

Instrukcja obsługi programu

Random Formula Generator

dla wersji v1.0.0

Jakub Semczyszyn
jakub.semczyszyn@student.agh.edu.pl
jakub.semczyszyn@gmail.com

Styczeń 2021

Spis treści

1	Cel dokumentu	3
2	Definicje	3
3	Wymagania i instalacja	3
4	Zawartość archiwum programu	4
5	Użytkowanie programu	4
5.1	Generowanie pojedynczych formuł	5
5.2	Wykorzystywanie jedynie części służącej do pomiarów	5
5.3	Przetwarzanie scenariuszy testowych	6
6	Dostępne typy problemów	8
7	Opis elementów programu	9
7.1	Moduł randomgen	10
7.1.1	__init__	10
7.1.2	generate	11
7.1.3	generateProblem1	12
7.1.4	generateProblem2	12
7.1.5	generateProblem3	13
7.1.6	generateProblem4	13
7.1.7	generateProblem5	13
7.1.8	generateProblem6	14
7.1.9	generateProblem7	14
7.1.10	generateProblem8	14
7.1.11	generateFormulaR	15
7.1.12	generateSafetyClause	15
7.1.13	generateLivenessClause	15
7.1.14	generateClause	15
7.1.15	getRandomAtomList	15
7.1.16	getRandomRelation	16
7.1.17	replaceORwithIMP	16
7.1.18	cleanup	16
7.1.19	getMostFrequentKey	16
7.1.20	saveFormulas	17
7.1.21	printFormula	17

7.1.22	translateAndSave	17
7.1.23	translateToProver9	18
7.1.24	translateToSPASS	19
7.1.25	joinProver9FilesWithPattern	21
7.1.26	joinSPASSFilesWithPattern	22
7.1.27	__getAtomStr	23
7.2	Moduł provrun	24
7.2.1	__init__	24
7.2.2	performMeasurements	24
7.2.3	runProver	24
7.2.4	getMeasuresFromOutputFile	25
7.3	Moduł testenv	26
7.3.1	Klasa Case	26
7.3.2	Klasa CaseMaker	27
7.3.3	Klasa TestEnv	27
7.4	Moduł utils	29
7.4.1	Klasa LogicToken	29
7.4.2	Funkcja zipToList	29
7.4.3	Funkcja readFormulaFile	29
7.5	Moduł customerrors	30
8	Objaśnienie nazw generowanych plików	30
9	Opis rzucanych wyjątków	30
10	Modyfikowanie programu	32
10.1	Dodawanie nowych typów problemów	32
10.2	Dodawanie nowych proverów	32
10.3	Zmienianie parametrów i limitów	33

1 Cel dokumentu

Celem niniejszej instrukcji jest wytłumaczenie użytkownikom sposobów implementacji poszczególnych funkcjonalności programu i zależności między nimi. Nie tłumaczy ona zasad wykorzystywanej logiki, ani sposobu działania wykorzystywanych w programie systemów dowodzenia twierdzeń (instrukcje dla nich można znaleźć pod adresami: [Prover9](#), [SPASS Prover](#)).

2 Definicje

W dalszej części dokumentu używane będą następujące określenia:

Atom to w formacie wewnętrznym programu typ tokenu zawierającego nazwę zmiennej logicznej oraz informację o jej negacji; w formatach proverów jest reprezentowany przez predykaty 0- i 1-argumentowe.

Klauzula to dysjunkcja dowolnej liczby atomów (lub implikacja pomiędzy dwoma takimi dysjunkcjami), do której zaaplikowano jeden lub więcej kwantyfikatorów.

Formuła to koniunkcja dowolnej liczby klauzul. (W bardziej złożonych problemach, formułą będzie również nazywane wiele w ten sposób zdefiniowanych formuł, które dla rozróżnienia będą wtedy nazywane formułami składowymi, połączonych dowolnymi symbolami logiki klasycznej lub symbolem równości.)

Klauzula bezpieczeństwa to klauzula postaci

$$\forall_{t. t \geq t_0} : \neg B(t)$$

gdzie B jest dysjunkcją atomów, t to zmienna modelowanego układu, a t_0 to minimalna wartość tej zmiennej.

Klauzula żywotności może występować w postaci ogólnej

$$\exists_{t. t \geq t_0} : Q(t)$$

gdzie Q jest dysjunkcją atomów, t to zmienna modelowanego układu, a t_0 to minimalna wartość tej zmiennej;

lub w postaci warunkowej

$$\forall_{t. t \geq t_0} : \exists_{t_1. t \geq t_1} : P(t_1) \Rightarrow Q(t)$$

gdzie P i Q to dysjunkcje różnych atomów, t to zmienna modelowanego układu, t_0 to minimalna wartość tej zmiennej, a t_1 to jakaś pośrednia wartość tej zmiennej, pomiędzy wartością minimalną, a bieżącą wartością.

3 Wymagania i instalacja

Do poprawnego działania wszystkich elementów aplikacji konieczne jest uprzednie **zainstalowanie aktualnych wersji SPASS Provera oraz Prover9, a następnie dodanie ich lokalizacji do zmiennej \$PATH**. Ponieważ proverzy te przeznaczone są na systemy UNIXowe, to również program został stworzony dla **systemu Linux**. Podczas instalacji najnowszych wersji systemów dowodzenia twierdzeń, użytkownik może napotkać pewne trudności. SPASS Prover wymaga wcześniejszej instalacji programów *bison* oraz *flex*, co można w prosty sposób uczynić komendą *apt-get install*. Instalacja

Prover9 może zwrócić błąd związany z funkcją *round* w pliku *search.c*. Należy wtedy edytować wspomniany plik i zastąpić warunek, w którym ta funkcja jest użyta, analogiczną dysjunkcją warunków z użyciem funkcji *floor* oraz *ceil*. Nie zmieni to działania kodu, a pozwoli programowi Make na zlinkowanie odpowiednich funkcji. Przedstawiony problem został zaobserwowany na dystrybucjach systemu Ubuntu 14 i nowszych (autor programu nie posiada wiedzy na temat jego występowania na innych dystrybucjach Linuxa) i prawdopodobnie nie zostanie rozwiązany przez twórcę provera, ponieważ ostatnia jego aktualizacja została wydana w listopadzie 2009 roku.

Żeby program nie powodował błędów krytycznych, wymagane są również **minimum 2 GB pamięci RAM**, ponieważ Prover9 może wykorzystywać cały 1 GB do swoich poszukiwań.

W obecnej wersji programu instalacja polega jedynie na rozpakowaniu archiwum z kodem programu. W przyszłości planowana jest publikacja problemu jako package, aby dało się go pobierać przy pomocy komendy *apt-get*.

4 Zawartość archiwum programu

Archiwum programu w wersji 1.0.0 zawiera:

- Folder *dane_testowe* - w którym znajdują się przykładowe próbne pliki wejściowe i wyjściowe dla obu proverów oraz przykładowy scenariusz.
- Plik *.gitignore* - przynależny do systemu kontroli wersji, nie będący częścią programu.
- Plik *README.md* - obecnie nie wykorzystywany, w przyszłości będzie potrzebny do dystrybucji programu.
- Moduł *customerrors.py* - zawierający definicje błędów używanych w programie.
- Plik *main.py* - przykładowy plik z dwoma sposobami uruchamiania programu.
- Moduł *provrn.py* - zawierający część programu odpowiedzialną za uruchamianie proverów i pomiar wydajności.
- Moduł *randomgen.py* - zawierający kod do generacji i translacji formuł.
- Plik *requirements.txt* - zawierający skróconą listę wymagań (pełniejsza jej wersja znajduje się w niniejszej instrukcji)
- Moduł *testenv.py* - zawierający kod środowiska testowego.
- Moduł *utils.py* - zawierający funkcje i klasy pomocnicze.

5 Użytkowanie programu

Wykorzystanie programu wymaga utworzenia krótkiego skryptu języka Python3, uruchamiającego odpowiednie jego części. Wszystkie generowane przez program pliki zostają umieszczone w folderze *generated_files*, znajdującym się w lokalizacji programu.

5.1 Generowanie pojedynczych formuł

Aby użyć programu, należy w pierwszej kolejności utworzyć instancję klasy `Generator`, przekazując do konstruktora pożądane parametry. Nastąpi wówczas sprawdzenie poprawności parametrów oraz wygenerowana zostanie na ich podstawie formuła w wewnętrznym formacie generatora. Formuła zostanie również zapisana do pliku. Następnie użytkownik musi wywołać metodę generatora `translateAndSave()`, co spowoduje, że formuła zostanie, w odpowiedni dla parametrów wejściowych generatora sposób, przetłumaczona na formaty SPASS Provera i Prover9, tłumaczenie to zostanie zapisane do plików, a użytkownikowi zwrócony zostanie słownik zawierający opisane nazwy tych plików. Na tym etapie działania programu, użytkownik posiada gotowe do użycia pliki wejściowe dla testowanych w tej pracy systemów dowodzenia twierdzeń, zawierające wygenerowaną losowo formułę. W następnym kroku możliwe jest uruchomienie proverów na tych plikach i uzyskanie wyników ich działania. W tym celu należy stworzyć instancję klasy `ProverRunner`, jako parametry konstruktora podając nazwę wybranego provera ("prover9" lub "spass") oraz ścieżkę do pliku. Następnie trzeba wywołać metodę runnera `performMeasurements()` z nazwą docelowego pliku wyjściowego provera, jako parametrem. Program utworzy wtedy nowy proces i uruchomi w nim system dowodzenia twierdzeń, przekierowując jego wynik do pliku o podanej nazwie, a po zakończeniu pracy systemu, odczyta z pliku wyniki jego działania i zwróci je użytkownikowi jako słownik, zawierający informacje o wykorzystanym czasie i pamięci oraz wynik dowodzenia, będący informacją o spełnialności, jeśli system zadziałał poprawnie, lub o błędzie, w przeciwnym wypadku. Poniżej przedstawiono przykładowe wywołania pozwalające na skorzystanie z funkcjonalności programu:

```
from provrun import ProverRunner
from randomgen import Generator

test = Generator('problem3', clauses_num=2000, atoms_num_coeff=3)
files = test.translateAndSave()
runner = ProverRunner("prover9", input_file=files["prover9_input"])
current_output_file = files["output"] + "_prover9.out"
usage_dict = runner.performMeasurements(current_output_file)
print(usage_dict)
```

Takie wykorzystanie programu spowoduje utworzenie pliku .txt z zapisem formuły w wewnętrznym formacie generatora, pliku wejściowego .in dla każdego z proverów oraz pliku wyjściowego .out dla każdego uruchomionego provera.

5.2 Wykorzystywanie jedynie części służącej do pomiarów

Możliwe jest również użycie programu jedynie w celu zmierzenia wydajności Prover9 lub SPASS Prover na posiadanym już pliku w formacie wejściowym danego provera. W takim wypadku wystarczy jedynie utworzyć instancję klasy `ProverRunner`, jako parametry konstruktora podając nazwę wybranego provera ("prover9" lub "spass") oraz ścieżkę do posiadanego pliku, a następnie wywołać metodę `performMeasurements()` z nazwą docelowego pliku wyjściowego provera, jako parametrem.

Przykład dla Prover9:

```
from provrun import ProverRunner

filepath = "~/Documents/Logical_problems/formula17_prover9.in"
runner = ProverRunner("prover9", input_file=filepath)
output_file = filepath[:-2] + "out"
```

```
usage_dict = runner.performMeasurements(output_file)
print(usage_dict)
```

Przykład dla SPASS Prover:

```
from provrun import ProverRunner

filepath = "~/Documents/Logical_problems/formula17_spass.in"
runner = ProverRunner("spass", input_file=filepath)
output_file = filepath[:-2] + ".out"
usage_dict = runner.performMeasurements(output_file)
print(usage_dict)
```

Takie wykorzystanie programu spowoduje utworzenie pliku wyjściowego .out dla każdego uruchomionego provera.

