej stanu (przechowywanego w grammar\_statistics), a następnie przekazać wygenerowane reguły do modułu sgcs.rule\_adding.

EvolutionSelector

Klasa abstrakcyjna reprezentująca pojedynczy selektor.

type(self) – metoda zwracająca typ selektora (niezbędne do zgrania z węzłami konfiguracyjnymi algorytmu).

select(self, service, grammar\_statistics, rule\_population) – abstrakcyjna metoda dokonująca selekcji jednej reguły z rule\_population, przy pomocy wiedzy otrzymanej z grammar\_statistics oraz service (EvolutionService).

RandomSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji losowej.

TournamentSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji turniejowej.

RouletteSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji ruletkowej.

EvolutionOperator

Abstrakcyjna klasa realizująca logikę operatora ewolucyjnego. Zdecydowano się tutaj na wysoce abstrakcyjną definicję operatora jako funkcję przyjmującą n reguł na wejściu i zwracającą k nowych reguł na wyjściu, co umożliwia dodawanie praktycznie dowolnych operatorów genetycznych oraz czyni model algorytmu genetycznego wysoce jednolitym. Szczegóły na temat wykorzystania operatorów o nietypowej n-arności i k-krotności wartości zwracanej zostaną podane przy okazji omówienia klasy EvolutionService.

arity – arność, czyli ilość argumentów przyjmowanych na wejściu przez algorytm.

arity\_keyfunc – metoda zwracająca wartość pola arity, wygodna w użyciu w funkcjach sortujących.

apply(self, service, rule\_population, \*rules) – metoda wywołująca operator genetyczny na podanych regułach. Jeżeli len(rules) != self.arity wówczas zostaje zgłoszony wyjątek InvalidArityException informujący o nieprawidłowej arności wywołania operatora genetycznego (operator musi być zawsze wywoływany dokładnie z tyloma argumentami, ile wynosi jego arność). Następnie metoda sprawdza, czy operator powinien zostać zaaplikowany (każdy operator genetyczny jest aplikowany z pewnym prawdopodobieństwem). Jeżeli tak zwraca wynik działania metody apply\_impl, w przeciwnym przypadku zwraca niezmodyfikowane reguły.

apply\_impl(self, service, rule\_population, \*rules) – abstrakcyjna metoda skupiająca się na wykonaniu samej operacji genetycznej.

get\_execution\_chance(self, service) – metoda zwracająca wartość prawdopodobieństwa zaaplikowania danego operatora (zazwyczaj wartość ta jest odczytywana z odpowiedniego pola service.configuration.operators).

CrossoverOperator

Binarny operator genetyczny krzyżownia.

InversionOperator

Unarny operator genetyczny inwersji.

MutationOperator

Abstrakcyjny unarny operator genetyczny mutacji. Jest ukonkretniony przez trzy klasy, które zajmują się osobno mutacją poszczególnych wartości reguły (tj. każdy z nich traktujemy jako niezależny operator, wykonując niezależne losowanie czy dana mutacja ma być zaaplikowana itd.).

ParentMutationOperator

Unarny operator genetyczny mutacji ojca reguły.

LeftChildMutationOperator

Unarny operator genetyczny mutacji lewego dziecka reguły.

RightChildMutationOperator

Unarny operator genetyczny mutacji prawego dziecka reguły.

EvolutionService

Klasa odpowiedzialna za obsługę całego algorytmu genetycznego. Działanie algorytmu genetycznego odbywa się w następujących krokach

1. Wybierz x reguł z powtórzeniami z populacji, gdzie x to ilość podanych w konfiguracji selektorów przez użytkownika (mogą być to selektory różnego rodzaju) i zapisz je jako selected\_rules.
2. Wybierz elitę populacji elite (jeżeli elityzm jest włączony).
3. Zgrupuj operatory względem arności.
4. Dla każdej dostępnej arności n:
   1. Dla każdego dostępnego operatora op o arności n i krotności wartości zwracanej k:
      1. Podziel selected\_rules na krotki długości n (powtarzając początek selected\_rules w ostatniej krotce jeżeli len(selected\_rules) % n != 0.
         1. Zaaplikuj op do wszystkich krotek, listy k wyników zsumuj i zapisz jako nowe selected\_rules.
5. Dodaj selected\_rules z elityzmem do populacji przy pomocy instancji klasy AddingRuleSupervisor z modułu sgcs.rule\_adding.

Grupowanie operatorów jest stabilne względem listy operators (tak więc kolejność dwóch operatorów o tej samej arności na liście nie ulegnie zmianie).

configuration - uchwyt do węzła konfiguracyjnego.

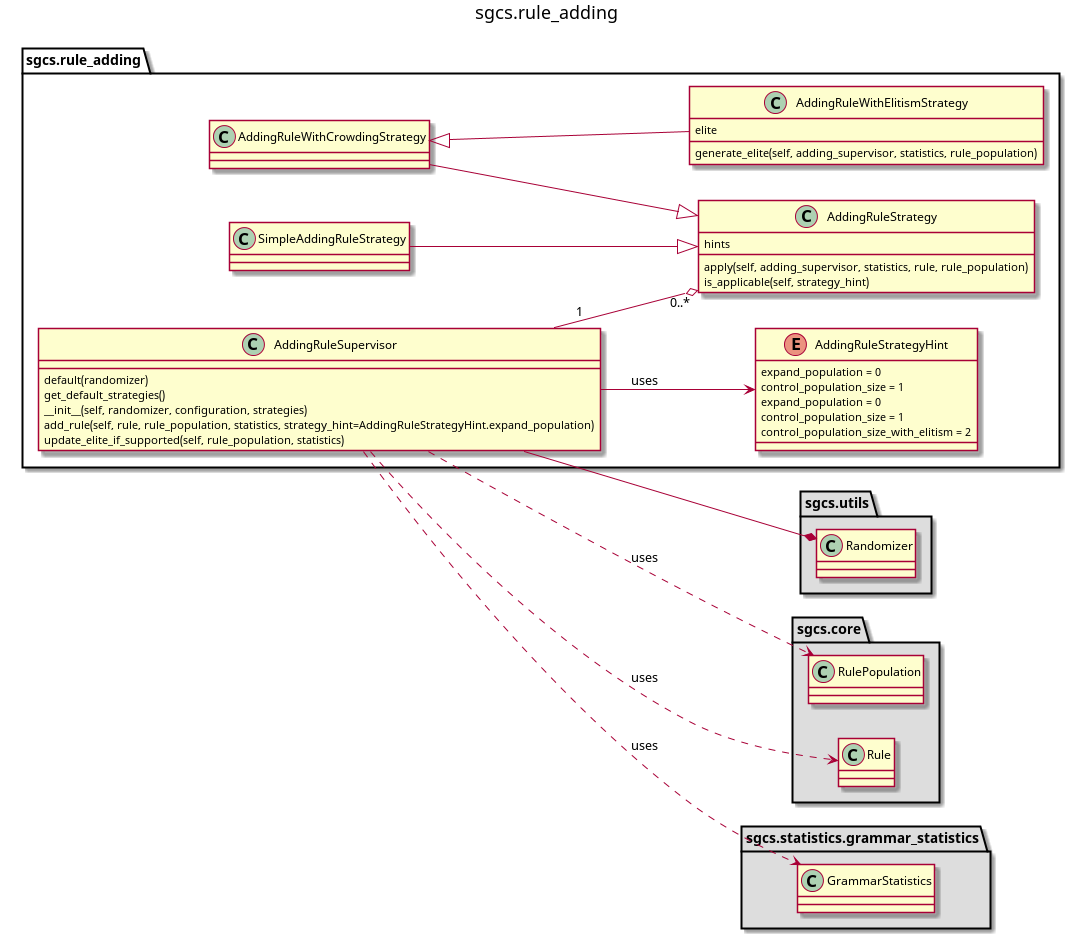
randomizer – uchwyt do instancji klasy Randomizer.

selectors – lista wspieranych selektorów (o rzeczywiście wykorzystanych decyduje konfiguracja).

operators – lista operatorów genetycznych.

run\_genetic\_algorithm(self, grammar\_statistics, rule\_population, rule\_adding, configuration) – Metoda uruchamiająca algorytm genetyczny.

sgcs.rule\_adding

Zadaniem tego modułu jest nadzorowanie dodawania nowych reguł do populacji. Praktycznie każde dodawanie nowej reguły odbywa się za pośrednictwem tego modułu.

AddingRuleSupervisor

Klasa odpowiedzialna za zarządzanie dodawaniem reguł.  
default(randomizer) – funkcja tworząca instancję o domyślnej konfiguracji.

get\_default\_strategies() - funkcja zwracająca listę zawierającą domyślny zestaw strategii.

add\_rule(self, rule, rule\_population, statistics, strategy\_hint = AddingRuleStrategyHint.expand\_population) – metoda powodująca dodanie reguły do populacji z uwzględnieniem strategy\_hint. Należy tutaj zwrócić uwagę na słowo „hint” - AddingRuleSupervisor może nadpisać to zachowanie w jednym z przypadków:

* Jeżeli strategy\_hint == AddingRuleStrategyHint.expand\_population, a osiągnęliśmy już maksymalny rozmiar populacji, wówczas będzie miało dodanie reguły z AddingRuleStrategyHind.control\_population\_size;
* Jeżeli strategy\_hint wymuszałby dodanie reguły do pustej populacji bez dokonywania zmiany jej rozmiaru, wówczas supervisor doda populację, zwiekszając tym samym jej rozmiar (do 1).

Pomijając te wyjątkowe sytuacje AddingRuleSupervisor znajdzie obiekt odpowiedniej strategii i z jej pomocą doda regułę do populacji.

update\_elite\_if\_supported(self, rule\_population, statistics) – funkcja aktualizująca stan elity przechowywanej przez funkcję. Metoda ta musi być wywoływana osobno, gdyż w przypadku dodawania kilku nowych reguł elita powinna być sprecyzowana przed dodaniem jakiejkolwiek z nich, nie powinna być generowana przy każdym dodaniu reguły.

AddingRuleStrategyHint

Klasa ta pełni funkcję wyliczenia z następującymi możliwymi wartościami:

* expand\_population;
* control\_population\_size;
* control\_population\_size\_with\_elitism.

AddingRuleStrategy

Abstrakcyjna klasa definiująca podstawową logikę funkcjonowania strategii dodawania reguł.

hints – lista AddingRuleStrategyHint, w przypadku których strategia jest aplikowalna.

apply(self, adding\_supervisor, statistics, rule, rule\_population) – metoda abstrakcyjna powodująca zaaplikowanie danej strategii.

is\_applicable(self, strategy\_hint) – metoda zwracająca True, jeżeli daną strategię da się zastosować z danym strategy\_hint.

SimpleAddingRuleStrategy

Najprostsza strategia dodawania reguł – poprzez rozszerzenie zbioru. Reguła zostaje dodana, obiekt statistics poinformowany o zmianach.

AddingRuleWithCrowdingStrategy

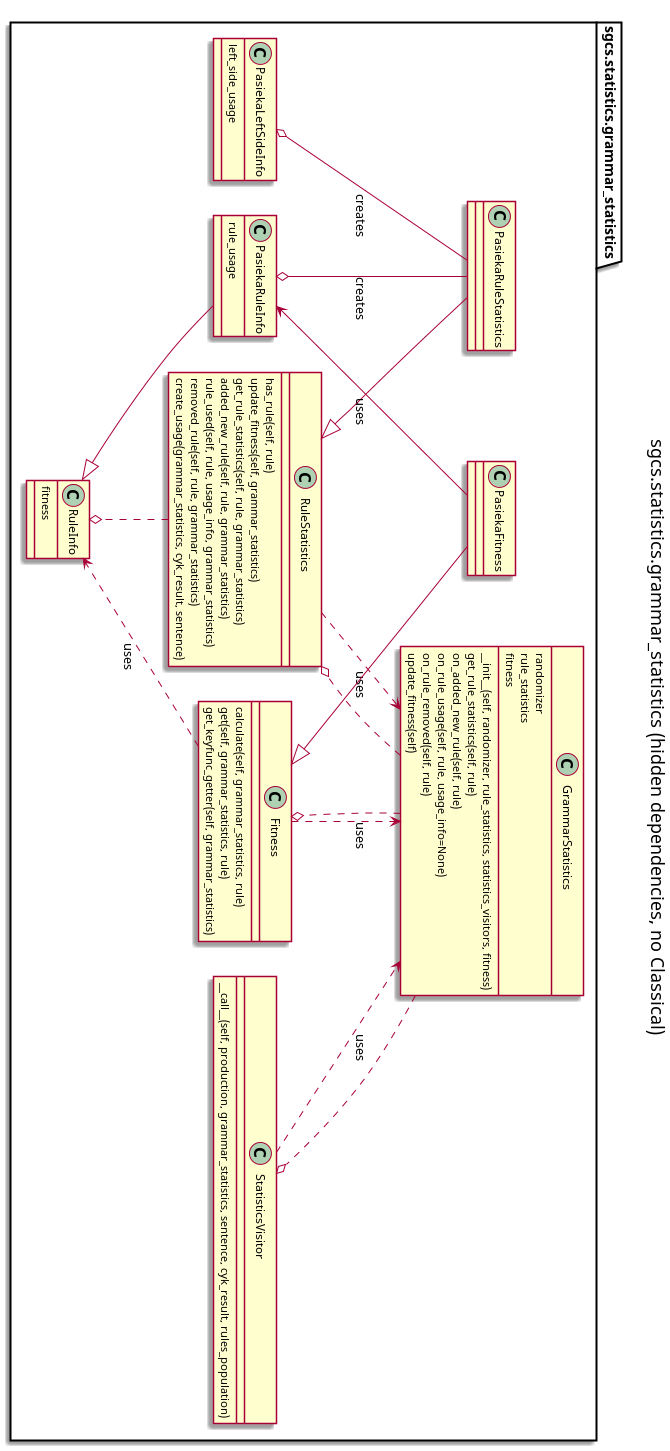
Strategia wykorzystująca mechanizm ścisku do dodawania reguł.

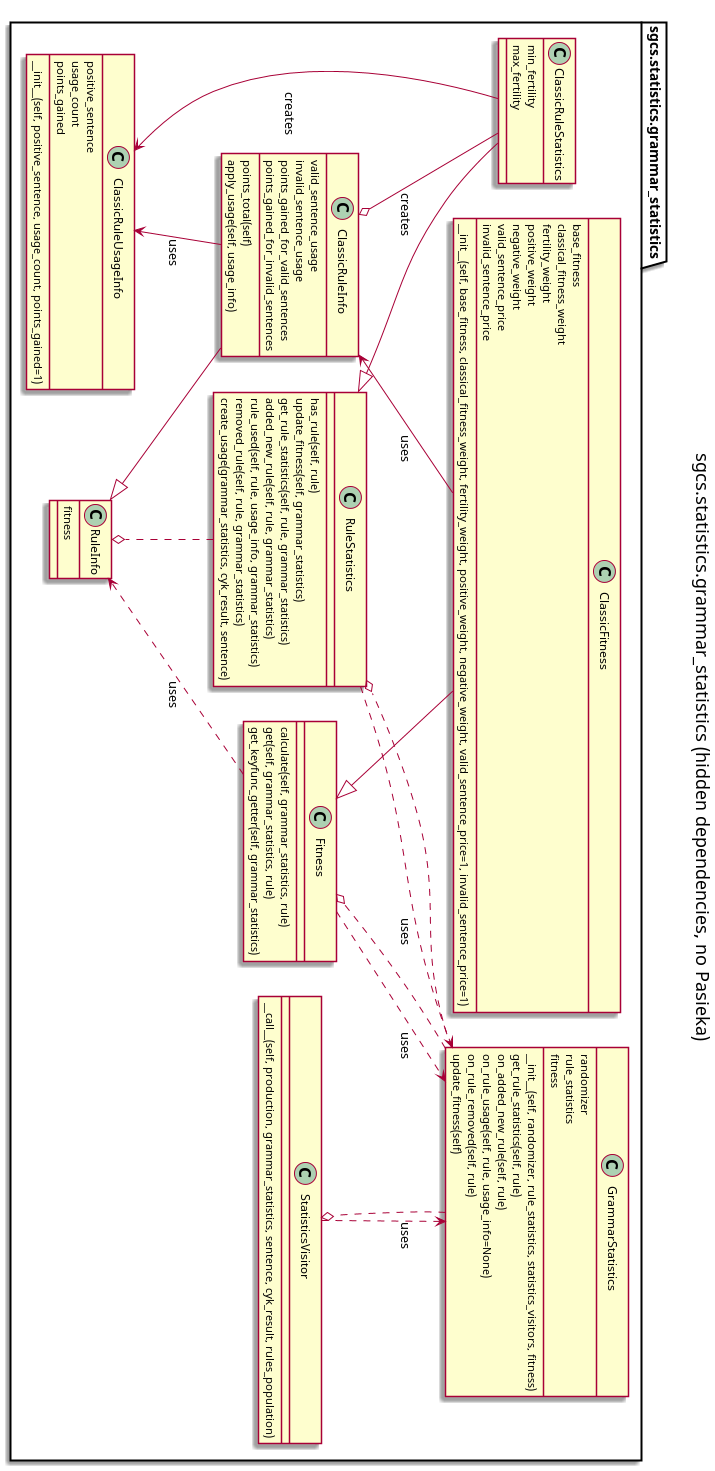
AddingRuleWithElitismStrategy

Strategia wykorzystująca mechanizm ścisku oraz elityzm do dodawania reguł.

generate\_elite(self, adding\_supervisor, statistics, rule\_population) – funkcja generująca elitę. Nie trzeba jej ręcznie wywoływać – zostanie automatycznie wywołana przez metodę AddingRuleSupervisor.update\_elite\_if\_supported.

sgcs.statistics.grammar\_statistics





Powyższe dwa diagramy dopiero po połączeniu dają całkowity obraz modułu sgcs.statistics.grammar\_statistics. Niestety nawet bez dołączania zewnętrznych zależności modułu jest to obraz wysoce nieczytelny. Dlatego też zdecydowano się na przedstawienie modułu w postaci dwóch diagramów – na jednym ukryto klasy związane z funkcją przystosowania Pasieka, na drugim ukryto klasy związane z klasyczną funkcją przystosowania. Obie grupy są od siebie wzajemnie niezależne, natomiast część wspólna obu diagramów została zdublowana.

Moduł ten zajmuje się obliczaniem wartości funkcji przystosowania.

GrammarStatistics

Główna klasa zajmująca się obsługą parametru fitness. Większość jej funkcji oddelegowuje swoją logikę do obiektu rule\_statistics, co czyni ją praktycznie fasadą tejże klasy.

randomizer – uchwyt obiektu typu Randomizer.

rule\_statistics – instancja obiektu dziedziczącego po abstrakcyjnym RuleStatistics. Klasa ta zajmuje się obsługą większości logiki GrammarStatistics, pozostawiając mu jedynie wspólną dla wszystkich funkcji fitness logikę.

fitness – instancja obiektu dziedziczącego po klasie Fitness.

statistics\_visitors – lista instancji obiektu StatisticsVisitor. Jest on na tyle generyczny, że bez względu na ilość i rodzaj obsługiwanych funkcji fitness nie ma potrzeby umieszczania na niej więcej niż jednego obiektu tego typu – inne obiekty mogłyby być przydatne przy zastosowaniu naprawdę niestandardowych funkcji przystosowania (patrz opis klasy StatisticsVisitor).

default(randomizer, configuration) – funkcja tworząca instancję GrammarStatistics przygotowaną pod względem liczenia funkcji klasycznej.

sgcs\_variant(randomizer, configuration) – funkcja tworząca instancję GrammarStatistics przygotowaną pod względem liczenia funkcji Pasieka.

get\_rule\_statistics(self, rule) – metoda zwracająca statystyki reguły rule (postać zależna od implementacji statystyk).

on\_added\_new\_rule(self, rule) – metoda obsługująca utworzenie nowej reguły.

on\_rule\_usage(self, rule\_usage, usage\_info=None) – metoda wywoływana, gdy reguła zostanie użyta. Można dołączyć opcjonalny parametr UsageInfo (umożliwiający zbieranie dodatkowych informacji przez bardziej skomplikowane funkcje fitness jak na przykład funkcja klasyczna).

on\_rule\_removed(self, rule) – metoda obsługująca usunięcie reguły.

update\_fitness(self) – przeliczenie funkcji fitness na podstawie obecnie zebranych statystyk.