5.3 Przetwarzanie scenariuszy testowych

Scenariusz należy zapisać w zwykłym pliku tekstowym. Każdy rodzaj testu musi zostać umieszczony w osobnym bloku (przez rodzaj testu rozumiany w tym miejscu jest typ problemu, ale również podtypy problemów 7 i 8 wymagają umieszczenia w osobnych blokach oraz wersje problemów 6, 7 i 8 z wykorzystaniem rozkładu Poissona muszą być umieszczone w innych blokach, niż te z rozkładem równomiernym). Każdy blok składa się z nagłówka, listy parametrów oraz stopki. Nagłówek to linia z typem problemu - przykładowo "problem2". Stopka to linia z napisem "end of problem". Lista parametrów to linie zaczynające się od nazwy parametru, po której następują kolejne wartości parametrów, rozdzielone spacjami. Lista dostępnych parametrów wygląda następująco:

- "clauses" - odpowiada za ilość klauzul. Kolejne wartości będą użyte w osobnych przypadkach testowych.
- "safety_prec" - odpowiada za procentowy udział klauzul bezpieczeństwa w formule. Kolejne wartości będą użyte w osobnych przypadkach testowych.
- "lengths" - odpowiada za długości klauzul. Jeżeli problem jest typu 4, to kolejne wartości będą użyte w osobnych przypadkach testowych, w innym wypadku, wszystkie zostaną użyte w każdym teście, złączone w listę długości.
- "n_coeff" - odpowiada za współczynnik ilości atomów. Kolejne wartości będą użyte w osobnych przypadkach testowych.
- "poisson" - decyduje, czy długości klauzul w formule mają mieć rozkład Poissona, czy równomierny. Przyjmuje pojedynczą wartość - "True" lub "False".
- "lambda" - odpowiada za wartość oczekiwaną rozkładu Poissona. Przyjmuje pojedynczą wartość i zostanie wykorzystana jedynie jeśli typ problemu to problem2 lub wartość parametru "poisson" to "True"
- "distribution" - odpowiada za typ rozkładu w problemie 5. Przyjmuje wartości "even", "more_short" oraz "more_long" i będą one użyte kolejno, w osobnych przypadkach testowych.

Aby testy wykonały się pomyślnie, wszystkie parametry muszą zostać podane zgodnie z wymaganiami przedstawionymi w opisach problemów. Dodatkowo, z uwagi na sposób przetwarzania scenariusza przez środowisko testowe, **konieczne jest, aby plik był zakończony pustą linią.**

Przykładowy scenariusz może wyglądać następująco:

```
1 problem1
2
3 clauses 50 100 200 500 1000 2000
4 safety_prec 50
5 lengths 2 3 4 6 8 10
6 n_coeff 0.5
7
8 end of problem
9
10 problem2
11
12 clauses 50 100 200 500 1000 2000
13 safety_prec 50
14 lengths 2 3 4 6 8 10
15 n_coeff 0.5
16 poisson True
17 lambda 3.5
18
19 end of problem
20
21 problem3
22
23 clauses 50 100 200 500 1000 2000
24 safety_prec 50
25 lengths 2 3 4 6 8 10
26 n_coeff 2 3 4 5 10
27
28 end of problem
29
```

Środowisko testowe automatycznie wywołuje wszystkie metody przedstawione w poprzednim punkcie. Użytkownik musi więc jedynie stworzyć plik ze scenariuszem testowym, a potem utworzyć obiekt klasy `TestEnv`, podając jako parametr ścieżkę do pliku scenariusza, i wywołać na nim metodę `makeTests()`. Następnie wystarczy już czekać na zakończenie pracy programu, co zostanie zasygnalizowane poprzez wyświetlenie stosownego napisu w konsoli (po którym wyświetlony powinien zostać znak zachęty). W zależności od złożoności scenariusza, może to zająć nawet kilka lub kilkanaście godzin.

Przykładowe uruchomienie środowiska testowego:

```
from testenv import TestEnv

enviroment = TestEnv("~/Documents/Logical_problems/Prover_scenarios/scenario2")
enviroment.makeTests()
```

Takie wykorzystanie programu spowoduje utworzenie pliku `.txt` z zapisem każdej wygenerowanej formuły w wewnętrznym formacie generatora, pliku wejściowego `.in` każdej formuły dla każdego z proverów, trzech plików wyjściowych `.out` dla każdej formuły dla każdego z proverów, pliku `error_log.txt` zawierającego informacje o napotkanych w trakcie działania programu błędach oraz pliku `test_session_results.csv` zawierającego zestawienie parametrów oraz wyników każdego z testów.

6 Dostępne typy problemów

Program udostępnia możliwość wygenerowania formuły reprezentującej jeden z ośmiu predefiniowanych problemów, które ilustrują różne, wymagające dla systemów dowodzenia twierdzeń, aspekty problemów modelowanych na podstawie świata rzeczywistego. Problemy te są oznaczone numerami, jak następuje:

1. Problem1 - jest to najbardziej podstawowy problem, na którym bazować będzie większość pozostałych. Służy badaniu, jak wielkość formuły wpływa na czas i pamięć wykorzystywane przy jej dowodzeniu. W tym przypadku formuła będzie składać się z klauzul o różnych długościach, a ilość klauzul każdej długości będzie w przybliżeniu jednakowa. Przy generowaniu tego problemu brane są pod uwagę następujące parametry: liczba klauzul oraz lista długości klauzul. Dodatkowo liczba atomów musi być równa połowie liczby klauzul, lista długości klauzul musi zawierać przynajmniej 2 elementy, a klauzul bezpieczeństwa i żywotności musi być w przybliżeniu po równo.
2. Problem2 - problem ten ma podobne założenia, co Problem1, jednak długości klauzul nie są z góry narzucone, a generowane na podstawie rozkładu Poissona. Zamiast listy długości klauzul, wykorzystywana jest wartość λ , opisująca generowany rozkład Poissona - określa ona długość klauzul, których w formule będzie najwięcej. Pozostałe ograniczenia są identyczne, jak w problemie pierwszym.
3. Problem3 - służy w szczególności badaniu, jak duża liczba atomów w formule wpływa na działanie proverów. Oprócz długości formuły i listy długości klauzul, jak w problemie pierwszym, przyjmuje dodatkowo współczynnik liczby atomów, którego wartość musi być większa od 1. Ilość atomów zostanie wyliczona poprzez pomnożenie liczby klauzul przez ten współczynnik i jednocześnie będzie on wyznaczał maksymalną długość klauzuli (wyższe długości z listy nie będą brane pod uwagę). Dodatkowo lista długości klauzul musi zawierać przynajmniej 2 elementy, a klauzul bezpieczeństwa i żywotności musi być w przybliżeniu po równo.
4. Problem4 - służy do mierzenia, jak systemy dowodzenia radzą sobie z formułami, w których wszystkie klauzule mają jednakową długość. Oprócz ilości klauzul, używa listy długości klauzul, jednak musi ona zawierać dokładnie jeden element. Ponadto wymagane jest, aby liczba atomów była równa połowie liczby klauzul, a klauzul bezpieczeństwa i żywotności było w przybliżeniu po równo.
5. Problem5 - służy do sprawdzenia, jak długość klauzul wpływa na osiągi proverów. Ze względu na zaprojektowane rozkłady, wymaga podania dokładnie czterech długości klauzul i żeby testy faktycznie pokazały ich wpływ, powinny one silnie różnić się od siebie. Dostępne rozkłady to: "even" - klauzul każdej długości jest w formule po 25%, "more_short" - klauzul o największej długości jest jedynie 1%, a pozostałych po 33%, "more_long" - klauzul o najmniejszej długości jest 1%, a pozostałych po 33%. Dodatkowo należy podać ilość klauzul, liczba atomów musi być równa połowie liczby klauzul, a klauzul bezpieczeństwa i żywotności musi być w przybliżeniu po równo.
6. Problem6 - służy do badania, jaki wpływ na wydajność proverów ma stosunek ilości klauzul bezpieczeństwa i żywotności. Oprócz listy długości klauzul oraz ilości klauzul należy podać, jaki procent klauzul ma być klauzulami bezpieczeństwa. Dodatkowo lista długości klauzul musi zawierać przynajmniej 2 elementy, atomów musi być dwa razy mniej niż klauzul i możliwe jest zignorowanie listy długości klauzul i wykorzystanie zamiast niej rozkładu Poissona, w sposób analogiczny, jak w problemie drugim.

7. Problem7 - służy do mierzenia, jak połączenie kilku formuł wpływa na czas i pamięć poszukiwań dowodów, a ponadto znacząco zwiększa obciążenie proverów, ponieważ formuły składowe są przedstawiana w Conjunctive Normal Form zamiast Clause Normal Form. Dostępne są 2 predefiniowane wzorce łączenia formuł:

- Problem7a: $\neg(F1 \vee F2 \vee F3) \vee R$
- Problem7b: $\neg(F1 \wedge F2 \wedge F3) \vee R$

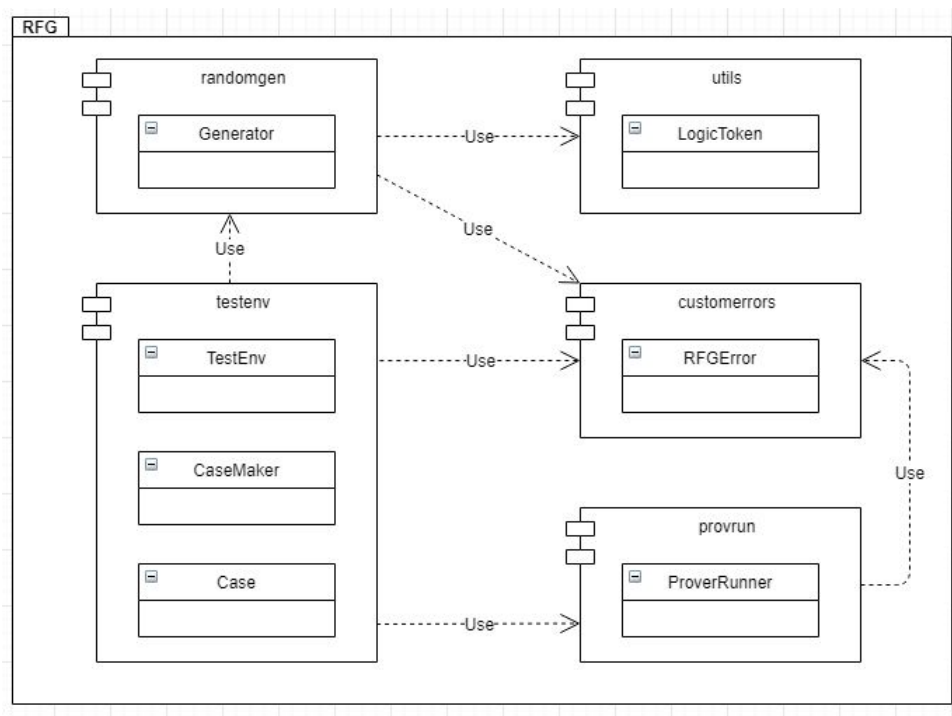
Generowane formuły typu F reprezentują Problem1, a formuła typu R składa się z pojedynczej formuły żywotności w postaci ogólnej, zawierającej 4 literały. Lista długości klauzul musi zawierać przynajmniej 2 elementy, atomów musi być dwa razy mniej niż klauzul i możliwe jest zignorowanie listy długości klauzul i wykorzystanie zamiast niej rozkładu Poissona, w sposób analogiczny, jak w problemie drugim.

8. Problem8 - podobnie, jak Problem7 polega na łączeniu formuł, a poszczególne podtypy problemu pozwalają na porównywanie wpływu różnych działań logicznych na wydajność. Formuły składowe reprezentują Problem1 lub, jeśli użytkownik wybierze rozkład Poissona, Problem2. Dostępne wzorce łączenia formuł to:

- Problem8a: $(\neg F1 \vee \neg F2) \wedge (F1 \vee F2)$
- Problem8b: $\neg(\neg F1 \wedge \neg F2)$
- Problem8c: $(\neg F1 \vee F2) \wedge \neg(\neg F2 \vee F1)$

7 Opis elementów programu

Aplikacja składa się z dwóch głównych modułów: generatora formuł logicznych (zwanego dalej generatorem) [randomgen] oraz menedżera pracy systemów dowodzenia twierdzeń (zwanego dalej prover runnerem lub po prostu runnerem) [provrn]. Oprócz nich w skład aplikacji wchodzi również dwa moduły pomocnicze: jeden z nich zawiera klasy i metody użytkowe, uniwersalne, nieprzynależne wyłącznie do jednego modułu [utils], drugi definiuje nową klasę wyjątków, przystosowaną do potrzeb diagnostycznych programu [customerrors]. Ponadto dostępny jest moduł implementujący środowisko testowe umożliwiające automatyzację testów [testenv]. Zależności między poszczególnymi modułami pokazano na diagramie [Figure 1].



Rysunek 1: Diagram przedstawiający zależności pomiędzy poszczególnymi modułami oraz zawarte w nich klasy.

7.1 Moduł randomgen

Za generowanie losowych formuł logicznych oraz translację wygenerowanych formuł odpowiada zawarta w module randomgen klasa Generator. Generowanie odbywa się już w trakcie wywołania konstruktora klasy, a jego implementacja jest zhierarchizowana - każda metoda biorąca w nim udział wywołuje bardziej szczegółową metodę, począwszy od konstruktora wywołującego metodę *generate()*, kończąc, na przykład, na metodzie *getRandomAtomList()*.

Tłumaczenie odbywa się poprzez wywołanie metody generatora *translateAndSave()*.

Do klasy Generator należą metody:

7.1.1 `__init__`

```
__init__(self, test_type, precentage_of_safety_clauses=50, clause_lengths=[2,3,4,6,8,10],
clauses_num=50, poisson=False, atoms_num_coeff=0.5, problem5_distribution="even",
lambda_value=3.5)
```

Konstruktor może przyjmować wszystkie parametry potrzebne w każdym z typów problemów, jednak jedynie sam typ problemu jest parametrem wymaganym, reszta ma ustawione wartości domyślne, w pełni zgodne z podstawowym problemem - typu Problem1. Możliwe parametry to:

- **test_type** - określa typ generowanego problemu. Musi być napisem w postaci "problemX*", gdzie 'X' to cyfra od 1 do 8, a '*' to, potrzebna przy problemie 7 i 8, litera od a do c.
- **precentage_of_safety_clauses** - określa, ile procent klauzul ma być klauzulami bezpieczeństwa (pozostałe to klauzule żywotności). Domyślnie ma wartość 50 i poza problemem 6 nie powinna być ona zmieniana.

- **clause_lengths** - lista długości klauzul, wyrażonych w ilości literałów. Domyślnie jest to [2, 3, 4, 6, 8, 10].
- **clauses_num** - ilość klauzul, czyli długość formuły. Domyślnie jest to 50.
- **poisson** - zmienna logiczna, która określa, czy generator ma korzystać z rozkładu Poissona. Domyślnie jest ustawiona na False, a jej zmiana jest możliwa jedynie w problemach 2, 6, 7 i 8.
- **atoms_num_coeff** - współczynnik ilości atomów, omówiony w opisie problemu 3. Domyślnie ma wartość 0,5, co oznacza, że liczba wykorzystywanych w formule atomów jest dwa razy mniejsza niż długość formuły (liczona w klauzulach) i nie powinna być ona zmieniana poza problemem 3.
- **problem5_distribution** - parametr ten jest używany wyłącznie w problemie 5 i jest objaśniony w jego opisie. Domyślnie przyjmuje wartość "even".
- **lambda_value** - wartość lambda używana do utworzenia rozkładu Poissona, w problemach, które go nie wykorzystują jest ignorowana. Domyślnie ma wartość 3,5.

Konstruktor inicjuje listy klauzul, reprezentujące 4 formuły (formula, formula2, formula3 oraz formulaR), choć większość problemów będzie wykorzystywać jedynie pierwszą z nich. Wylicza liczbę atomów, mnożąc ilość klauzul przez współczynnik ilości atomów i zaokrąglając wynik do pełnych jedności. Parametry, które będą potrzebne na "głębszym" poziomie generowania, niż metody typu *generateProblem*, zostają zapamiętane w polach klasy. Następnie konstruktor tworzy słownik mapujący nazwy atomów na ilość ich wystąpień. Nazwy są generowane po kolei od var1 do varN, gdzie N to wyliczona liczba atomów, a ilość wystąpień każdego atomu jest początkowo ustawiana na 0. W kolejnym kroku generowany jest trzon nazwy wszystkich plików, które będą związane z wygenerowaną formułą. Ma on postać: "{test_type}_c{clauses_num}_a{atoms_num}_prec{percentage_of_safety_clauses}_lengths{zawartość listy clause_lengths rozdzielona znakami ' '}{_poisson}*{_distribution}_{problem5_distribution}**" (*ten człon występuje jeśli w formule wykorzystany jest rozkład Poissona; **ten człon występuje jeśli problem jest typu 5).

Na koniec, konstruktor wywołuje kolejno: metodę *generate()*, przekazując jej wszystkie potrzebne parametry wejściowe, aby wygenerować formułę, metodę *cleanup()*, aby dokonać korekty formuły oraz metodę *saveFormulas()*, aby zapisać formułę do pliku, stosując wewnętrzny format zapisu.

7.1.2 generate

generate(self, percentage_of_safety_clauses, clause_lengths, clauses_num, poisson, atoms_num_coeff, problem5_distribution)

Parametry tej metody bezpośrednio odpowiadają tym z konstruktora i wszystkie są wymagane:

- **percentage_of_safety_clauses** - określa, ile procent klauzul ma być klauzulami bezpieczeństwa (pozostałe to klauzule żywotności).
- **clause_lengths** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **poisson** - zmienna logiczna, która określa, czy generator ma korzystać z rozkładu Poissona.
- **atoms_num_coeff** - współczynnik ilości atomów, omówiony w opisie problemu 3.
- **problem5_distribution** - parametr ten jest używany wyłącznie w problemie 5 i odpowiada za rozkład długości klauzul, jest objaśniony w opisie problemu.

Metoda ta sprawdza, czy wszystkie parametry wejściowe są zgodne z typem problemu (to znaczy, czy spełniają wymagania zawarte w opisie tego problemu). Jeśli tak jest, to wywołuje odpowiednią metodę typu *generateProblem*, przekazując jej potrzebne parametry, a jeśli nie to rzuca wyjątek.

7.1.3 generateProblem1

generateProblem1(self, clauses_lengths, clauses_num, safety_coeff=0.5, target_formula=1)

- **clauses_lengths** - lista długości klauzul, wyrażonych w ilości literalów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **safety_coeff** - współczynnik ilości klauzul bezpieczeństwa.
- **target_formula** - parametr ten wskazuje, do której listy zapisana ma być formuła. 1 oznacza *self.formula*, 2 oznacza *self.formula2*, a 3 oznacza *self.formula3*.

Jako, że problem typu 1 jest bazowym modelem dla pozostałych problemów, metoda generująca go jest wzorcową metodą dla metod generujących inne problemy, które albo wykorzystują ją bezpośrednio, albo implementują jej algorytm z drobnymi modyfikacjami.

Najpierw wyliczane jest, ile klauzul każdej długości ma zostać wygenerowanych oraz, ile z nich ma być klauzulami bezpieczeństwa. Ze względu na zaokrąglenia, odwracając działania, za pomocą których obliczane są te wartości, można uzyskać sumaryczną ilość klauzul mniejszą, bądź większą, niż ta przekazana jako argument, stąd w kodzie występują dodatkowe warunki i korekty. Dla każdej długości z listy długości klauzul generowana jest odpowiednia ilość klauzul bezpieczeństwa i żywotności, pod warunkiem, że docelowa ilość klauzul nie została osiągnięta, w takim przypadku dalsze klauzule nie zostaną wygenerowane i tym samym najdłuższych klauzul będzie o kilka mniej, niż pozostałych. Wygenerowane klauzule są dodawane do tymczasowej listy klauzul, nazywanej formułą. Po zakończeniu generowania, sprawdzane jest, czy z powodu zaokrągleń nie zostało wygenerowane zbyt mało klauzul. Jeśli tak jest, to dodatkowe klauzule są generowane z losowymi długościami z listy, przy zachowaniu stosunku ilości klauzul bezpieczeństwa do żywotności, a następnie dodawane do tymczasowej formuły. Ostatecznie tymczasowa formuła zostaje skopiowana do formuły odpowiadającej numerowi przekazanemu jako parametr.

7.1.4 generateProblem2

generateProblem2(self, clauses_num, safety_coeff=0.5, target_formula=1)

- **clauses_num** - ilość klauzul, czyli długość formuły.
- **safety_coeff** - współczynnik ilości klauzul bezpieczeństwa.
- **target_formula** - parametr ten wskazuje, do której listy zapisana ma być formuła. 1 oznacza *self.formula*, 2 oznacza *self.formula2*, a 3 oznacza *self.formula3*.

Metoda ta implementuje wzorcowy algorytm dla metod używających rozkładu Poissona. Główną różnicą między nim, a algorytmem dla problemu 1 jest sposób wyliczania ilości klauzul poszczególnych długości. W tym przypadku nie jest to jedna ilość, a osobna ilość dla każdej długości, wynikająca z rozkładu Poissona. Dla kolejnych liczb naturalnych, reprezentujących długości klauzul, począwszy od 1, wyliczany jest iloczyn funkcji masy prawdopodobieństwa dla rozkładu Poissona przy zadanej wartości λ oraz ilości klauzul. Jego zaokrąglenie do pełnych jedności zapisywane jest w liście ilości klauzul

poszczególnych długości oraz podobnie jak w problemie 1, wyliczane jest, ile z nich będzie klauzulami bezpieczeństwa i ta wartość również jest zapisywana, do osobnej listy. Procedura jest przerywana, gdy obecnie przetwarzana długość przekroczyła wartość λ , a wynik zaokrąglenia iloczynu jest równy 0 - oznacza to, że dla każdej następnej długości również będzie równy 0 i poprzednio wygenerowana długość była maksymalną. W dalszej części algorytmu, zamiast pojedynczej ilości klauzul każdej długości oraz klauzul bezpieczeństwa każdej długości, używane są kolejno wartości z wygenerowanych w ten sposób list.

7.1.5 generateProblem3

generateProblem3(self, clause_length, clauses_num, atoms_num_coeff)

- **clause_length** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **atoms_num_coeff** - współczynnik ilości atomów - stosunek ilości atomów do ilości klauzul.

Jako, że ilość atomów została już obliczona w konstruktorze, a same atomy zostały również wygenerowane, jedyną różnicę między problemem 3, a problemem 1 na tym etapie stanowi fakt, że współczynnik ilości atomów ogranicza maksymalną długość klauzul. Metoda ta usuwa więc z listy długości klauzul te, które nie spełniają powyższego kryterium i, z tak zmienioną listą, uruchamia metodę *generateProblem1()*.

7.1.6 generateProblem4

generateProblem4(self, clause_length, clauses_num, safety_coeff=0.5)

- **clause_length** - długość klauzul, wyrażona w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **safety_coeff** - współczynnik ilości klauzul bezpieczeństwa.

Użycie pojedynczej długości klauzul w problemie 4 sprawia, że nie ma potrzeby zaokrąglania ilości klauzul, więc algorytm może zostać uproszczony o korekty i sprawdzenia. Wyliczana jest zaokrąglona ilość klauzul bezpieczeństwa, a następnie generowane są klauzule bezpieczeństwa w tej ilości i klauzule żywotności w ilości równej różnicy ilości klauzul i tej ilości. Wszystkie klauzule dodawane są do formuły.

7.1.7 generateProblem5

generateProblem5(self, clauses_length, clauses_num, distribution, safety_coeff=0.5)

- **clauses_length** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **distribution** - typ rozkładu długości klauzul. Przy posortowanej liście długości, wartość "even" oznacza rozkład 25%-25%-25%-25%, wartość "more_long" oznacza rozkład 1%-33%-33%-33%, a wartość "more_short" oznacza rozkład 33%-33%-33%-1%.
- **safety_coeff** - współczynnik ilości klauzul bezpieczeństwa.

Algorytm w tej metodzie jest zmieniony podobnie, jak w przypadku problemu 2, ale wartości w listach ilości klauzul każdej długości i ilości klauzul bezpieczeństwa każdej długości są wyliczane przez proste mnożenie ilości wszystkich klauzul przez procent, jaki klauzule danej długości mają stanowić w formule.

7.1.8 generateProblem6

generateProblem6(self, clause_length, clauses_num, safety_precentage, poisson)

- **clause_length** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **safety_precentage** - procent klauzul bezpieczeństwa (liczba całkowita).
- **poisson** - zmienna logiczna, która określa, czy generator ma korzystać z rozkładu Poissona.

Ponieważ metody *generateProblem1()* i *generateProblem2()* mają możliwość przekazania jako parametr, ile procent klauzul stanowią klauzule bezpieczeństwa, to w zależności od wartości parametru *poisson*, ta metoda uruchamia jedną z nich, przekazując odpowiednie parametry.

7.1.9 generateProblem7

generateProblem7(self, clause_length, clauses_num, poisson)

- **clause_length** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **poisson** - zmienna logiczna, która określa, czy generator ma korzystać z rozkładu Poissona.

Metoda ta, w zależności od wartości parametru *poisson*, trzykrotnie uruchamia metodę *generateProblem1()* lub *generateProblem2()*, za każdym razem z inną wartością parametru odpowiedzialnego za formułę docelową. Dodatkowo, wywołuje metodę *generateFormulaR()*, żeby wygenerować czwartą, krótszą formułę, oznaczaną w opisie problemu jako R.

7.1.10 generateProblem8

generateProblem7(self, clause_length, clauses_num, poisson)

- **clause_length** - lista długości klauzul, wyrażonych w ilości literałów.
- **clauses_num** - ilość klauzul, czyli długość formuły.
- **poisson** - zmienna logiczna, która określa, czy generator ma korzystać z rozkładu Poissona.