RuleStatistics

Klasa abstrakcyjna, której konkretyzacje realizują większość logiki GrammarStatistics. Posiada ona dwie dodatkowe metody:

has\_rule(self, rule) – zwraca True, jeżeli reguła rule już istnieje.

create\_usage(grammar\_statistics, cyk\_result, sentence) – tworzy obiekt UsageInfo, jeżeli jest on wspierany przez funkcję fitness. Obiekty te nie mają określonego z góry interfejsu, jako że są to obiekty służące właśnie do przekazywania danych typowych wyłącznie dla poszczególnych funkcji fitness i to one są ich producentem i konsumentem. Jeżeli nie ma potrzeby wykorzystania takiego obiektu, zwrócona zostanie wartość None.

RuleInfo

Klasa przechowująca informacje niezbędne do wyliczenia wartości fitness (oraz obecną wartość tegoż parametru).

StatisticsVisitor

Obiekt ten powstał w celu bycia wykorzystanym przez algorytm traceback. W chwili kiedy zostanie wywołany jak funkcja, w odpowiedni sposób zapisze informację o wykorzystaniu reguły. Ukrycie logiki liczenia funkcji fitness czyni go wysoce uniwersalnym, dzięki czemu wydaje się, że raczej nie będzie konieczności dodawania kolejnych wizytatorów (przy założeniu, że funkcje przystosowania nie zaczną wymagać wysoce specyficznych danych).

Fitness

Abstrakcyjna klasa reprezentująca sam wzór na funkcję przystosowania.

calculate(self, grammar\_statistics, rule) – abstrakcyjna metoda licząca wartość funkcji przystosowania.

get(self, grammar\_statistics, rule) – metoda zwracająca wartość funkcji przystosowania. Jeżeli wartość ta nie została jeszcze nigdy policzona, wówczas wywołanie tej metody spowoduje te obliczenia.

get\_keyfunc\_getter(self, grammar\_statistics) – metoda zwracająca delegata funkcji get, który jest wygodny w zastosowaniu w wyrażeniach funkcyjnych (jak na przykład polecenie map() ).

Classic\_\_\_

Funkcja klasyczna wykorzystuje szereg różnorodnych klas w celu wyliczenia wartości fitness. Są to:

* ClassicRuleStatistics;
* ClassicRuleInfo;
* ClassicRuleUsageInfo;
* ClassicFitness.

Klasy te nie wymagają większego komentarza – są ukonkretnionymi klasami omówionych klas abstrakcyjnych, mających za zadanie obsłużyć liczenie klasycznej funkcji fitness. Warto tutaj jedynie zauważyć fakt istnienia klasy ClassicRuleUsageInfo – w klasie tej są zbierane informacje jak punktów zdobytych przez regułę, czy zdanie było pozytywne oraz pozostałe informacje wymagane przez klasyczną funkcję przystosowania.

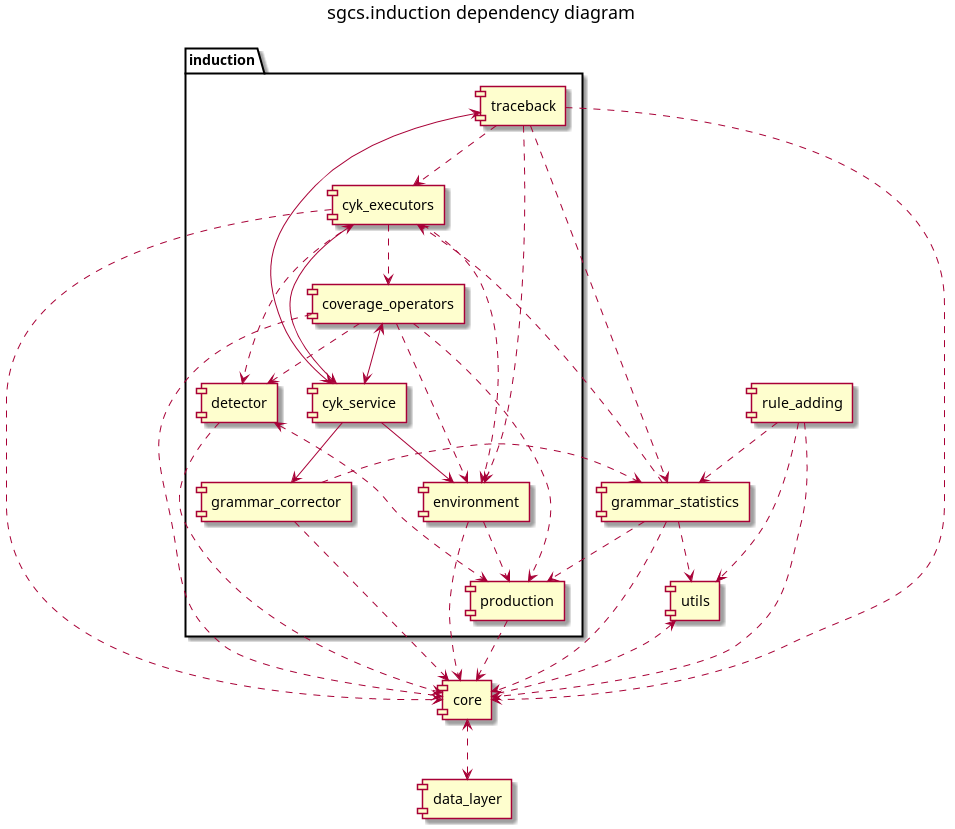
Pasieka\_\_\_

Funkcja Pasieka również posiada swój zestaw klas niezbędnych do wyliczenia wartości przystosowania. Są to:

* PasiekaRuleStatistics;
* PasiekaFitness;
* PasiekaRuleInfo;
* PasiekaLeftSideInfo.

PasiekaLeftSideInfo gromadzi informacje na temat wykorzystania reguł o Symbolu lewej strony produkcji, trzymanie tej informacji jest niezbędne i nie ma sensu jej duplikować w każdej regule posiadającej ten symbol w lewej części produkcji. Pokazuje to równocześnie uniwersalność modułu, który może ukryć w ten sposób pod omówionymi wcześniej klasami spore fragmenty nietrywialnej mechaniki, nie wymagając przy tym samym modyfikacji interfejsu.

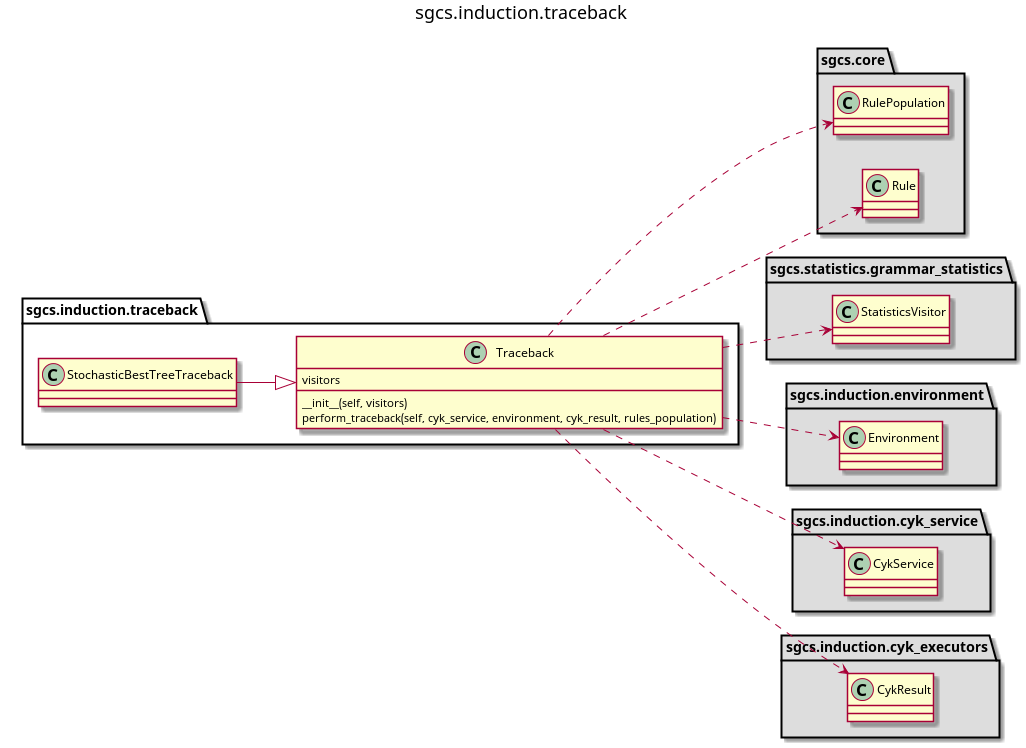
Właściwy algorytm – część indukcyjna

Powyżej znajduje się diagram zależności modułów indukcyjnej części algorytmu genetycznego. Jest to oczywiście najważniejsza i najbardziej złożona część algorytmu. Wszystkie moduły związane z indukcją gramatyki umieszczono w przestrzeni nazw sgcs.induction.

sgcs.induction.cyk\_service

Moduł ten zawiera dwie klasy – CykService oraz dziedziczącą po niej StochasticCykService. Klasy te gromadzą obiekty z całego modułu indukcji i składają je w jedną, dostępną dla wszystkich składowych strukturę, w zależności od wykorzystanej klasy struktura ta jest nastawiona na indukcję gramatyk stochastycznych lub klasycznych. Znajduje się tu też główna pętla indukcji gramatyki.

sgcs.induction.traceback



Moduł ten zawiera klasy implementujące rożne warianty algorytmu traceback. Wykorzystują one wzorzec wizytatora – traceback zapewnia jedynie spójny interfejs do poruszania się po drzewie rozbioru zdania, zaś sam wizytator decyduje w jaki sposób wykorzysta zebrane podczas wędrówki dane. Pozwala to na separację systemu nagradzania odwiedzonych węzłów od samej metody poruszania się po drzewie rozbioru.

Traceback

Standardowa implementacja algorytmu traceback wykorzystywana przez algorytm GCS. Algorytm ten gorliwie przechodzi po wszystkich możliwych drzewach rozkładu zdania.

visitors – lista wizytatorów. Wizytatorzy muszą mieć następującą sygnaturę:

1. Aktualnie odwiedzana produkcja.
2. Obiekt GrammarStatistics.
3. Obiekt Sentence (reprezentacja całego zdania).
4. Obiekt CykResult.
5. Obiekt RulePopulation.

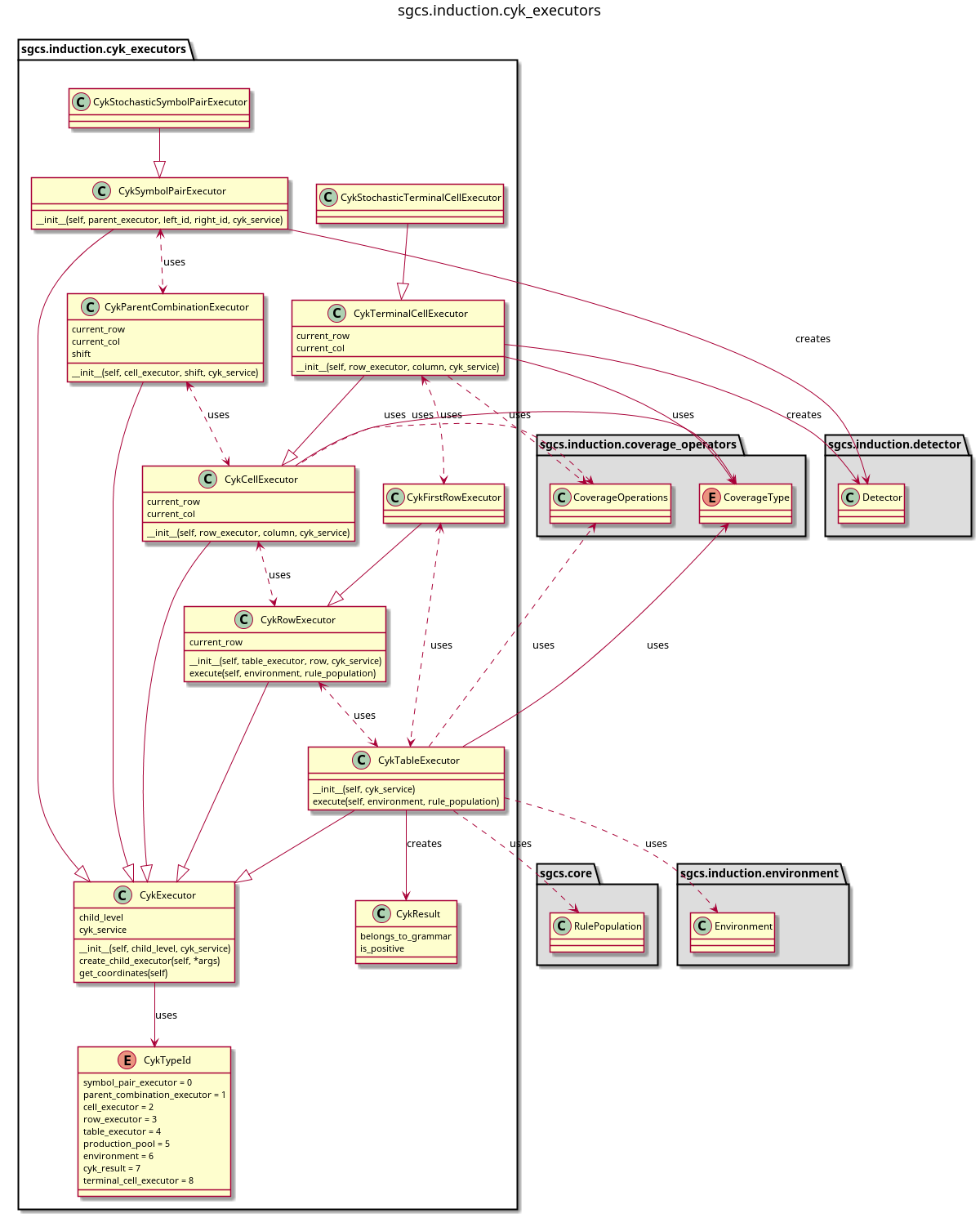
Kolejność tych parametrów ma znaczenie.

perform\_traceback(self, cyk\_service, environment, cyk\_result, rules\_population) – metoda powodująca wywołanie algorytmu traceback. Obiekt environment stanowi reprezentację tablicy CYK, cyk\_result zawiera między innymi werdykt przydzielony mu przez algorytm oraz informację o tym, czy zdanie w rzeczywistości należy do gramatyki, czy też nie.

StochasticBestTreeTraceback

Implementacja algorytmu traceback wykorzystywana przez algorytmy stochastyczne. Algorytm ten przechodzi jedynie po najbardziej prawdopodobnym drzewie rozkładu zdania. Dziedziczy po klasie Traceback i nie rozszerza istniejącego interfejsu.

sgcs.induction.cyk\_executors

Moduł sgcs.induction.cyk\_executors jest rozbitą na czynniki pierwsze pętlą algorytmu CYK. Kolejni wykonawcy skupiają się na coraz głębszych elementach algorytmu CYK, wszyscy są tworzeni przez odpowiednio przygotowaną wcześniej fabrykę. Takie rozdrobnienie tego kluczowego miejsca algorytmu umożliwia łatwe modyfikowanie funkcjonowania procesu indukcji.

CykTypeId

Wyliczenie przedstawiające identyfikatory typów klas, z jakimi mamy do czynienia w module. Jest ono wykorzystywane przez fabrykę obiektów, w efekcie czego żadna z klas powiązanych z poniższymi wartościami nie jest tworzona w inny sposób niż przy pomocy fabryki. Domyślnie moduł ten promuje następujący podział klas (który może zostać rozszerzony, jeżeli zmienimy zachowanie fabryki:

1. table\_executor – klasa wykorzystywana na najwyższym poziomie, odpowiada za wypełnianie całej tablicy CYK.
2. row\_executor – klasa odpowiadająca za uzupełnienie całego wiersza tablicy CYK.
3. cell\_executor – klasa odpowiedzialna za uzupełnienie pojedynczej komórki tabeli CYK.
4. parent\_combination\_executor – klasa odpowiedzialna za wpisanie do komórki tabeli CYK reguł powstałych z reguł z pojedynczej kombinacji komórek – rodziców.
5. symbol\_pair\_executor – najniższy poziom uzupełniania tabeli CYK, generująca reguły powstałe z pojedynczej pary symboli.
6. environment – klasa zarządzająca zawartością tabeli CYK.
7. production\_pool – klasa zarządzająca zawartością pojedynczej komórki tabeli CYK.
8. cyk\_result – klasa niosąca informacje o rezultacie parsowania (oraz o prawdziwej przynależności zdania).

Klasy oznaczone identyfikatorami typów 1-5 są klasami wykonawców (uporządkowanych malejąco względem ich odpowiedzialności). Klasa bezpośrednio kontrolowana przez wykonawcę nadrzędnego będzie od tej pory nazywana „podwykonawcą”.

CykResult

Prosta klasa przechowująca wynik parsowania zdania przy pomocy algorytmu CYK oraz prawdziwą przynależność zdania.

belongs\_to\_grammar – zmienna boolowska ustalająca czy zdanie rzeczywiście należy do gramatyki.

is\_positive – zmienna boolowska stwierdzająca czy CYK zdołał sparsować zdanie.

CykExecutor

Abstrakcyjna klasa dostarczająca trochę podstawowej logiki klasom wykonawców. Zawiera uchwyt do instancji CykService, jest w stanie stworzyć podwykonawcę oraz podać swoje koordynaty w tabeli CYK.

child\_level – identyfikator podwykonawcy.

cyk\_service – uchwyt instancji CykService.

create\_child\_executor(self, \*args) – metoda zwracająca obiekt podwykonawcy. Musimy wiedzieć dokładnie jakie parametry są wymagane przez konstruktor.

get\_coordinates(self) – metoda zwracająca koordynaty wykonawcy w tabeli CYK.

CykTableExecutor

Klasa najwyższego poziomu algorytmu CYK. Jest w stanie ustalić, czy zdane zostało sparsowane przez tabelę CYK, może uruchomić operatory pokrycia zarejstrowane na CoverageType.no\_starting\_symbol oraz zlecić podwykonawcom wypełnianie wierszy tabeli CYK. Jej podwykonawcy to CykFirstRowExecutor (dla pierwszego wiersza) oraz CykRowExecutor (dla kolejnych).

\_\_init\_\_(self, cyk\_service) – konstruktor.

execute(self, environment, rule\_population) – metoda wywołująca algorytm CYK.

CykRowExecutor

Podwykonawca klasy CykTableExecutor. Przeprowadza algorytm CYK na poziomie pojedynczego wiersza, przeznaczony do wszystkich wierszy powyżej pierwszego). Jego podwykonawcą jest CykCellExecutor.

current\_row – zwraca numer obecnego wiersza.

\_\_init\_\_(self, table\_executor, row, cyk\_service) – kontruktor. Przekazywanie siebie podwykonawcy często ma miejsce w klasach wykonawców.

CykFirstRowExecutor

Klasa uzupełniająca pierwszy wiersz tabeli CYK. Dziedziczy po CykRowExecutor. Jej podwykonawcą jest CykTerminalStateExecutor lub CykStochasticTerminalCellExecutor (w przypadku algorytmu stochastycznego).

CykCellExecutor

Klasa uzupełniająca pojedynczą komórkę (powyżej pierwszego wiersza). Może wywołać operatory pokrycia zarejestrowane na CoverageType.no\_effector\_type oraz zlecić podwykonawcom wpisywanie przodukcji powstałych z konkretnych par komórek – rodziców. Jej podwykonawcą jest CykParentCombinationExecutor.

current\_col – zwraca numer obecnej kolumny.