Metoda ta, w zależności od wartości parametru *poisson*, dwukrotnie uruchamia metodę *generateProblem1()* lub *generateProblem2()*, za pierwszym razem z wartością parametru odpowiedzialnego za formułę docelową równą 1, a za drugim 2.

7.1.11 generateFormulaR

generateFormulaR(self)

Formuła R składa się wyłącznie z jednej klauzuli żywotności w postaci ogólnej o długości 4, jednak, ponieważ klauzule żywotności o długości większej od 1 są automatycznie generowane w postaci warunkowej, metoda ta wywołuje metodę *generateSafetyClause()*, a następnie zmienia w wygenerowanej klauzuli pierwszy token, reprezentujący kwantyfikator, z FORALL na EXISTS, gdyż na tym etapie jest to jedyna różnica między tymi typami klauzul.

7.1.12 generateSafetyClause

generateSafetyClause(self, clause_length)

- **clause_length** - ilość atomów, z których ma zostać złożona klauzula.

Ta metoda wywołuje metodę *generateClause*, przekazując jej żadaną długość klauzuli, a następnie do zwróconego wyniku dołącza, na początek listy, token FORALL. Metoda zwraca gotową klauzulę bezpieczeństwa.

7.1.13 generateLivenessClause

generateLivenessClause(self, clause_length)

- **clause_length** - ilość atomów, z których ma zostać złożona klauzula.

Ta metoda wywołuje metodę *generateClause*, przekazując jej żadaną długość klauzuli, a następnie, jeśli długość ta była większa od 1, zamienia zwróconą klauzulę na postać warunkową poprzez wywołanie metody *replaceORwithIMP()*. Na koniec, do otrzymanej klauzuli dołącza, na początek listy, token EXISTS. Metoda zwraca gotową klauzulę żywotności.

7.1.14 generateClause

generateClause(self, length)

- **length** - ilość atomów, z których ma zostać złożona klauzula.

W metodzie tej tworzone są dwie listy: literałów oraz relacji, które są następnie łączone naprzemiennie przy użyciu funkcji *zipToList()* z modułu *utils*. Lista literałów jest tworzona poprzez wywołanie metody *getRandomAtomList()*, a lista relacji poprzez wywoływanie w pętli metody *getRandomRelation()*. Metoda zwraca klauzulę nie zawierającą kwantyfikatorów.

7.1.15 getRandomAtomList

getRandomAtomList(self, length)

- **length** - ilość atomów, z których ma zostać złożona klauzula.

Wynikiem działania tej metody jest lista tokenów typu ATOM, składających się z typu tokenu, nazwy atomu oraz flagi informującej o negacji. Na początku tworzona jest lista dostępnych atomów, na podstawie kluczy słownika wszystkich atomów. Aby utworzyć token, w pierwszej kolejności z listy dostępnych atomów losowany jest jeden, który jest z niej następnie usuwany, żeby uniknąć powtórzeń, jego nazwa jest zapamiętywana w tokenie, a jego ilość wystąpień jest zwiększana o 1 w słowniku. Wartość flagi negacji jest losowana, a utworzony token zostaje dodany do listy wynikowej.

7.1.16 getRandomRelation

getRandomRelation(self)

Ponieważ metoda ta była tworzona z zamysłem bardziej uniwersalnego zastosowania, losuje ona typ zwracanego tokenu z listy relacji klasy LogicToken. Lista ta, przy obecnym projekcie problemów, zawiera jednak jedynie element OR, więc w praktyce metoda każdorazowo zwraca token typu OR.

7.1.17 replaceORwithIMP

replaceORwithIMP(self, clause)

- **clause** - klauzula, w której ma zostać zmieniony token.

Metoda ta losuje z podanej jako argument klauzuli żywotności jeden token typu OR i zastępuje go tokenem typu IMP, co powoduje, że klauzula z postaci ogólnej zostaje zamieniona na postać warunkową, o losowej długości poprzednika i następnika.

7.1.18 cleanup

cleanup(self, precentage_of_saftey_clauses)

- **precentage_of_saftey_clauses** - procent, jaki w formule stanowią klauzule bezpieczeństwa (liczba całkowita).

Po wygenerowaniu całej formuły, konieczne jest zastosowanie dwóch korekt.

Pierwsza z nich służy zapewnieniu, że wszystkie atomy zostaną wykorzystane (nie ma to zastosowania w przypadku problemu 3, ponieważ wygenerowane w nim formuły mają liczbę atomów większą od sumarycznej długości klauzul). Odbywa się to poprzez iteracyjne wyszukiwanie w słowniku atomów takich kluczy, które mają ilość wystąpień w formule równą 0. Jeśli taki atom zostanie znaleziony, to, przy pomocy metody pomocniczej *getMostFrequentKey()* wyszukiwany jest atom o największej ilości wystąpień i jego pierwsze wystąpienie jest zastępowane brakującym atomem, a liczby wystąpień w słowniku są odpowiednio modyfikowane.

Druga koryguje niezgodność stosunku ilości klauzul bezpieczeństwa i żywotności, powstałą na skutek zaokrągleń. W przypadku, gdy wygenerowane zostało zbyt dużo klauzul bezpieczeństwa, odpowiednia ilość ostatnich z nich zostanie zamieniona na klauzule żywotności poprzez zmianę kwantyfikatora i ewentualne wstawienie implikacji.

7.1.19 getMostFrequentKey

getMostFrequentKey(self)

Metoda ta przeszukuje słownik wszystkich atomów i zwraca atom o największej ilości wystąpień w formule. Jeśli najczęściej występujący atom występuje tylko raz, oznacza to, że jest zbyt dużo atomów i, jeżeli nie jest to formuła reprezentująca problem 3, rzucony zostaje wyjątek.

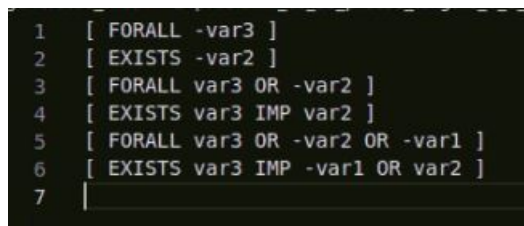
7.1.20 saveFormulas

saveFormulas(self)

W celach diagnostycznych oraz aby umożliwić użycie programu z innymi niż przewidziane proverami poprzez stworzenie jedynie nowego translatora, metoda *saveFormulas()* zapisuje wszystkie wygenerowane formuły do plików .txt. Formuły są zapisywane w formacie bezpośrednio reprezentującym zawartość listy klauzul:

- Formuła jest zapisywana w postaci Clause Normal Form - każda linia zawiera jedną klauzulę.
- Każda linia składa się z nawiasu otwierającego '[', ciągu tokenów oraz nawiasu zamykającego ']'. Wszystkie elementy są rozdzielone spacjami.
- Tokeny typu ATOM są reprezentowane poprzez nazwę atomu, poprzedzoną znakiem '-', jeśli flaga negacji była ustawiona.
- Tokeny pozostałych typów reprezentowane są przez nazwę typu.

Zapis przykładowej wygenerowanej formuły przedstawiono na rysunku [Figure 2]. Odczytanie tak zapisanej formuły umożliwia funkcja *readFormulaFile()* z modułu utils.



```
1 [ FORALL -var3 ]
2 [ EXISTS -var2 ]
3 [ FORALL var3 OR -var2 ]
4 [ EXISTS var3 IMP var2 ]
5 [ FORALL var3 OR -var2 OR -var1 ]
6 [ EXISTS var3 IMP -var1 OR var2 ]
7
```

Rysunek 2: Zapis przykładowej wygenerowanej formuły. Formuła reprezentuje problem1, ma 6 klauzul, a co za tym idzie, 3 atomy, długości klauzul to 1, 2 oraz 3, klauzul bezpieczeństwa jest tyle samo, co żywotności. Parametry takie zostały wybrane, ponieważ pozwalają na równy podział ilości klauzul i pokazanie wszystkich ich typów, a ponadto wygenerowana formuła jest krótka i czytelna. W rzeczywistości formuły są o wiele bardziej rozbudowane.

7.1.21 printFormula

printFormula(self, source=//)

- **source** - formuła, która ma zostać wypisana.

Metoda wypisuje do konsoli wybraną formułę w formacie analogicznym, jak w metodzie *saveFormulas()*. Jeśli żadna formuła nie została podana to wypisywana jest *self.formula*.

7.1.22 translateAndSave

translateAndSave(self)

W przypadku większości typów problemów, działanie tej metody sprowadza się do wywołania kolejno metody *translateToProver9()* i metody *translateToSPASS()*, jednak gdy typ problemu to jeden z podtypów problemu 7 lub 8, to, z uwagi na fakt, że formuła zostaje wtedy wygenerowana jako kilka osobnych

podformuł, metoda ta tłumaczy podformuły do oddzielnych plików, nie zawierających części formalnych danego formatu, a następnie tworzy wzorzec, według którego pliki zostaną połączone i uruchamia odpowiednią dla każdego provera metodę `join{nazwa_provera}WithPattern()`. Gdy tłumaczenie się zakończy, zwrócony zostaje słownik, przyporządkowujący kluczom "prover9_input", "spass_input" oraz "output" odpowiednie nazwy utworzonych (lub czekających na utworzenie, w przypadku output) plików.

7.1.23 translateToProver9

translateToProver9(self, raw=False, file_name_appendix="_", source_formula=[])

- **raw** - flaga informująca, czy formuła ma być tłumaczona na surowy format, nie zawierający nagłówków, stopek i innych elementów formatu prover9.
- **file_name_appendix** - człon wstawiany w nazwie pliku wyjściowego pomiędzy parametry formuły, a nazwę prover9.
- **source_formula** - formuła, która ma zostać przetłumaczona.

Pierwszym, a zarazem najważniejszym etapem działania metody jest utworzenie listy klauzul, w formie stringów, w formacie Prover9, na podstawie wybranej, wygenerowanej przez generator formuły (jeśli nie została wybrana, to tłumaczona jest formuła `self.formula`). W tym celu, dla każdej klauzuli w formule, tworzone są trzy listy: przechowująca elementy związane z kwantyfikatorami (w skrócie: lista kwantyfikatorów), przechowująca literały po lewej stronie ewentualnej implikacji (w skrócie: lewa lista) oraz przechowująca literały po jej prawej stronie (w skrócie: prawa lista), zerowana jest flaga informująca o występowaniu implikacji, przetwarzany jest każdy token w klauzuli, a następnie wszystkie trzy listy są odpowiednio łączone.

Przetwarzanie tokenów przebiega następująco:

- Tokeny typu FORALL oraz EXISTS są tłumaczone na odpowiednie symbole kwantyfikatorów, a następnie, wraz z symbolem zmiennej 'u', tłumaczenie to dodawane jest do listy kwantyfikatorów.
- Tokeny typu OR powodują dodanie symbolu dysjunkcji do prawej listy, jeśli flaga informująca o implikacji jest ustawiona lub do lewej listy, jeśli flaga jest wyzerowana.
- Tokeny typu ATOM są konwertowane na string, następnie pierwsza litera w ich nazwie zamieniana jest na wielką literę ('v' na 'V'), a na ich koniec doklejany jest string "(u)". Tak przygotowany napis dodawany jest do prawej listy, jeśli flaga informująca o implikacji jest ustawiona lub do lewej listy, jeśli flaga jest wyzerowana.
- Wystąpienie tokenu typu IMP powoduje dodanie do listy kwantyfikatorów symbolu kwantyfikatora egzystencjalnego oraz symbolu zmiennej 'u1' (ponieważ jednoznacznie określa, że przetwarzana klauzula jest klauzulą żywotności w postaci warunkowej) oraz ustawienie flagi informującej o implikacji.

Jeśli, po przetworzeniu wszystkich tokenów w klauzuli, flaga implikacji jest ustawiona, to pierwszy element listy kwantyfikatorów zostaje zamieniony na kwantyfikator uniwersalny - dzieje się to w sytuacji, gdy najpierw token EXISTS spowodował dodanie jednego kwantyfikatora egzystencjalnego, a potem algorytm, na podstawie tokenu IMP, dodał drugi kwantyfikator egzystencjalny - pierwszy z nich zostaje zmieniony na uniwersalny, aby utworzyć postać warunkową formuły żywotności.