CykCellTerminalExecutor

Klasa uzupełniająca pojedynczą komórkę pierwszego wiersza tabeli CYK. W przypadku nieznalezienia żadnych produkcji może wywołać operatory pokrycia zarejestrowane na typ pokrycia CoverageType.unknown\_terminal\_symbol. Nie posiada podwykonawcy.

CykStochasticTerminalCellExecutor

Klasa dziedzicząca po CykCellTerminalExecutor, uzupełniająca dodatkowo prawdopodobieństwa produkcji.

CykParentCombinationExecutor

Klasa tworząca wszystkie produkcje powstałe z jednej pary komórek – rodziców. Jej podwykonawcą jest CykSymbolPairExecutor lub CykStochasticSymbolPairExecutor w przypadku gramatyk stochastycznych.

shift – koordynat przesunięcia w tabeli CYK – określa jednoznacznie koordynaty komórek rodziców.

CykSymbolPairExecutor

Klasa tworząca produkcje powstałe z pojedynczej kombinacji symboli. Nie ma podwykonawcy.

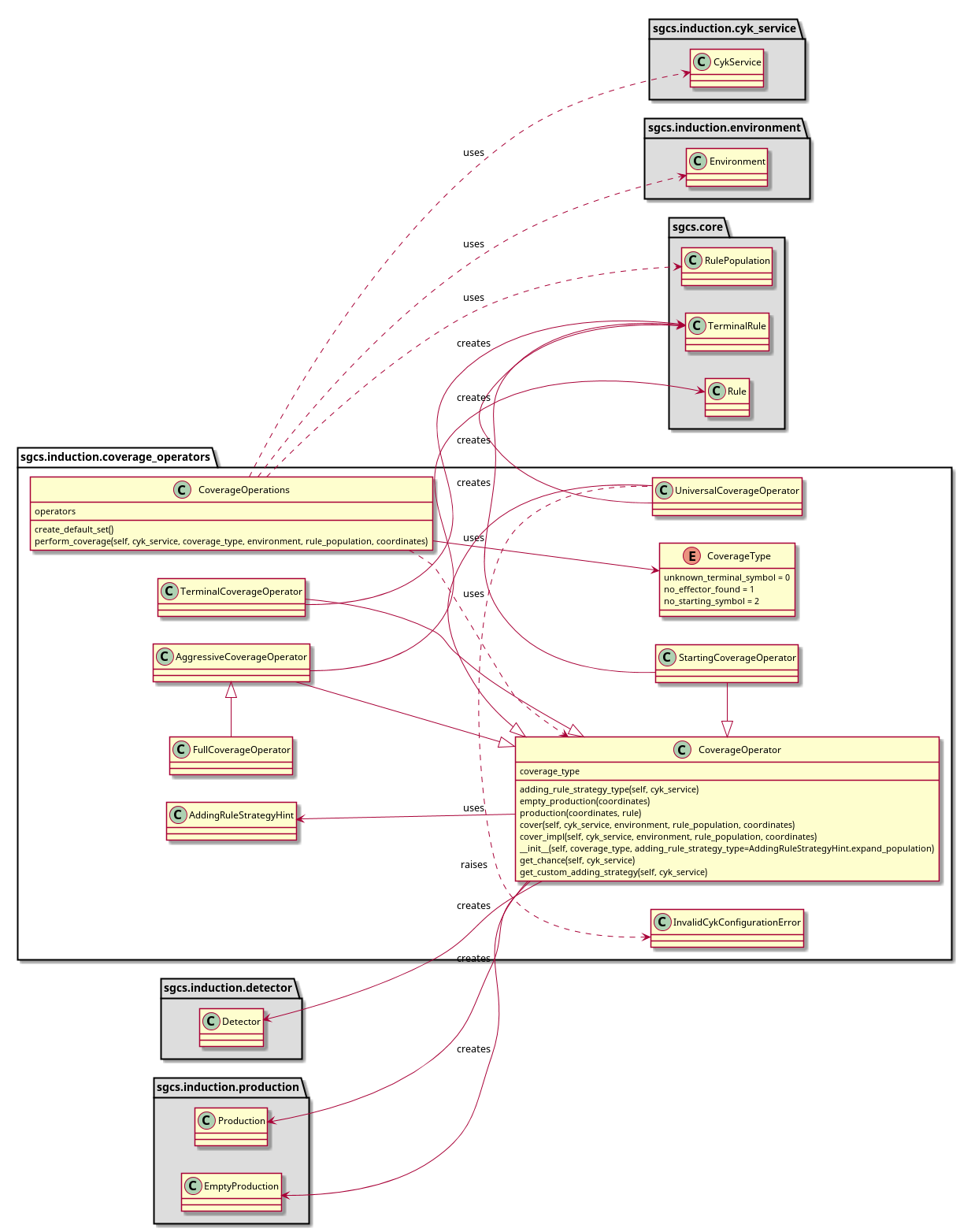
left\_id – koordynat lewego symbolu – jednoznacznie określa symbol z lewej komórki odpowiedzialny za tworzone produkcje.

right\_id – koordynat prawego symbolu – jednoznacznie określa symbol z prawej komórki odpowiedzialny za tworzone produkcje.

CykStochasticSymbolPairExecutor

Klasa rozszerzająca CykSymbolPairExecutor o aktualizację prawdopodobieństw produkcji.

sgcs.induction.coverage\_operators

Moduł sgcs.induction.coverage\_operators dostarcza implementację standardowych operatorów pokrycia. Zostały one oddzielone od modułu sgcs.induction.cyk\_executors przy pomocy identyfikatorów typu pokrycia CoverageType. Kiedy w głównej pętli algorytmu następuje potrzeba uruchomienia operatorów pokrycia, wówczas jest wywoływana metoda CoverageOperations.perform\_coverage z odpowiednim identyfikatorem typu pokrycia, na który zareagują wyłącznie operatory pokrycia zarejestrowane na ten typ. Na przykład przy natrafieniu na nieznany symbol nieterminalny jest wykorzystywany identyfikator CoverageType.unknown\_terminal\_symbol, na który reagują odpowiednio operatory pokrycia terminalnego oraz uniwersalnego. Umożliwia to łatwe rozszerzanie algorytmu o nowe operatory pokrycia bez potrzeby wprowadzania modyfikacji do głównej pętli algorytmu, zmiany dotyczące operatorów pokrycia dokonuje się wyłącznie w module operatorów pokrycia. Po stworzeniu nowych reguł są one dodawane do populacji reguł z pomocą klasy AddingRuleSupervisor. Operatory pokrycia posiadają domyślne strategie dodawania reguł (na przykład operator pokrycia agresywnego zawsze dodaje reguły ze ściskiem), aczkolwiek można nadpisać to zachowanie przy pomocy odpowiedniego ustawienia węzła konfiguracyjnego tego operatora.

W celu ujednolicenia modelu wszystkie operatory posługują się prawdopodobieństwem – oznacza to, że jest możliwe uruchomienie na przykład operatora pokrycia uniwersalnego z pewnym prawdopodobieństwem (zamiast dyskretnie zawsze lub nigdy).

CoverageType

Typ wyliczeniowy przechowujący dostępne rodzaje operatorów pokrycia – są to obecnie unknown\_terminal\_symbol, no\_effector\_found oraz no\_starting\_symbol. CoverageType najłatwiej zrozumieć jako powód, dla którego może być konieczne uruchomienie operatora pokrycia – uruchamiamy je gdy natrafimy na nieznany symbol gramatyki, nie udało się wpisać nic do jednej z komórek tabeli CYK lub komórka nie zawiera symbolu startowego gramatyki.

CoverageOperations

Klasa agregująca operatory pokrycia. Jest zdolna do wygenerowania ich podstawowego zestawu oraz uruchomienia operatorów pokrycia danego typu, kiedy zaistnieje taka potrzeba.

operators – lista operatorów pokrycia

create\_default\_set() - funkcja zwracająca domyślną listę (czyli zawierającą wszystkie dostarczone wraz z tym modułem) operatorów pokrycia.

perform\_coverage(self, cyk\_service, coverage\_type, environment, rule\_population, coordinates) – funkcja ta znajduje wszystkie operatory zarejestrowane na coverage\_type i uruchamia dany operator (który ostatecznie zostanie uruchomiony lub też nie, zależnie od prawdopodobieństwa oraz innych dodatkowych warunków wymaganych przez operator). Jeżeli udało się wygenerować nową produkcję, to jest ona dodawana do tabeli CYK, zaś nowa reguła z nią związana dodana z pomocą AddingRuleSupervisor do populacji reguł. Metoda próbuje zachłannie wszystkich operatorów pokrycia.

CoverageOperator

Klasa abstrakcyjna dostarczająca części wspólnej logiki operatorom pokrycia.

coverage\_type – powiązany z operatorem identyfikator typu pokrycia.

\_\_init\_\_(self, coverage\_type, adding\_rule\_strategy\_type = AddingRuleStrategyHint.expand\_population) – konstruktor, pobierający typ pokrycia operatora oraz domyślną strategię dodawania nowych reguł.

adding\_rule\_strategy\_type(self, cyk\_service) – zwraca domyślną dla operatora strategia dodawania nowych reguł, chyba że została ona nadpisana w konfiguracji własną strategią.

empty\_production(coordinates) – Funkcja pomocnicza zwracająca obiekt pustej produkcji dla koordynatów coordinates.

production(coordinates, rule) – Funkcja pomocnicza zwracająca obiekt produkcji powiązany z regułą rule i koordynatami coordinates.

cover(self, cyk\_service, environment, rule\_population, coordinates) – metoda z prawdopodobieństwem wywołania operatora pokrycia zwraca rezultat metody cover\_impl (której zachowanie jest zależne od implementacji konkretnego operatora pokrycia, ale spodziewa się od niej zwrócenia nowej produkcji, lub też nie).

cover\_impl(self, cyk\_service, environment, rule\_population, coordinates) – abstrakcyjna metoda, która jeżeli warunki uruchomienia operatora są spełnione (na przykład operator pokrycia startowego próbuje pokryć zdanie należące do gramatyki) wykona odpowiednią operację pokrycia.

get\_chance(self, cyk\_service) – abstrakcyjna metoda zwracająca prawdopodobieństwo uruchomienia operatora pokrycia (zazwyczaj jest to wartość zapisana w którymś z węzłów konfiguracji).

get\_custom\_adding\_strategy(self, cyk\_service) – abstrakcyjna metoda zwracająca podaną własną strategię dodawania reguł (zazwyczaj jest to wartość zapisana w którymś z węzłów konfiguracji).

TerminalCoverageOperator

Klasa reprezentująca operator pokrycia terminalnego. Typ pokrycia tego operatora to CoverageType.unknown\_terminal\_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk\_service.configuration.coverage.operators.terminal.adding\_hint, zaś prawdopodobieństwo w cyk\_service.configuration.coverage.operators.terminal.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.expand\_population.

UniversalCoverageOperator

Klasa reprezentująca operator pokrycia uniwersalnego. Typ pokrycia tego operatora to CoverageType.unknown\_terminal\_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk\_service.configuration.coverage.operators.universal.adding\_hint, zaś prawdopodobieństwo w cyk\_service.configuration.coverage.operators.universal.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.expand\_population. Korzystanie z tego operatora wymaga sprecyzowania symbolu uniwersalnego w populacji reguł – jeżeli symbol ten nie jest zdefiniowany, przy próbie użycia operatora zostanie zgłoszony wyjątek InvalidCykConfigurationError (gdyż oznacza to sprzeczność w samej konfiguracji).

StartingCoverageOperator

Klasa reprezentująca operator pokrycia startowego. Typ pokrycia tego operatora to CoverageType.no\_starting\_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk\_service.configuration.coverage.operators.starting.adding\_hint, zaś prawdopodobieństwo w cyk\_service.configuration.coverage.operators.starting.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control\_population\_size. Operator ten posiada dodatkowy warunek uruchomienia – można go wykorzystać wyłącznie w zdaniu pozytywnym.

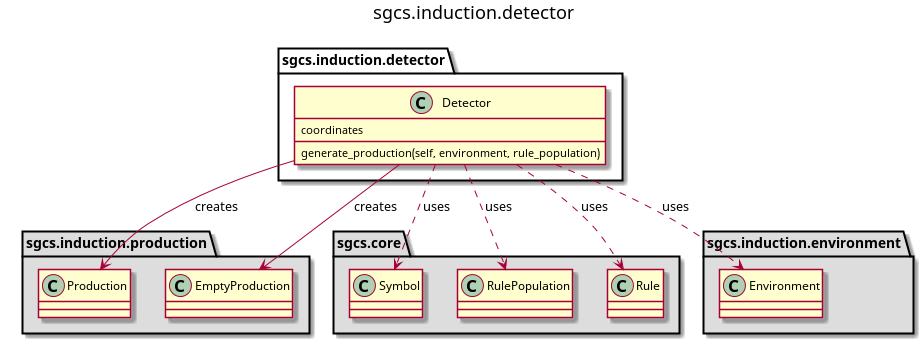
AggressiveCoverageOperator

Klasa reprezentująca operator pokrycia agresywnego. Typ pokrycia tego operatora to CoverageType.no\_effector\_found. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk\_service.configuration.coverage.operators.aggressive.adding\_hint, zaś prawdopodobieństwo w cyk\_service.configuration.coverage.operators.aggressive.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control\_population\_size.

FullCoverageOperator

Klasa reprezentująca operator pokrycia pełnego. Dziedziczy po AggressiveCoverageOperator. Typ pokrycia tego operatora to CoverageType.no\_starting\_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk\_service.configuration.coverage.operators.full.adding\_hint, zaś prawdopodobieństwo w cyk\_service.configuration.coverage.operators.full.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control\_population\_size.

sgcs.induction.detector

Moduł zawierający klasę Detector.

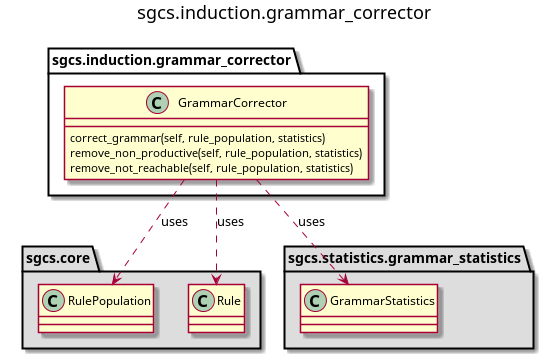
Detector

Wygodny wrapper na koordynaty produkcji. Zawiera również logikę generowania produkcji na podstawie symboli jednoznacznie wskazywanych przez krotkę coordinates w tabeli CYK.

coordinates – jednoznaczne określenie położenia symboli wskazywanych przez detektor. Dysponując tablicą CYK można powiedzieć, których dwóch symboli dotyczy.

generate\_productions(self, environment, rule\_population) – metoda znajdująca wszystkie możliwe produkcje dla danego detektora.

sgcs.induction.grammar\_corrector

Moduł zawierający klasy odpowiedzialne za korekcję gramatyki. Obecnie wspiera usuwanie reguł nieproduktywnych oraz nieosiągalnych; usuwanie reguł redundantnych nie jest potrzebne, gdyż budowa populacji reguł chroni przed wystąpieniem powtarzających się reguł.

GrammarCorrector

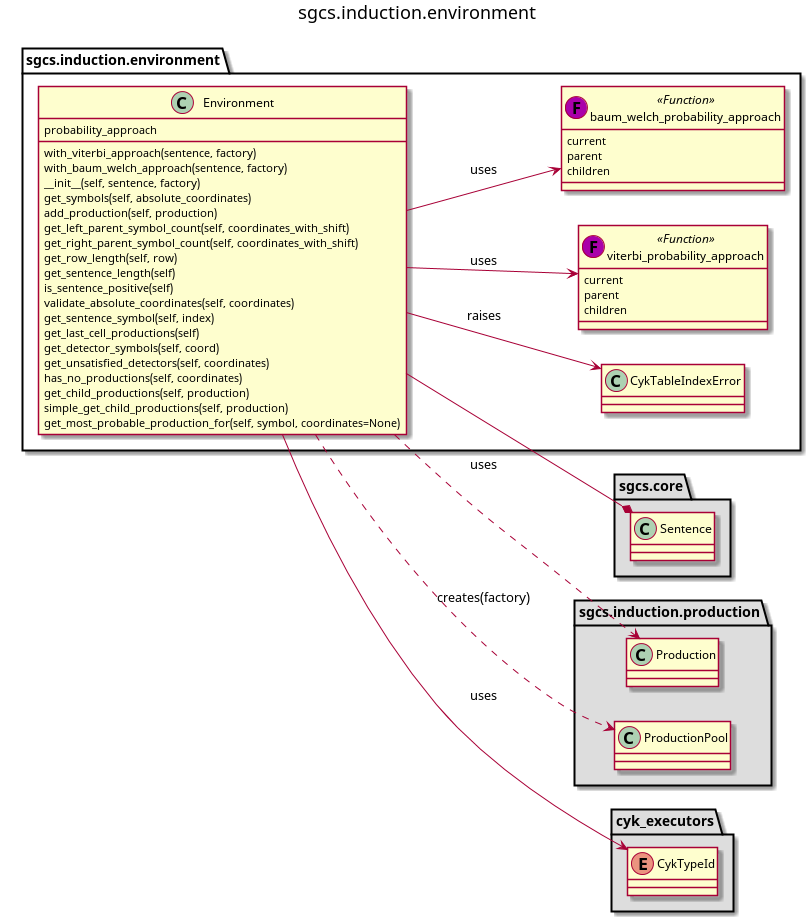
Klasa odpowiedzialna za korekcję gramatyki.

correct\_grammar(self, rule\_population, statistics) – metoda wywołująca remove\_non\_productive, a następnie remove\_not\_reachable.

remove\_non\_productive(self, rule\_population, statistics) – metoda usuwająca reguły, z których nie da się wyprowadzić reguł terminalnych.

remove\_not\_reachable(self, rule\_population, statistics) – metoda usuwająca reguły, których nie da się wyprowadzić z symbolu startowego gramatyki.

sgcs.induction.environment

Moduł ten zawiera klasy ściśle powiązane z obsługą tabeli CYK.

Environment

Klasa reprezntująca tabelę CYK, zdolna do obsługi zarówno danych stochastycznych, jak i klasycznych.

probability\_approach – delegat sposobu liczenia prawdopodobieństw kolejnych komórek. Dostępni obecnie delegaci to viterbi\_probablity\_approach (liczenie prawdopodobieństw metodą Viterbiego) oraz baum\_welch\_probability\_approach (liczenie prawdopodobieństw metodą Bauma-Welcha). Jeżeli klasa nie jest wykorzystywana przez algorytm stochastyczny, wówczas pole to przybiera wartość None.

with\_viterbi\_approach(sentence, factory) – funkcja tworząca instancję Environment z podejściem Viterbiego.

with\_baum\_welch\_approach(sentence, factory) – funkcja tworząca instancję Environment z podejściem Bauma-Welcha.

\_\_init\_\_(self, sentence, factory) – konstruktor pobierający zdanie, na podstawie którego zostanie wygenerowana tabela CYK oraz fabrykę factory, z której będzie możliwe uzyskanie instancji obiektów ProductionPool (reprezentujących pojedyncze komórki tabeli).

get\_symbols(self, absolute\_coordinates) – metoda zwracająca wszystkie efektory z komórki położonej na pozycji absolute\_coordinates.

add\_production(self, production) – metoda dodająca nową produkcję (wraz z ewentualnym przeliczeniem prawdopodobieństw).

get\_left\_parent\_symbol\_count(self, coordinates\_with\_shift) – metoda zwracająca ilość efektorów w komórce lewego rodzica.

get\_right\_parent\_symbol\_count(self, coordinates\_with\_shift) – metoda zwracająca ilość efektorów w komórce prawego rodzica.

get\_row\_length(self, row) – metoda zwracająca rzeczywistą długość wiersza tabeli CYK (tabela CYK jest tabelą trójkątną).

get\_sentence\_length(self) – metoda zwracająca długość zdania parsowanego w tabeli.

is\_sentence\_positive(self) – metoda zwracająca True, jeżeli zdanie należy do gramatyki.

get\_sentence\_symbol(self, i) – metoda zwracająca i-ty Symbol zdania.

get\_last\_cell\_productions(self) – metoda zwracająca produkcje ostatniej komórki tabeli CYK.

get\_detector\_symbols(self, coord) – metoda zwracająca symbole rodziców w postaci krotki kolekcji.

get\_unsatisfied\_detectors(self, coordinates) – metoda zwracająca detektory, dla których nie dopasowano żadnych reguł (przydatne przy generowaniu nowych).

has\_no\_productions(self, coordinates) – metoda zwraca True, jeżeli w danej komórce tabeli CYK nie zapisano żadnych produkcji.