Łączenie list rozpoczyna się od zespojenia listy kwantyfikatorów w jeden napis, poprzedziany spacjami. Następnie sprawdzany jest rodzaj klauzuli na podstawie flagi implikacji oraz pierwszego elementu listy kwantyfikatorów. W przypadku klauzuli żywotności w postaci warunkowej, w lewej liście wszystkie wystąpienia zmiennej 'u' zamieniane są na 'u1', po czym do zaczętego wcześniej napisu dodawana jest koniunkcja warunku "(u >= u1)" z implikacją, w której poprzednikiem jest zespojona lewa lista, a następnikiem zespojona prawa lista. Dla innych klauzul jedynym działaniem jest dołączenie do napisu zespolonej lewej listy - zanegowanej w przypadku klauzuli bezpieczeństwa i niezanegowanej w przypadku klauzuli żywotności w postaci ogólnej. Jeśli formuła nie jest tłumaczona w formacie surowym, to na koniec klauzuli wstawiana jest kropka. Tak przygotowane tłumaczenie klauzuli dodawane jest do listy klauzul.

Kolejnym etapem działania metody jest wygenerowanie ścieżki do pliku wyjściowego na podstawie przekazanych do niej parametrów. Będąc w posiadaniu ścieżki, program może otworzyć plik i przystąpić do zapisywania formuły. Domyślnie na początek pliku wstawiane jest ograniczenie pamięci oraz otwarcie formuły, a na koniec domknięcie formuły, które zostały opisane wcześniej, a sama lista klauzul jest zapisywana ze znakiem nowej linii jako separatorem. W przypadku zapisu w surowym formacie zapisywana jest jedynie zawartość listy klauzul, w jednej linii, rozdzielona znakami koniunkcji i odpowiednio obłożona nawiasami, co ma umożliwić późniejsze złączenie kilku takich formuł dowolnymi operatorami. W takim wypadku formuła przyjmuje postać Conjunctive Normal Form zamiast Clause Normal Form.

```

1  assign(max_megs, 1024).
2
3  formulas(sos).
4
5  all u (-(-Var3(u))).
6  exists u (-Var2(u)).
7  all u (- (Var3(u) | -Var2(u))).
8  all u exists u1 ((u >= u1) & ((Var3(u1)) -> (Var2(u)))).
9  all u (- (Var3(u) | -Var2(u) | -Var1(u))).
10 all u exists u1 ((u >= u1) & ((Var3(u1)) -> (-Var1(u) | Var2(u)))).
11
12 end_of_list.
```

Rysunek 3: Przykładowa formuła przetłumaczona na format Prover9.

7.1.24 translateToSPASS

translateToSPASS(self, raw=False, file_name_appendix="_", source_formula=[])

- **raw** - flaga informująca, czy formuła ma być tłumaczona na surowy format, nie zawierający nagłówków, stopek i innych elementów formatu prover9.
- **file_name_appendix** - człon wstawiany w nazwie pliku wyjściowego pomiędzy parametry formuły, a nazwę prover9.
- **source_formula** - formuła, która ma zostać przetłumaczona.

Ze względu na prefiksowy zapis operatorów oraz bardziej rozbudowane metadane, metoda *translateToSPASS()* jest bardziej złożona, jednak jej główna część to, podobnie jak przy tłumaczeniu na format Prover9, pętla przechodząca po wszystkich klauzulach w wygenerowanej formule. W trakcie tłumaczenia na format SPASS, zapamiętywanych jest jednak więcej danych: flaga informująca o wystąpieniu

implikacji - analogiczna jak w Prover9, lista wszystkich atomów - będzie budowana w trakcie analizy tokenów, kolejka - gromadząca krotki łączące tokeny według zakresów oddziaływania, lista atomów - służąca do budowy krotek, lista atomów powiązanych dysjunkcją - służąca do budowy krotek dla tokenu typu OR, pierwsza krotka - przechowująca informacje o kwantyfikatorze, krotka implikacji - przechowująca informacje o implikacji. Dla każdej klauzuli następuje kolejno: analiza tokenów, dopełnienie kolejki, analiza kolejki i dopisanie wyniku do listy przetłumaczonych klauzul.

Podczas analizy tokenów rozpatrywany jest każdy token w bieżącej klauzuli:

- Tokeny typu FORALL oraz EXISTS są zapisywane w pierwszej krotce - jako pierwszy element, drugim jest pusta lista.
- Tokeny typu OR powodują, jeśli lista atomów połączonych dysjunkcją jest pusta, przypisanie do niej listy atomów.
- Tokeny typu ATOM są dodawane do listy atomów, a ich nazwy do listy wszystkich atomów.
- Wystąpienie tokenu typu IMP powoduje wpisanie go do krotki implementacji (z pustą listą jako drugim elementem), ustawienie flagi informującej o implikacji oraz wyczyszczenie list - do kolejki dodawana jest krotka zawierająca typ OR oraz listę atomów połączonych dysjunkcją, jeśli ta lista nie była pusta (czyli jeśli po lewej stronie implikacji występowała dysjunkcja), w przeciwnym wypadku, krotka zawierająca typ ATOM oraz listę atomów, jeśli ta nie była również pusta (czyli jeśli po lewej stronie implikacji występował pojedynczy literal), a następnie lista atomów oraz lista atomów połączonych dysjunkcją zostają opróżnione.

Po zakończeniu analizy tokenów następuje dodanie do kolejki krotki zawierającej typ OR i listę atomów, jeśli lista atomów połączonych dysjunkcją nie jest pusta (czyli jeśli wystąpiła dysjunkcja, a nie wystąpiła implikacja lub jeśli po prawej stronie implikacji wystąpiła dysjunkcja), w przeciwnym wypadku dodana zostanie krotka zawierająca typ ATOM i listę atomów, jeśli ta lista nie jest pusta (czyli jest to przypadek, gdy nie wystąpiła ani dysjunkcja, ani implikacja - klauzula jest długości 1 lub gdy po prawej stronie implikacji wystąpił tylko pojedynczy literal). Kolejnym krokiem jest dopełnienie kolejki - jeśli flaga implikacji jest ustawiona, to jako obecna zmienna zostaje ustawiona "(T1)", a krotka implementacji zostaje dodana do kolejki, jeśli flaga jest wyzerowana, to jako obecna zmienna zostaje ustawiona "(T)". Niezależnie od wartości flagi, następuje dodanie pierwszej krotki do kolejki.

Następnie rozpoczyna się budowa wyniku tłumaczenia klauzuli. W pętli analizowana jest każda krotka w kolejce:

- Krotki o typie ATOM powinny w liście zawierać pojedynczy token typu ATOM. Zostaje on przetłumaczony przy użyciu funkcji pomocniczej, która przyłącza do jego nazwy obecną zmienną oraz, jeśli ma on być zanegowany, wstawia go do operatora "not()". Tak przygotowane tłumaczenie zostaje dopisane do wyniku, a obecna zmienna zostaje zmieniona (z "(T1)" na "(T)" lub odwrotnie), ponieważ krotka ta stanowiła całe zdanie logiczne.
- Krotki typu OR powinny zawierać w liście więcej niż 1 token typu ATOM. Każdy z nich zostaje przetłumaczony funkcją pomocniczą, a następnie lista zostaje zespojona przy użyciu separatora ";", a wynik tego działania zostaje zamknięty w operatorze "or()". Całość zostaje dopisana do wyniku, a obecna zmienna zostaje zmieniona, podobnie, jak w przypadku krotki typu ATOM.
- Krotka typu IMP powoduje otoczenie dotychczasowego wyniku operatorem "implies()".
- Krotki typu FORALL oraz EXISTS powodują dopisanie kwantyfikatorów na początek wyniku. Jeśli w kolejce występuje krotka typu IMP, oznacza to, że tłumaczona jest klauzula żywołności w postaci warunkowej i dotychczasowy wynik zostanie rozszerzony w następujący sposób:

"forall([T], exists([T1], and(GTE(T, T1), {dotychczasowy_wynik})))". Jeśli nie, to jeśli krotka jest typu EXISTS, przetwarzana klauzula jest klauzulą żywotności w postaci ogólnej i wynik zostanie rozszerzony według wzoru: "exists([T], {dotychczasowy_wynik})". Jeśli zaś krotka jest typu FORALL, to klauzula jest klauzulą bezpieczeństwa, a więc wynik zostanie rozszerzony następująco: "forall([T], not({dotychczasowy_wynik}))".

Po przeanalizowaniu całej kolejki, przetłumaczony wynik jest gotowy i zostaje dodany do listy tłumaczeń klauzul.

Oprócz listy klauzul, tworzone są również lista opisu problemu, lista symboli oraz ścieżka do pliku. Lista opisu symboli jest zawsze taka sama, z uwzględnieniem nazwy pliku. Lista symboli składa się z listy funkcji i listy predykatów. Lista funkcji zawiera stałe "(t1,0), (t2,0)", jeśli w formule wystąpiła implikacja, w przeciwnym razie w ogóle nie występuje. Lista predykatów jest tworzona na podstawie listy wszystkich atomów, poprzez dodanie do każdego liczby argumentów: 1 oraz dołożenie predykatu "(GTE, 2)", jeśli w formule wystąpiła implikacja. Elementy listy klauzul zostają ujęte w operatory "formula().", a do całej listy zostaje dodany nagłówek i stopka, jeśli formuła nie jest tłumaczona w formacie surowym. Jeśli tak by było, to pozostanie niezmieniona.

Na koniec, jeśli formuła tłumaczona jest na format surowy, to do pliku zostaje zapisana zawartość listy klauzul, ujęta w operator "and()" - w postaci Conjunctive Normal Form zamiast Clause Normal Form. W innym wypadku zapisane zostają po kolei wszystkie listy, a sama lista klauzul jest porozielana znakami nowej linii. Zwrócona zostaje ścieżka do zapisanego pliku.

```

1  begin_problem(problem1_c6_a3_prec50_lengths_1_2_3).
2
3  list_of_descriptions.
4  name({*problem1_c6_a3_prec50_lengths_1_2_3*}).
5  author({*Jakub Semczyszyn*}).
6  status(unknown).
7  description({*The problem was generated randomly*}).
8  end_of_list.
9
10 list_of_symbols.
11 functions[(t1,0), (t2,0)].
12 predicates[(var3, 1), (var2, 1), (var1, 1), (GTE, 2)].
13 end_of_list.
14
15 list_of_formulae(axioms).
16 formula(GTE(t2,t1)).
17 formula(forall([T], not(not(var3(T))))).
18 formula(exists([T], not(var2(T)))).
19 formula(forall([T], not(or(var3(T), not(var2(T)))))).
20 formula(forall([T], exists([T1], and(GTE(T, T1), implies(var3(T1), var2(T)))))).
21 formula(forall([T], exists([T1], and(GTE(T, T1), or(var3(T), not(var2(T)), not(var1(T)))))).
22 formula(forall([T], exists([T1], and(GTE(T, T1), implies(var3(T1), or(not(var1(T)), var2(T)))))).
23 end_of_list.
24
25 end_problem.

```

Rysunek 4: Przykładowa formuła przetłumaczona na format SPASS.

7.1.25 joinProver9FilesWithPattern

joinProver9FilesWithPattern(self, files_list, pattern, remove=True)

- **files_list** - lista ścieżek do plików z tłumaczeniami poszczególnych formuł składowych.
- **pattern** - lista znaczników, określających, jakimi operatorami mają zostać połączone formuły.

- **remove** - flaga informująca, czy pliki z formułami składowymi mają zostać usunięte po utworzeniu zespolonego pliku.

Pierwszym krokiem działania metody jest otworenie kolejno wszystkich plików z listy i zaczytanie ich zawartości do listy zawartości plików. Pliki, które są poddawane łączeniu powinny być w formacie surowym, to znaczy składać się z jednej linii, zawierającej klauzulę połączone koniunkcją. Następnie analizowany jest wzorzec, przy jednoczesnym tworzeniu listy z wynikową formułą:

- Element typu FILE* powoduje dodanie do listy wynikowej zawartości pliku, będącej w liście zawartości pod indeksem równym numerowi pliku (zastępującemu *).
- Element typu NOT powoduje dodanie do listy wynikowej znaku '¬'.
- Element typu LP powoduje dodanie do listy wynikowej znaku '('.
- Element typu RP powoduje dodanie do listy wynikowej znaku ')'
- Element typu OR powoduje dodanie do listy wynikowej ciągu " | ".
- Element typu AND powoduje dodanie do listy wynikowej ciągu " & ".

Na koniec listy wynikowej dodana zostaje kropka. Do nowego pliku zostają zapisane nagłówki, zespolona lista wynikowa oraz stopka. Jeśli parametr *remove* nie był wyzerowany, to pliki ze składowymi formułami zostają usunięte. Zwrócona zostaje ścieżka do nowo powstałego pliku.