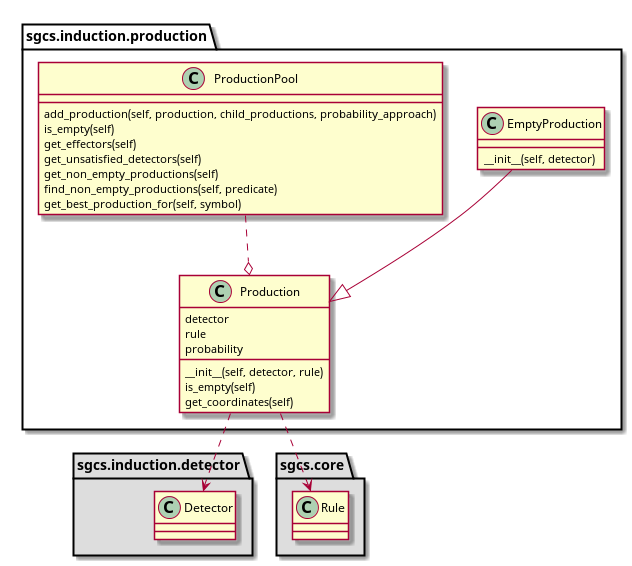
simple\_get\_child\_productions(self, production) – metoda zwracająca produkcje produkowane przez produkcję production (wykorzystywane między innymi przy przemierzaniu drzewa algorytmem Traceback). Jeżeli mamy do czynienia z algorytmem stochastycznym, prawdopodobieństwa są również zwracane.

get\_most\_probable\_production\_for(self, symbol, coordinates=None) – Zwraca z komórki CYK o koordynatach coordinates produkcję o rodzicu symbol taką, że jej prawdopodobieństwo jest największe. Jeżeli pominie się pole coordinates, wówczas metoda przyjmie koordynaty ostatniej komórki tabeli CYK.

CykTableIndexError

Wyjątek zgłaszany w przypadku wykroczenia indeksem poza granice tabeli CYK.

sgcs.induction.production

Moduł służący do zarządzania produkcjami w systemie. Przez produkcję należy tutaj rozumieć obiekt wiążący detektor z regułą populacji.

Production

Produkcja jest obiektem relacji klasy Detector oraz Rule. Oznacza to, że daje on wiedzę, skąd dana reguła została uzyskana oraz w jakim miejscu tabeli CYK się znajduje. Są to dane niezmiernie przydatne, ułatwiają między innymi wykorzystanie algorytmu traceback, który musi tylko poruszać się po zapamiętanej wcześniej sieci współrzędnych.

detector – pole przechowujące obiekt klasy Detector.

rule – pole przechowujące obiekt klasy Rule.

probability – obiekt przechowujący wartość prawdopodobieństwa danej produkcji (jest to prawdopodobieństwo całego drzewa generowanego przez daną produkcję, nie zaś prawdopodobieństwo pojedynczej produkcji). To pole jest opcjonalne – jeżeli klasa jest wykorzystywana poza algorytmem stochastycznym, wówczas jego wartość jest ignorowana.

is\_empty(self) – metoda zwracająca True, jeżeli produkcja jest „pusta”. Pusta, czyli nie generuje żadnej reguły (rule = None).

get\_coordinates(self) – zwraca współrzędne przechowywane przez detector.

EmptyProduction

Produkcja nieposiadającą reguły. Klasa ta w zasadzie nie rozszerza klasy Production, a stanowi jedynie ekspresywny alias na Production(detector, None).

ProductionPool

Pula produkcji, wykorzystywana między innymi do reprezentacji zawartości pojedynczej komórki w tabeli CYK.

add\_production(self, production, child\_productions, probability\_approach) – metoda dodająca produkcję do puli. Wykonywane są przy okazji różne czynności pomocnicze jak zapisywanie efektorów, przeliczanie prawdopodobieństw itp.

is\_empty(self) – zwraca True, jeżeli pula nie zawiera niepustych produkcji, False w przeciwnym przypadku.

get\_effectors(self) – zwraca efektory zarejestrowane w puli.

get\_unsatisfied\_detectors(self) – zwraca detektory pustych populacji (przydatne przy generowaniu nowych reguł).

get\_non\_empty\_productions(self) – zwraca wszystkie niepuste produkcje.

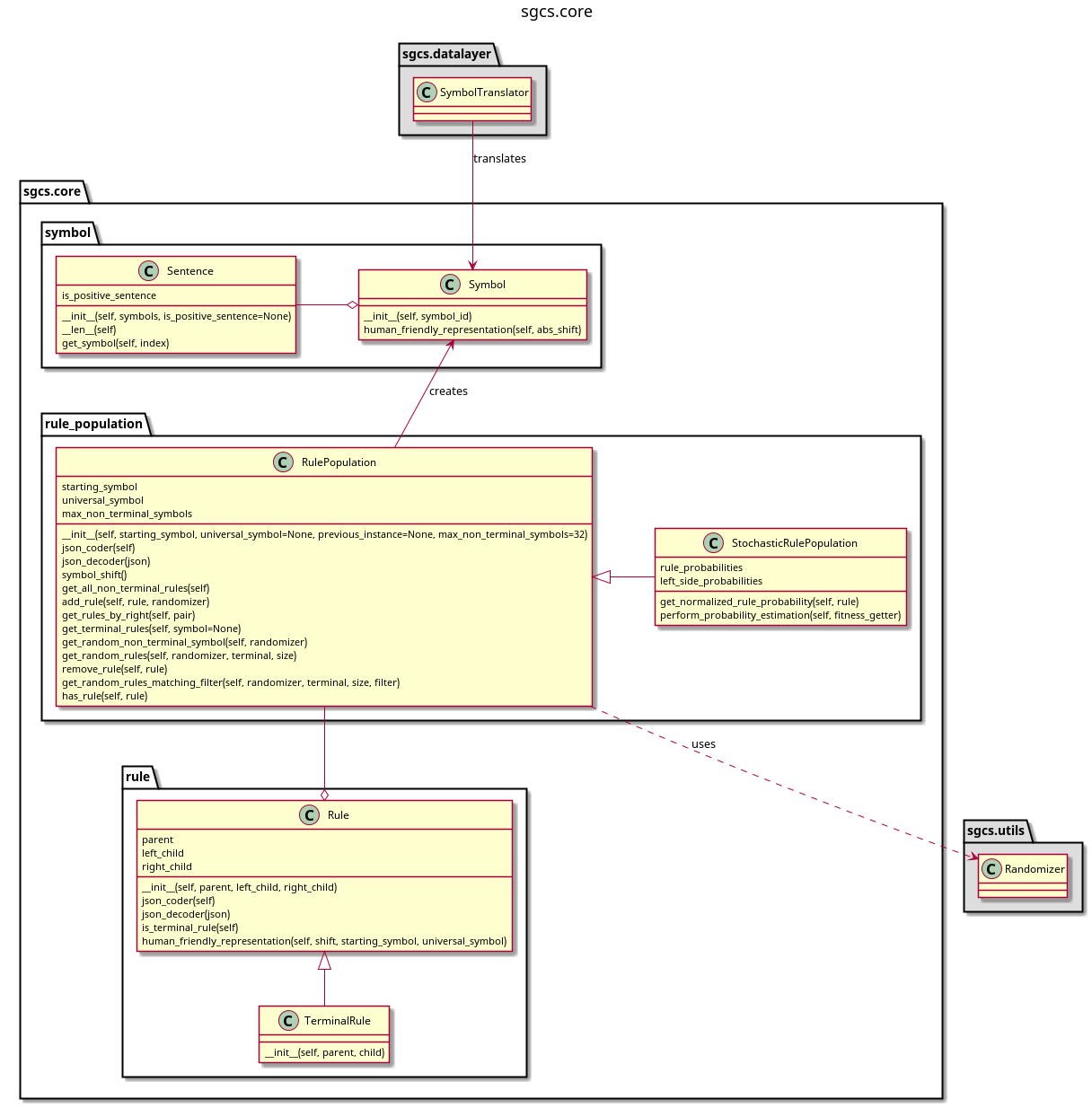
find\_non\_empty\_productions(self, predicate) – zwraca wszystkie niepuste produkcje spełniające predykat predicate.

get\_best\_production\_for(self, symbol) – zwraca najbardziej prawdopodobną produkcję dla efektora symbol.