7.1.26 joinSPASSFilesWithPattern

joinSPASSFilesWithPattern(self, files_list, pattern, remove=True)

- **files_list** - lista ścieżek do plików z tłumaczeniami poszczególnych formuł składowych.
- **pattern** - lista znaczników, określających, jakimi operatorami mają zostać połączone formuły.
- **remove** - flaga informująca, czy pliki z formułami składowymi mają zostać usunięte po utworzeniu zespolonego pliku.

Pierwszym krokiem działania metody jest otworenie kolejno wszystkich plików z listy i zaczytanie ich zawartości do listy zawartości plików. Pliki te powinny być w formacie surowym, czyli zawierać jedną linię z pojedynczą koniunkcją wszystkich klauzul. Generowanie listy opisu oraz listy symboli przebiega podobnie, jak w metodzie tłumaczącej na format SPASS Provera. Z uwagi na prefiksowy zapis operatorów, analiza wzorca i generowanie wyniku przebiega w sposób bardziej rozbudowany, niż w przypadku Prover9. Tworzone są zmienne pomocnicze: flaga informująca o konieczności wstawienia nawiasu zamykającego (w skrócie: flaga nawiasu) - początkowo wyzerowana, stos zawierający indeksy początków zakresów w liście wynikowej - początkowo zawiera 0, reprezentujące początek najbardziej zewnętrznego zakresu, obejmującego całą formułę (w skrócie: stos zakresów), liczba zapamiętanych nawiasów zamykających, stos operatorów - początkowo pusty. Analiza wzorca przebiega w następujący sposób:

- Element typu FILE* powoduje dodanie do listy wynikowej zawartości pliku, będącej w liście zawartości pod indeksem równym numerowi pliku (zastępującemu *). Dodatkowo, jeśli flaga nawiasu jest ustawiona, oznacza to, że zakres tego pliku zaczyna się od poprzedzającego elementu NOT, nie ma więc potrzeby dodawać kolejnego indeksu do stosu zakresów. W takim wypadku do listy wynikowej dopisywany jest znak ')', a flaga nawiasu jest zerowana. Jeśli flaga była wyzerowana, to zamiast tego bieżący indeks jest dodawany do stosu zakresów, jako początek zakresu tego pliku.

- Element typu NOT powoduje dodanie ciągu "not(" do listy wynikowej. Jeśli następny element wzorca to LP, to przy jego obsłudze zostanie dodany zakres, a więc jedynym działaniem jest zwiększenie licznika zapamiętanych nawiasów, aby później dodać nawias zamykający, odpowiadający dodanemu nawiasowi otwierającemu. Jeśli następny element nie jest typu LP, znaczy to że NOT odnosi się do pojedynczej zmiennej w formule, więc na stos zapisany zostaje początek zakresu zanegowanej zmiennej i flaga nawiasu zostaje ustawiona.
- Ponieważ wszystkie nawiasy otwierające są dodawane wraz z predykatami, to nie ma potrzeby, aby wstawiać je dodatkowo, wystąpienie elementu LP powoduje więc jedynie dodanie początku zakresu na stos. W przypadku gdy jest to sam początek formuły i lista wynikowa jest pusta, wstawione zostaje kolejne 0, natomiast gdy lista nie jest pusta, to zostaje dodany indeks ostatniego elementu, czyli odpowiednik miejsca wystąpienia nawiasu we wzorcu.
- Wystąpienie elementu RP oznacza, że należy domknąć wszystkie oczekujące zakresy. Do listy wynikowej dodawane jest tyle znaków ')', ile wynosi wartość licznika zapamiętanych nawiasów (po czym jest ona zerowana) oraz tyle samo elementów jest usuwanych ze stosu zakresów. Dodatkowo usuwany jest jeszcze jeden element ze stosu zakresów - odpowiadający elementowi LP stanowiącemu parę z przetwarzanym elementem RP oraz, jeśli stos operatorów jest niepusty, jeden element ze stosu operatorów, ponieważ jego zasięg działania został przed chwilą zamknięty.
- Elementy OR i AND mogą powodować dwa różne działania. Jeśli przetwarzany element jest tego samego typu, co wierzchołek stosu operatorów, oznacza to, że jest to dalszy ciąg tej samej dysjunkcji lub koniunkcji (na przykład drugi OR we fragmencie wzorca [FILE0, OR, FILE1, OR, FILE2]), a zatem do listy wynikowej dodawany jest znak ',' rozdzielający argumenty predykatu `or()` lub `and()`. W innym wypadku rozpoczyna się nowa koniunkcja lub dysjunkcja, a więc odpowiedni operator jest dodawany na stos operatorów, a ciąg "and(" lub "or(" jest wstawiany na początek ostatniego zakresu, czyli na miejsce w liście wynikowej o indeksie równym wierzchołkowi stosu zakresów. Na stos zakresów zostaje wstawiony początek zakresu nowego operatora, a początki zakresów występujących po nim są odpowiednio inkrementowane. Na koniec listy wynikowej dodany zostaje znak ',' i zwiększona o 1 zostaje liczba zapamiętanych nawiasów. Niezależnie od podjętych dotychczas działań, na końcu przetwarzania elementów OR i AND sprawdzone zostaje, czy wierzchołek stosu zakresów jest równy przedostatniemu indeksowi w liście końcowej, jeśli tak by było, to można go usunąć ze stosu, ponieważ został już wykorzystany przy wstawianiu operatora i nie będzie więcej potrzebny - jego zasięg się skończył.

Po przetworzeniu całego wzorca może się okazać, że pozostały jakieś zapamiętane nawiasy zamykające, są one wtedy wstawiane na koniec formuły. Formuła wraz ze wszystkimi elementami formatu SPASS jest zapisywana do nowego pliku, jeśli parametr *remove* nie był wyzerowany, to pliki ze składowymi formułami zostają usunięte. Zwrócona zostaje ścieżka do nowo powstałego pliku.

7.1.27 `__getAtomStr`

`__getAtomStr(self, token, time_str)`

- **token** - token typu ATOM, który ma zostać przetłumaczony na format SPASS.
- **time_str** - zmienna (w nawiasach), która ma zostać dodana do atomu, aby utworzyć predykat.

Metoda ta zwraca nazwę atomu z doklejoną na koniec podaną zmienną. Jeśli atom miał być zanegowany, to całość dodatkowo umieszczana jest wewnątrz operatora "not()".

7.2 Moduł provrun

Uruchamianie systemów dowodzenia twierdzeń oraz pobieranie ich wyników jest zadaniem klasy `ProverRunner`. Jej działanie ma charakter sekwencyjny: użytkownik tworzy jej instancję konstruktorem, następnie wywołuje metodę `performMeasurements()`, która z kolei wywołuje metodę `runProver()`, a gdy ta się powiedzie, zwraca wynik wywołania metody `getMeasuresFromOutputFile()`.

Do klasy `ProverRunner` należą metody:

7.2.1 `__init__`

`__init__(self, prover, input_file)`

- **prover** - nazwa provera ("prover9" lub "spass").
- **input_file** - ścieżka do pliku wejściowego dla provera.

Konstruktor klasy `ProverRunner` jest prosty - oprócz utworzenia instancji, jedynie zapisuje przekazane mu parametry.

7.2.2 `performMeasurements`

`performMeasurements(self, output_file)`

- **output_file** - ścieżka, pod którą ma zostać utworzony plik wynikowy provera.

Metoda ta stanowi warstwę pomiędzy kodem użytkownika, a metodami klasy. Wywołuje w bloku *try* metodę `runProver`, przekazując jej swój parametr, zawierający docelową ścieżkę do pliku wyjściowego i wylapuje ewentualne błędy związane z przekroczeniem limitu czasu działania procesu provera - `TimeoutExpired` z pakietu `subprocess`. Gdy błąd zostanie złapany, rzucony zostanie kolejny, tym razem klasy `RFGTimeoutError`, co pozwala na łatwiejsze obsłużenie takiego błędu przez środowisko testowe. W innym wypadku metoda ta wywoła metodę `getMeasuresFromOutputFile` i zwróci jej wynik użytkownikowi.

7.2.3 `runProver`

`runProver(self, output_file=None, time_limit=300)`

- **output_file** - ścieżka, pod którą ma zostać utworzony plik wynikowy provera.
- **time_limit** - ilość sekund, po jakiej praca provera ma zostać przerwana.

Metoda zapisuje przekazany jej parametr ze ścieżką do pliku w polu klasy (jeśli żaden nie został przekazany to plik zostanie zapisany pod nazwą "prover_output.out" w lokalizacji modułu provrun), a następnie buduje komendę terminala, służącą do uruchomienia systemu dowodzenia twierdzeń odpowiadającego wartości pola `prover` i uruchamia ją w nowym procesie przy pomocy funkcji `subprocess.call`. Komendy dla obu proverów zawierają limit czasowy. Limit ten został zaimplementowany poprzez wykorzystanie odpowiednich parametrów tych programów. W przypadku SPASS Provera użyty został parametr `TimeLimit`, który powoduje, że, podczas wybierania każdej kolejnej klauzuli do wnioskowania, sprawdzany jest czas pracy systemu dowodzenia. Takie podejście sprawia, że prover może przekroczyć zadany czas o najwyżej kilka sekund. Prover9 udostępnia natomiast parametr `-t`, który

powoduje ograniczenie czasu poszukiwania rozwiązań. Nie ma jednak wpływu na czas wnioskowania, a to niejednokrotnie, przy dużych formułach, potrafi zająć kilkanaście lub nawet kilkadziesiąt minut. Konieczne zatem było wdrożenie drugiego ograniczenia czasowego, poprzez timeout nowo utworzonego procesu. Takie przerwanie sprawia, że rzucony zostaje wyjątek, którego obsługa została nakreślona w opisie poprzedniej metody.

Metoda ta może być wykorzystywana samodzielnie, niezależnie od reszty programu, jako interfejs do uruchamiania proverów z limitem czasowym, przekazywanym przy pomocy parametru *time_limit*.

7.2.4 getMeasuresFromOutputFile

getMeasuresFromOutputFile(self)

Jako że, zarówno Prover9, jak i SPASS Prover, umieszczają blok ze statystykami na końcu danych wynikowych, a jego elementy występują w ściśle, z góry określonej kolejności, w tej metodzie do odczytu plików wyjściowych użyto maszyny stanów.

Dla przyspieszenia przeszukiwania, algorytm sprawdza jedynie od 1 do 5 pierwszych znaków każdej linii pliku i odrzuca wszystkie linie, które rozpoczynają się innymi znakami, niż te szukane.

W nieodrzuconych liniach, w przypadku Prover9, w pierwszym stanie szukany jest nagłówek bloku STATISTICS. W kolejnym, odczytywany jest wynik działania systemu dowodzenia. Ponieważ polega ono na poszukiwaniu dowodu na nieprawdziwość zadanej formuły, to znalezienie go oznacza, że jest ona niespełnialna, a nieznanie, że jest spełnialna. Dodatkowo domyślne wywołanie Prover9 limituje ilość poszukiwanych dowodów do 1. Zatem liczba znalezionych dowodów równa 1 będzie ustawać zmienną informującą o spełnialności na wartość "False", a liczba 0 na wartość "True". Pewną komplikacją tego rozumowania jest fakt, że w przypadku przekroczenia czasu poszukiwania lub maksymalnej pamięci, liczba dowodów w raporcie również wynosi 0. Przekroczenie czasu poszukiwania bez przekroczenia czasu działania procesu jest jednak teoretycznie niemożliwe, a praktycznie występuje niesamowicie rzadko, w przypadku, gdy przerwanie procesu zostanie opóźnione przez inne działanie systemu operacyjnego i skutkuje ponownym uruchomieniem provera, co najprawdopodobniej spowoduje już przerwanie we właściwy sposób, więc przypadek ten nie zaburza odczytu. Przekroczenie maksymalnej pamięci jest natomiast sprawdzane w kolejnym stanie. Główne zadanie wykonywane w nim to odczytanie ilości wykorzystanych przez prover megabajtów pamięci, jeśli jednak wartość ta jest równa lub przekracza 1024, będące sztywno ustalonym limitem, oznacza to, że poszukiwania rozwiązań zostały przerwane przedwcześnie, a więc zerowa ilość znalezionych dowodów nie jest wywołana przez spełnialność formuły, tylko przez błąd, toteż wartość zmiennej informującej o spełnialności jest wtedy zmieniana na "Memory limit". W następnym stanie odczytywany jest wykorzystany czas procesora. Jeśli algorytm dotarł do tego momentu, oznacza to, że wszystkie potrzebne elementy zostały odnalezione w pliku wyjściowym i odczytane, zostaje więc ustawiony stan sukcesu i zakańczane jest przeszukiwanie pliku.

```

===== STATISTICS =====
Given=0. Generated=1. Kept=0. proofs=1.
Usable=0. Sos=0. Demods=0. Linbo=0. Disabled=456. Hints=0.
Kept_by_rule=0. Deleted_by_rule=0.
Forward_subsumed=0. Back_subsumed=0.
Sos_limit_deleted=0. Sos_displaced=0. Sos_removed=0.
New_demodulators=0 (0 lex), Back_demodulated=0. Back_unit_deleted=0.
Demod_attempts=0. Demod_rewrites=0.
Res_instance_prunes=0. Para_instance_prunes=0. Basic_paramod_prunes=0.
Nonunit_fsub_feature_tests=0. Nonunit_bsub_feature_tests=0.
Megabytes=0.75.
User CPU=0.04, System CPU=0.01, Wall clock=0.
===== end of statistics =====

```

Rysunek 5: Przykładowy blok ze statystykami z pliku wynikowego Prover9. Na niebiesko zaznaczono odczytywane przez algorytm wartości.