Warstwa rdzenia

Warstwa ta jest sercem algorytmu. Klasy zawarte w jej jedynym module sgcs.core są wykorzystywane w wielu miejscach aplikacji, zazwyczaj reprezentują najbardziej podstawowe pojęcia związane z algorytmem wnioskowania gramatycznego. Są to klasy reprezentujące podstawowe dane pamięci algorytmu.

sgcs.core

RulePopulation

Klasa odpowiedzialna za zarządzanie populacją reguł.

starting\_symbol – reprezentacja symbolu startowego.

universal\_symbol – reprezentacja symbolu uniwersalnego.

max\_non\_terminal\_symbols – stała, określa maksymalną ilość symboli nieterminalnych.

json\_coder(self) – metoda dokonująca tłumaczenia populacji reguł na format .json.

json\_decoder(self, json) – metoda wczytująca stan wewnętrzny obiektu z obiektu w formacie .json.

symbol\_shift – wartość bezwzględna, od której zaczyna się numerowanie indeksów (każdy index taki, że |index| < symbol\_shift jest traktowany jako index zarezerwowany na symbol specjalny).

get\_all\_non\_terminal\_rules(self) – metoda zwracająca wszystkie reguły nieterminalne.

add\_rule(self, rule, randomizer) – dodaje nową regułę do populacji, dokonując losowania prawdopodobieństwa, jeżeli jest to wymagane.

get\_rules\_by\_right(self, pair) – zwraca wszystkie reguły, których prawa strona produkcji = pair.

get\_terminal\_rules(self, symbol=None) – zwraca wszystkie terminale reguły lub terminalne reguły produkujące symbol (jeżeli podany).

get\_random\_non\_terminal\_symbol(self, randomizer) – tworzy losowy symbol.

get\_random\_rules(self, randomizer, terminal, size) – zwraca maksymalnie size losowych reguł.

remove\_rule(self, rule) – usuwa wybraną regułę z populacji.

get\_random\_rules\_matching\_filter(self, randomizer, terminal, size, filter) – metoda zwracająca maksymalnie size losowych reguł, które spełniają warunek filter.

has\_rule(self, rule) – metoda zwraca True, jeżeli reguła rule znajduje się w populacji.

StochasticRulePopulation

Klasa rozszerzająca klasę RulePopulation o obsługę reguł stochastycznych.

rule\_probabilities – mapa reguła – prawdopodobieństwo.

left\_side\_probabilities – mapa ojciec reguły – prawdopodobieństwo symbolu.

get\_normalized\_rule\_probability(self, rule) – metoda zwracająca znormalizowaną wartość prawdopodobieństwa reguły.

perform\_probability\_estimation(self, fitness\_getter) – metoda powodująca wyliczenie nowych prawdopodobieństw dla reguł.

Rule

Klasa stanowiąca reprezentację reguły.

parent – rodzic Rule.

left\_child – lewe dziecko Rule.

right\_child – prawe dziecko Rule.

STARTING\_SYMBOL\_REPR – stała, reprezentacja symbolu startowego.

UNIVERSAL\_SYMBOL\_REPR – stała, reprezentacja symbolu uniwersalnego.

json\_coder(self) – metoda dokonująca tłumaczenia reguły na format .json.

json\_decoder(self, json) – metoda wczytująca stan wewnętrzny obiektu z obiektu w formacie .json.

is\_terminal\_rule(self) – zwraca True, jeżeli reguła jest terminalna.

human\_friendly\_representation(self, shift, starting\_symbol, universal\_symbol) – funkcja tłumacząca obiekt Rule na obiekt łatwy w odczytaniu przez człowieka.

TerminalRule

Cukier syntaktyczny na wyrażenie Rule(A, B, None).

Sentence

Obiekt stanowiący reprezentację zdania.

is\_positive\_sentence – czy zdanie jest tak naprawdę zdaniem pozytywnym.

\_\_init\_\_(self, symbols, is\_positive\_sentence=None) – konstruktor. Pobiera zestaw symboli, z jakich zostanie skonstruowane oraz informację czy podane zdanie w rzeczywistości należy do gramatyki, czy też nie.

\_\_len\_\_(self) – zwraca długość zdania.

get\_symbol(self, index) – Zwraca symbol na pozycji index.

Symbol

Klasa stanowiąca reprezentację pojedynczego symbolu gramatyki. Jeżeli ma się do czynienia z instancją Symbol reprezentującą symbol terminalny, to klasa ta nie posiada informacji wystarczającej do odtworzenia skrywanej wartości – potrzeba do tego dodatkowej wiedzy w postaci obiektu SymbolTranslator z modułu sgcs.datalayer.

human\_friendly\_representation(self, abs\_shift) – Wyświetla ludzką reprezentację symbolu (jak zostało to szczegółowo omówione przy oknie „Population Editor”). abs\_shift jest wartością przesunięcia symboli gramatyki, czyli powinna być tu użyta wartość RulePopulation.rule\_shift. Metody tej powinniśmy używać tylko wyłącznie w przypadku gdy dany Symbol jest symbolem nieterminalnym – nie należy stosować tej metody w przypadku symboli terminalnych oraz specjalnych. Najlepiej zostawić wówczas proces tłumaczenia samej klasie SymbolTranslator, która potrafi prawidłowo wykorzystywać tę metodę.

from\_human\_friendly\_representation(human\_repr, abs\_shift) – Metoda umożliwiająca odtworzenie reprezentacji wewnętrznej symbolu nieterminalnego z ludzkiej reprezentacji. Powstała ona z myślą o byciu wywoływaną przez klasę SymbolTranslator i wszelkie jej wywołania w innym kontekście powinny być wykonywane ze szczególną ostrożnością.

Warstwa danych

Warstwa ta skupia się na prawidłowej serializacji i deserializacji danych zawartych w module sgcs.core. Zawiera tylko jeden moduł – sgcs.datalayer.

sgcs.datalayer

Moduł odpowiedzialny za serializację, deserializację oraz zapisywanie i wczytywanie danych z lokalnego systemu plików.

SymbolTranslator

Klasa zapewniająca dwustronne tłumaczenie symboli z wewnętrznej na ludzką reprezentację i odwrotnie. Jest to klasa o stosunkowo wysokim poziomie abstrakcji, nie wymagając od użytkownika wiedzy na temat sposobu przechowywania danych.

create(path) – funkcja tworząca domyślnie skonfigurowaną instancję klasy SymbolTranslator z gorliwym ładowaniem pliku path w chwili pierwszej potrzeby tłumaczenia.

\_\_init\_\_(self, tokenizer, negative\_allowed=True) – konstruktor pobierający instancję tokenizera (obiekt zwracający dane wejściowe już w poddanej tokenizacji formie) oraz informację czy zdania negatywne również powinny być parsowane (w przypadku negative=False zdania negatywne zwracane przez tokanizera będą ignorowane).

get\_sentences(self) – zwraca generator zwracający kolejne zdania gramatyki. Zdania są pobierane leniwie z tokenizera (od klas położonych niżej zależy czy są wczytywane gorliwie, czy też leniwie).

symbol\_to\_word(self, symbol) – klasa tłumacząca symbol. Generuje reprezentację dla symboli nieterminalnych (z wykorzystaniem metod klasy sgcs.core.Symbol), zwraca reprezentację symboli specjalnych oraz odczytuje na podstawie zdobytej podczas parsowania wiedzy symbole terminalne. Aby metoda ta prawidłowo zadziałała, symbol terminalny musiał wystąpić już w jakimś zdaniu, aczkolwiek ponieważ klasa ta jest jedynym źródłem zdań, a zatem również symboli terminalnych, to każdy symbol musiał już wystąpić w jakimś zdaniu podczas działania algorytmu. Niekoniecznie jest to prawdą w przypadku edycji populacji poza działającym algorytmem i to jest powodem, dla którego należy przy jej dokonywaniu zachować szczególną ostrożność.

word\_to\_symbol(self, word) – metoda tłumacząca słowo terminalne na instancję klasy Symbol. Dotyczą się jej te same uwagi i obostrzenia co metody symbol\_to\_word.

rule\_population\_to\_string(self, rule\_population) – funkcja zwracająca napis stanowiący wygodną w czytaniu reprezentację populacji reguł. Klasa ta wykorzystuje metodę symbol\_to\_word, co może być potencjalnym źródłem problemów napotykanych przy wykorzystaniu tej metody.

EagerTokenizer

Klasa, której zadaniem jest podział linii zwróconych przez obiekt data\_fetcher na łatwe w parsowaniu tokeny. Nazwa może być myląca, EagerTokenizer dokonuje leniwej tokenizacji.

\_\_init\_\_(self, data\_fetcher) – konstruktor przyjmujący obiekt data\_fetcher odpowiedzialny za dostarczanie zdań w postaci linii tekstu.

get\_chunk\_generator(self) – metoda tworząca generator zwracający kolejne linie w postaci listy tokenów

EagerFileFetcher

Klasa odpowiedzialna za gorliwe wczytywanie pliku z danymi wejściowymi. Zamiast tej klasy można zdecydować się na LazyFileFetcher (wczytujący plik leniwie).

\_\_init\_\_(self, path) – konstruktor zapamiętujący ścieżkę path z danymi wejściowymi.

get\_chunk\_generator(self) – zwraca generator zwracający kolejne zdania (w sposób leniwy lub chowając pod sobą listę).

BasicJsonizer

Klasa tłumacząca obiekty dziedziczące po klasie SimpleJsonNode na format .json i odwrotnie. Tłumaczona klasa oraz jej składowe musi być typu podstawowego albo równocześnie dziedziczyć po klasie SimpleJsonNode i być na liście expected\_classes. Klasę tę wykorzystuje się głównie do tłumaczenia węzłów konfiguracji.

node\_id – stała będąca kluczem, pod którym zapisuje się w json nazwy klas, dzięki czemu jest możliwe ich późniejsze odtworzenie.

\_\_init\_\_(self, expected\_classes) – konstruktor pobierający listę obiektów klas, które mają być obsługiwane przez instancję (wszystkie powinny dziedziczyć po klasie SimpleJsonNode).

to\_json(self, obj) – metoda tłumacząca obj na napis w formacie json.

from\_json(self, json) – metoda tłumacząca napis w formacie json na obiekt.

RulePopulationJsonizer

Klasa odpowiedzialna za serializację obiektów typu RulePopulation.

make\_binding\_map(binding\_list) – funkcja przyjmująca listę klas, z których następnie tworzy mapę nazwa klasy – klasa.

\_\_init\_\_(self, bindings) – konstruktor przyjmujący mapę nazwa-klasa obsługiwanych klas (na przykład RulePopulation oraz klasy dziedziczące).

to\_json(self, population) – klasa zwracająca napis w formacie json będący tłumaczeniem obiektu population.

from\_json(self, json, randomizer, \*args, \*\*kwargs) – metoda zwracająca obiekt obiekt populacji tworzony na podstawie napisu w formacie json. Ponieważ populacje potrzebują zazwyczaj dodatkowych parametrów, są one również przekazywane.