W przypadku SPASS, blok ze statystykami rozpoczyna się od napisu "SPASS beiseite: ", po którym następuje informacja o wyniku poszukiwań. Zadaniem pierwszego stanu będzie więc odczytanie tej informacji. Podobnie jak w przypadku Prover9, SPASS również próbuje dowieść nieprawdziwości formuły, więc wynik "Proof found." znaczyć będzie, że jest ona niespełnialna, a "Completion found.", że jest spełnialna. Gdy przekroczony zostanie maksymalny czas, to prover zwróci wynik "Ran out of time. SPASS was killed.", a wartość zmiennej określającej spełnialność zostanie ustawiona na "Timeout". Gdy przekroczony zostaje limit pamięci, SPASS Prover zgłasza krytyczny, wewnętrzny, niemożliwy do obsłużenia błąd, nie zapisując żadnych wyników, więc plik wyjściowy jest wtedy pusty. W kolejnych stanach odczytywana jest linia zawierająca informacje o wykorzystanej pamięci (SPASS podaje ją w kilobajtach, dla spójności danych konieczne jest więc przeliczenie wartości na megabajty), a następnie, ta z informacjami o wykorzystanym czasie. Jeśli wszystkie informacje udało się odczytać (a więc również pod warunkiem, że nie wystąpił błąd przekroczenia pamięci), to ustawiony zostaje stan sukcesu i odczytywanie pliku zostaje przerwane.

```
SPASS beiseite: Proof found.
Problem: /home/jakub/Documents/RFC/generated_files/problem1_c100_a50_prec50_lengths_2_3_4_6_8_10_spass.in
SPASS derived 0 clauses, backtracked 0 clauses, performed 0 splits and kept 48 clauses.
SPASS allocated 85611 KBytes.
SPASS spent    0:00:00.07 on the problem.
               0:00:00.02 for the input.
               0:00:00.02 for the FLOTTER CNF translation.
               0:00:00.00 for inferences.
               0:00:00.00 for the backtracking.
               0:00:00.00 for the reduction.
```

Rysunek 6: Przykładowy blok ze statystykami z pliku wynikowego SPASS Provera. Na niebiesko zaznaczono odczytywane przez algorytm wartości.

Jeżeli po zakończeniu przeszukiwania pliku bieżący stan to stan startowy SPASS Provera, to sprawdzane jest, czy plik wynikowy jest pusty, jeśli tak, to wynik jest ustawiany na "Memory limit", a stan jest zmieniany na sukces. Następnie, jeśli stan nie jest ustawiony na sukces, to rzucony zostaje błąd, informujący o nieodnalezieniu potrzebnych informacji. W przeciwnym razie, metoda zwraca słownik postaci: "{*"memory"*: total_memory, *"time"*: total_time, *"sat"*: sat}", gdzie: total_memory to ilość wykorzystanych przez prover megabajtów pamięci; total_time to liczba wykorzystanych sekund; sat to informacja o wyniku poszukiwań, której wartość "True" oznacza spełnialność, wartość "False" niespełnialność, wartość "Timeout" oznacza przekroczenie limitu czasu, a wartość "Memory limit" przekroczenie limitu pamięci.

7.3 Moduł testenv

Na moduł testenv, implementujący środowisko testowe, składają się trzy klasy: TestEnv, CaseMaker oraz Case.

7.3.1 Klasa Case

Służy do przechowywania listy parametrów pojedynczego przypadku testowego i przekazywania jej do konstruktora klasy Generator w celu utworzenia jej instancji.

```
__init__(self, parameters_list)
```

- **parameters_list** - lista parametrów formuły w reprezentowanym przypadku testowym.

Konstruktor zapisuje przekazaną mu listę.

createGenerator(self)

Metoda zwraca instancję klasy Generator, utworzoną przy pomocy listy parametrów.

7.3.2 Klasa CaseMaker

Klasa pomocnicza, której zadaniem jest tworzenie pojedynczych przypadków testowych na podstawie parametrów. Klasa posiada pola, odpowiadające wszystkim parametrom konstruktora klasy Generator.

makeCases(self)

Metoda ta iteruje po listach każdego z parametrów i, łącząc je w unikalne kombinacje, buduje obiekty klasy Case, umieszczając je w liście, którą na końcu zwraca.

7.3.3 Klasa TestEnv

Klasa ta jest trzonem środowiska testowego, odpowiedzialnym za przetworzenie i wykonanie scenariusza testowego.

__init__(self, config_file_path)

- **config_file_path** - ścieżka do pliku ze scenariuszem.

Konstruktor inicjuje listę przypadków testowych, a następnie wywołuje metodę *readTestConfigFile()*, odczytującą scenariusz z pliku.

readTestConfigFile(self, path)

- **path** - ścieżka do pliku ze scenariuszem.

Metoda czyta plik scenariusza linia po linii i jeżeli napotka linię z nazwą problemu, tworzy instancję klasy CaseMaker, a następnie zapisuje typ problemu i przechodzi w tryb odczytywania parametrów. W trybie tym szuka linii rozpoczynających się od nazw parametrów i zapisuje odczytane wartości parametrów w polach utworzonej wcześniej instancji klasy CaseMaker. Gdy napotka linię "end of problem", wywołuje metodę *makeCases()* klasy CaseMaker, otrzymane wyniki zapisuje w liście przypadków testowych, a następnie usuwa instancję klasy i wraca do poszukiwania linii z typem problemu.

makeTests(self)

Metoda rozpoczyna działanie od otwarcia dwóch plików: "error_log.txt" - służącego do logowania błędów programu oraz "test_session_results.csv" - w którym zapisywane będą wyniki pomiarów. W pierwszej kolejności, do pliku CSV wpisywany jest wiersz z nagłówkami kolumn. Są to kolejno:

- "Problem" - typ problemu.
- "Number of atoms" - liczba atomów użytych do wygenerowania formuły (w przypadku problemu 3 nie jest ona tożsama z liczbą atomów faktycznie występujących w formule).
- "Precentage of safety clauses" - procent klauzul, które były klauzulami bezpieczeństwa.
- "Clauses lengths" - lista długości klauzul w formule.
- "Number of clauses" - liczba klauzul w formule.

- "Distribution of lengths" - rozkład długości klauzul. Może mieć wartość "poisson", "even", "more_short" lub "more_long".
- "Satisfiability" - wynik działania systemów dowodzenia twierdzeń. Jeśli oba uznały formułę za niespełnialną, w kolumnie tej znajdzie się wartość "False", jeśli oba uznały ją za spełnialną, to będzie to wartość "True", jeśli wyniki będą się różnić między proverami to zgłoszony zostanie błąd. Ponadto wartości związane z przekroczeniem limitów mają pierwszeństwo przed wartościami określającymi spełnialność, to znaczy, że jeśli któryś z proverów przekroczy limit pamięci, w kolumnie pojawi się wartość "Memory limit", jeśli przekroczy limit czasu, pojawi się wartość "Timeout". Jeśli przekroczone zostaną oba limity, to zgłoszone zostanie przekroczenie limitu pamięci, ponieważ powoduje ono bardziej krytyczne błędy proverów.
- "Memory used by prover9" - pamięć wykorzystana przez Prover9, wyrażona w megabajtach.
- "Time used by prover9" - czas wykorzystany przez Prover9, wyrażony w sekundach.
- "Memory used by SPASS" - pamięć wykorzystana przez SPASS Prover, wyrażona w megabajtach.
- "Time used by SPASS" - czas wykorzystany przez SPASS Prover, wyrażony w sekundach.

Następnie metoda iteruje po wszystkich przypadkach testowych w zawierającej je liście i próbuje wywołać trzy bloki kodu, wyłapując ewentualne błędy po każdym z nich.

W pierwszym bloku wywoływana jest metoda *createGenerator()* obecnie przetwarzanej instancji klasy *Case*. Oznacza to, że na tym etapie wygenerowana i zapisana do pliku zostaje cała formuła. Następnie na utworzonej instancji klasy *Generator* wywoływana jest metoda *translateAndSave()*, która tłumaczy formułę na formaty proverów i zapisuje tłumaczenia w plikach, zwracając listę ścieżek do nich. Jeśli w trakcie tych operacji wystąpią błędy typu *RFGError*, to są wyłapywane i zapisywane w logu błędów, a program kontynuuje działanie. Jeśli błędy są innego typu, oznacza to, że miała miejsce krytyczna, nieprzewidziana sytuacja, błąd nie zostaje wyłapany i program kończy działanie.

Drugi blok zawiera wywołanie metody *runProver()* dla Prover9. Wywołanie to jest ujęte w osobny blok ze względu na możliwe wystąpienie błędu typu *RFGTimeoutError* w przypadku przekroczenia limitu czasu przez Prover9. Jeśli taki błąd zostanie złapany, to informacja o tym jest logowana, a wartości zwracane zostają "manualnie" ustawione na {"sat": "Timeout", "memory": 0, "time": 300}. Ponadto błędy typu *RFGError* są logowane, a pozostałe powodują przerwanie pracy programu.

Na ostatni blok składa się reszta metody, a więc kolejno: wywołanie metody *runProver()* dla SPASS Provera, porównanie wyników między systemami, ustalenie zawartości kolumny "Satisfiability", przygotowanie wiersza z wynikami, zapisanie wiersza do pliku CSV i wyczyszczenie procesów. Przy porównaniu wyników sprawdzane jest, czy nie nastąpiła sytuacja, w której jeden z proverów stwierdził, że formuła jest spełnialna, a drugi, że niespełnialna. Jeśli tak by się zdarzyło, to rzucony zostanie wyjątek zawierający informacje o wynikach obu z nich. Ustalanie zawartości kolumny "Satisfiability" zostało wyjaśnione w opisie tej kolumny. Przygotowanie wiersza do zapisu polega na zebraniu parametrów z obiektu klasy *Case* oraz wyników działania proverów i wpisaniu ich do listy. Po zapisaniu wiersza następuje czyszczenie procesów - jeśli któryś z systemów dowodzenia twierdzeń nie zakończyłby pracy we właściwy sposób ze względu na błędy, zostaje do niego wysłany sygnał zakończenia poprzez komendę *pskill*. Powstałe w trakcie wywoływania bloku błędy typu *RFGError* są wyłapywane i logowane do pliku, pozostałe powodują przerwanie pracy programu.

runProver(self, prover_name)

- **prover_name** - nazwa provera, który ma zostać uruchomiony ("prover9" lub "spass").

Metoda ta tworzy instancję klasy `ProverRunner` i trzykrotnie wywołuje jej metodę `performMeasurements()`, za każdym razem z inną nazwą pliku wyjściowego (do nazwy pliku dodawany jest napis `"attemptN"`, gdzie `N` to numer próby) i zapisuje kolejne wyniki w listach. Następnie sprawdzane jest, czy nie wystąpiły dwa różne wyniki - czy prover w różnych próbach nie uznał formuły za spełnialną i niespełnialną, jeśli tak by było, to rzucony zostaje błąd. Zwrócony zostaje słownik przypisujący: napisowi `"memory"` średnią z pomiarów pamięci, napisowi `"time"` średnią z pomiarów czasu i napisowi `"sat"` modalną z wyników provera.

7.4 Moduł `utils`

Moduł ten zawiera elementy użytkowe, wykorzystywane przez różne moduły: klasę `LogicToken`, funkcję `zipToList()` oraz funkcję `readFormulaFile()`.

7.4.1 Klasa `LogicToken`

Klasa ta reprezentuje używane w programie tokeny wyrażeń logicznych. Tokeny składają się z typu tokenu, a w przypadku typu `ATOM`, dodatkowo z nazwy atomu i flagi negacji. Dostępne typy specjalne to `EXISTS` - reprezentujący kwantyfikator egzystencjalny, `FORALL` - reprezentujący kwantyfikator uniwersalny, `ATOM` - reprezentujący literał, `IMP` - reprezentujący implikację, natomiast z typów relacyjnych dostępny jest jedynie `OR` - reprezentujący dysjunkcję. Klasa udostępnia konstruktor umożliwiający tworzenie tokenów oraz redefiniuje funkcję konwersji na string tak, aby tokeny typu `ATOM` były wypisywane jako nazwa atomu, poprzedzona znakiem `'-`', jeśli flaga negacji jest ustawiona, a pozostałe tokeny jako nazwa ich typu.

7.4.2 Funkcja `zipToList`

zipToList(list1, list2)

- **list1** - pierwsza z łączonych list - pierwszy jej element będzie pierwszym elementem złączonej listy.
- **list2** - druga z łączonych list - pierwszy jej element będzie drugim elementem złączonej listy.

Funkcja pozwala na złączenie dwóch list w jedną, zawierającą naprzemiennie elementy każdej z nich. Jeśli wejściowe listy nie są równej długości, to elementy są wstawiane naprzemiennie dopóki wystarcza elementów z krótszej listy, a następnie wstawiane są pozostałe elementy dłuższej listy.

7.4.3 Funkcja `readFormulaFile`

readFormulaFile(path)

- **path** - ścieżka do pliku z zapisaną formułą.

Funkcja działa odwrotnie do metody `saveFormulas()` klasy `Generator` - odczytuje formułę z pliku `.txt` i na jej podstawie tworzy listę klauzul, będących listami tokenów.

7.5 Moduł customerrors

Moduł ten implementuje dwie klasy błędów: `RFGError` - dziedziczącą po `RuntimeError` oraz `RFG-TimeoutError` - dziedziczącą po `RFGError`. Klasy te mają zdefiniowane konstruktory i funkcje odpowiedzialne za konwersję na string, tak, aby oprócz wiadomości informującej o powodzie rzucenia wyjątku, zwracały również czas wystąpienia błędu z dokładnością co do sekundy i parametry przetwarzanego wtedy problemu.

8 Objaśnienie nazw generowanych plików

W konstruktorze klasy `Generator` tworzony jest trzon nazwy większości generowanych plików dla aktualnie przetwarzanej formuły, w postaci:

```
"{test_type}_c{clauses_num}_a{atoms_num}_prec{percentage_of_safty_clauses}_lengths{zawartość
listy clause_lengths rozdzielona znakami '_' }{_poisson}*{_distribution_{problem5_distribution}}**"
(*ten człon występuje jeśli w formule wykorzystany jest rozkład Poissona; **ten człon występuje jeśli
problem jest typu 5).
```

Do tego trzonu dodawane są różne elementy, w zależności od rodzaju pliku:

- W przypadku plików z zapisem formuły - na koniec dodawany jest ciąg `"_F@_generated.txt"`, gdzie @ to identyfikator formuły ze zbioru $[1,2,3,R]$.
- W przypadku plików wejściowych Prover9 - na koniec dodawany jest ciąg `"{separator}_prover9.in"`, gdzie {separator} w przypadku pojedynczych formuł jest znakiem `"_"`, a w przypadku formuł składowych ma postać `"_file@"`, gdzie @ to identyfikator formuły ze zbioru $[1,2,3,R]$.
- W przypadku plików wejściowych SPASS Prover - na koniec dodawany jest ciąg `"{separator}_spass.in"`, gdzie {separator} w przypadku pojedynczych formuł jest znakiem `"_"`, a w przypadku formuł składowych ma postać `"_file@"`, gdzie @ to identyfikator formuły ze zbioru $[1,2,3,R]$.
- W przypadku plików wyjściowych Prover9 - na koniec dodawany jest ciąg `"_attempt@_prover9.out"`, gdzie @ to numer kolejnej próby testowej ze zbioru $[1,2,3]$.
- W przypadku plików wyjściowych SPASS Prover - na koniec dodawany jest ciąg `"_attempt@_spass.out"`, gdzie @ to numer kolejnej próby testowej ze zbioru $[1,2,3]$.

9 Opis rzucanych wyjątków

Wszystkie wyjątki opisane na poniższej liście są obsługiwane przez środowisko testowe. Jeśli wystąpi błąd, nie zawarty na tej liście, oznacza to krytyczną, nie przewidzianą sytuację - program przerwie działanie, a użytkownik powinien o zaistniałej sytuacji poinformować autora programu.

- **"Generator.generate: Test of type {test_type} should have atoms number coefficient equal to 0,5."** - błąd ten jest spowodowany użyciem współczynnika ilości atomów różnego od 0,5 w problemie innym niż 3. Do testowania różnych współczynników służy właśnie problem 3.
- **"Generator.generate: Test of type {test_type} should have more than one clause length given."** - błąd ten oznacza, że dla problemu, różnego od 4 lub 5, została podana tylko jedna długość klauzul. Do testowania formuł o jednej długości klauzul służy problem 4.

- **"Generator.generate: Tests of type {test_type} should have precentage of safety clauses equal to 50."** - błąd ten oznacza, że w problemie innym, niż 6 została podana ilość klauzul bezpieczeństwa różna od 50%. Do testowania innych stosunków ilości klauzul służy problem 6.
- **"Generator.generate: Tests of type {test_type} do not support poisson distribution."** - błąd ten oznacza, że zażądany został rozkład Poissona w problemie, który go nie umożliwia (czyli 1, 3, 4 lub 5).
- **"Generator.generate: Test of type problem3 should have atoms number coefficient grater than 1."** - błąd ten oznacza, że w problemie 3 podany został współczynnik atomów mniejszy lub równy 1, co uniemożliwia wygenerowanie klauzul.
- **"Generator.generate: Test of type problem4 should have single clause length given."** - błąd ten oznacza, że w problemie 4 podana została więcej, niż jedna długość klauzul.
- **"Generator.generate: Test of type problem5 should have exactly four clauses lengths given."** - błąd ten oznacza, że w problemie 5 podana została inna ilość długości klauzul, niż 4.
- **"Generator.generate: Unknown subtype of problem7."** - błąd ten oznacza, że podany został nieznan podtyp problemu 7 (do znanych należą 'a', 'b' oraz 'c') lub w ogóle nie został podany podtyp.
- **"Generator.generate: Unknown subtype of problem8."** - błąd ten oznacza, że podany został nieznan podtyp problemu 8 (do znanych należą 'a' oraz 'b') lub w ogóle nie został podany podtyp.
- **"Generator.generate: Unknown name of problem."** - błąd ten oznacza, że podany został nieprawidłowy typ problemu.
- **"Generator.generateProblem1: The number of required different clauses lengths is higher than the number of clauses."** - błąd ten oznacza, że w problemie 1 podano więcej długości klauzul, niż wynosi liczba klauzul.
- **"Generator.generateProblem3: No usable lengths"** - błąd ten oznacza, że w problemie 3, po zastosowaniu ograniczeń wynikających z podanego współczynnika ilości atomów, nie pozostały żadne długości klauzul.
- **"Generator.generateProblem5: The number of required different clauses lengths is higher than the number of clauses."** - błąd ten oznacza, że w problemie 5 podano więcej długości klauzul, niż wynosi liczba klauzul.
- **"Generator.generateProblem5: Invalid distribution. Choose from ["even","more_long", "more_short"]"** - błąd ten oznacza, że w problemie 5 podano nieprawidłową nazwę rozkładu.
- **"Generator.generateProblem6: Safety clauses precentage not in range [0;100]"** - błąd ten oznacza, że podany w problemie 6 procent klauzul bezpieczeństwa jest błędny - nie należy do przedziału [0;100].
- **"Generator.getRandomAtomList: The number of atoms is smaller than length of clause."** - błąd ten oznacza, że jedna z długości klauzul jest większa od ilości atomów w formule.
- **"Generator.getMostFrequentKey: The number of atoms is to big to use all of them."** - błąd ten oznacza, że w problemie innym niż typu 3 liczba atomów jest większa niż łączna długość wszystkich klauzul i niemożliwe jest użycie wszystkich atomów.

- **"ProverRunner.getMeasuresFromOutputFile: Results not found in output file."** - błąd ten oznacza, że w pliku wyjściowym provera nie zostały odnalezione wszystkie potrzebne elementy i nie było to wywołane przekroczeniem limitu czasu przez Prover9, ani przekroczeniem limitu pamięci przez SPASS Prover.
- **"TestEnv.makeTests: Provers got different results. Spass: {spass_stats['sat']}; Prover9: {prover9_stats['sat']}"** - błąd ten oznacza, że jeden z proverów stwierdził spełnialność, a drugi niespełnialność tej samej formuły.
- **"TestEnv.runProver: Different results for the same formula."** - błąd ten oznacza, że prover w różnych próbach stwierdził spełnialność i niespełnialność tej samej formuły.
- **"LogicToken.__init__: Token type not accepted."** - błąd ten oznacza, że nastąpiło żądanie utworzenia tokenu o niepoprawnym typie.
- **"LogicToken.__init__: ATOM token must have a value and negation flag set."** - błąd ten oznacza, że nastąpiło żądanie utworzenia tokenu typu ATOM, bez podania jego nazwy i wartości flagi negacji.
- **"readFormulaFile: Invalid token - {token}"** - błąd ten oznacza, że przy odczytywaniu pliku tekstowego z zapisaną formułą wystąpił niepoprawny token.
- **"ProverRunner.performMeasurements: prover9 timed out."** - błąd ten oznacza, że Prover9 przekroczył limit czasu. Występuje często i nie jest błędem w typowym tego słowa rozumieniu, służy bardziej odnotowaniu faktu, że proces provera musiał zostać przedwcześnie przerwany.

10 Modyfikowanie programu

Ze względu na brak typowo obiektowego podejścia do budowy programu, jego modyfikacja i rozwój wymagają dobrej znajomości kodu. Pomocne mogą się okazać komentarze, które opisują niemal każdą linię kodu.

10.1 Dodawanie nowych typów problemów

Dodanie nowego typu problemu wymaga:

- Dodania nowego sprawdzenia parametrów i wywołania w metodzie *generate()* klasy Generator.
- Utworzenia nowej metody typu *generateProblem* w klasie Generator, odpowiadającej charakterystyce problemu.

A ponadto uwzględnienia wszelkich specyficznych wymagań nowego problemu, co może wymagać modyfikacji konstruktora, metody *cleanup()* lub metody *translateAndSave()* klasy Generator, metody *makeCases()* klasy CaseMaker, metody *makeTests()* klasy TestEnv lub też każdej innej metody, na którą te wymagania będą miały wpływ.

10.2 Dodawanie nowych proverów

Dodanie nowego provera wymaga:

- Dodania nowej metody typu *translateTo*.
- Dodania nowej metody typu *joinFilesWithPattern*.

- Uwzględnienia powyższych metod we wszystkich miejscach, gdzie metody tego typu są wywoływane.
- Dodania nowego elementu w słowniku zwracanym przez metodę *translateAndSave()* klasy Generator.
- Dodania nowego wywołania komendy w metodzie *runProver()* klasy ProverRunner.
- Dodania nowego zestawu stanów w metodzie *getMeasuresFromOutputFile()* klasy ProverRunner.
- Dodania odpowiednich kolumn w pliku CSV generowanym przez metodę *makeTests()* klasy TestEnv.
- Dodania nowego wywołania metody *runProver()* w metodzie *makeTests()* klasy TestEnv.
- Przerobienia sprawdzeń odpowiedzialnych za porównywanie wyników spełnialności i zapisywanie ich do pliku CSV w klasie TestEnv.
- Dodania nazwy provera do listy czyszczonych procesów pod koniec metody *makeTests()* klasy TestEnv.

Ponadto wymaga uwzględnienia wszelkich zachowań charakterystycznych dla nowego provera, co może skutkować zmianami w niemal dowolnym miejscu w kodzie.

10.3 Zmienianie parametrów i limitów

Zmiana wymaganych lub domyślnych parametrów dla jakiegoś problemu wymaga sprawdzenia warunków analizujących parametry w klasie Generator oraz uwzględnienia zmian również w klasach pomocniczych modułu testenv.

Zmiana domyślnego limitu czasu wymaga zmiany wartości domyślnej parametru *time_limit* w metodzie *runProver()* klasy ProverRunner oraz zmiany wartości w obsłudze błędu RFGTimeoutError w metodzie *makeTests()* klasy TestEnv.

Zmiana domyślnego limitu pamięci dla Prover9 wymaga zmiany wartości zapisywanej do pliku w metodach *translateToProver9()* oraz *joinProver9FilesWithPattern()* klasy Generator i zmiany tej wartości w sprawdzeniu w metodzie *getMeasuresFromOutputFile()* klasy ProverRunner.

Zmiana jakichkolwiek innych elementów wymaga indywidualnego podejścia.