



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

*Analiza wydajności wybranych systemów dowodzenia twierdzeń dla
logiki pierwszego rzędu*

*Performance analysis of selected theorem provers
for the first order logic*

Autor:

Jakub Wiktor Semczyszyn

Kierunek studiów:

Informatyka

Opiekun pracy:

dr hab. inż. Radosław Klimek

Kraków, 2021

*Pragnę podziękować mojemu promotorowi,
dr hab. inż. Radosławowi Klimkowi,
za poświęcony czas, cierpliwość i szerokie wspar-
cie merytoryczne.*

Spis treści

Streszczenie.....	5
Wstęp.....	7
1. Dowodzenie formuł logicznych	11
1.1. Logika pierwszego rzędu.....	11
1.1.1. Podstawowe pojęcia.....	11
1.1.2. Rodzaje klauzul.....	12
1.2. SAT solvery	13
1.2.1. Problem spełnialności	13
1.2.2. Algorytm DLL	14
1.2.3. Metoda CDCL.....	14
1.2.4. Usprawnienia i kierunki rozwoju.....	15
1.3. Systemy dowodzenia twierdzeń	16
1.3.1. Nowe wyzwania w logice pierwszego rzędu	16
1.3.2. Charakterystyka Prover9.....	16
1.3.3. Charakterystyka SPASS Prover	17
2. Implementacja programu	18
2.1. Podział logiczny programu.....	18
2.2. Implementacja modułów	20
2.2.1. Generowanie formuł logicznych.....	20
2.2.2. Translacja do formatów zgodnych z systemami dowodzenia	28
2.2.3. Pomiar wydajności.....	38
2.2.4. Moduły pomocnicze.....	41
2.3. Skrócona instrukcja użytkownika.....	42
3. Testy wydajności	45
3.1. Środowisko testowe	45
3.1.1. Przygotowanie scenariusza testowego	45
3.1.2. Odczyt scenariusza.....	46

3.1.3.	Realizacja scenariusza i zapis wyników	48
3.1.4.	Sposób użytkowania środowiska testowego	49
3.2.	Wyniki pomiarów	50
3.3.	Wnioski.....	59
Podsumowanie	60
Bibliografia	63
Dodatek A – lista załączników	64

Streszczenie

Celem niniejszej pracy była analiza wydajności systemów dowodzenia twierdzeń dla logiki pierwszego rzędu Prover9 oraz SPASS Prover oraz stworzenie programu generującego losowe formuły logiczne. W ramach pracy powstało oprogramowanie pozwalające na wygenerowanie losowych formuł logicznych, reprezentujących jeden z ośmiu przewidzianych problemów, odzwierciedlających sytuacje mogące wystąpić przy modelowaniu wymagań rzeczywistych systemów za pomocą problemu spełnialności. Formuły są generowane w postaci Clause Normal Form (a w niektórych szczególnych przypadkach Conjunctive Normal Form) i składają się z klauzul bezpieczeństwa i żywotności, które dobrze oddają naturalne wymagania stawiane przed rzeczywistymi systemami. Oprogramowanie pozwala również na zapisywanie wygenerowanych formuł oraz tłumaczenie ich do formatów wykorzystywanych przez Prover9 i SPASS Prover, a następnie zapisywanie tych tłumaczeń. Ponadto zawiera funkcjonalność pozwalającą na uruchamianie systemów dowodzenia twierdzeń na utworzonych plikach z formułami lub też, bez korzystania z reszty programu, na istniejących już plikach i odczytywanie wyniku dowodzenia, wydajności czasowej oraz wydajności pamięciowej z plików wynikowych systemów.

W celu analizy wydajności systemów dowodzenia, przeprowadzonych zostało ponad 1200 testów na ponad 200 formułach. Aby było to możliwe w krótkim czasie, konieczne było dodanie do programu modułu, pozwalającego na automatyzację generowania formuł oraz uruchamiania systemów dowodzenia twierdzeń i pobierania ich wyników. Umożliwia on tworzenie scenariuszy testowych, które opisują parametry formuł, jakie mają kolejno zostać wygenerowane i przetestowane. Ponadto obsługuje możliwe błędy bez przerywania pracy i zapisuje uzyskane wyniki do pliku CSV.

Analiza uzyskanych wyników pozwoliła na określenie sytuacji, w których lepszą wydajność osiąga każdy z systemów dowodzenia twierdzeń, nie umożliwiła wyłonienia jednoznacznie lepszego z nich. Ze względu na lepsze działanie dla formuł o średnich parametrach, SPASS Prover okazał się bardziej przydatny przy dowodzeniu formuł reprezentujących rzeczywiste problemy, jednak przy dowodzeniu formuł, których parametry reprezentują ekstrema, czyli również formuły małych, które w modelowaniu rzeczywistych systemów występują dość często, większą wydajnością wykazał się Prover9.

W pracy zarysowano wykorzystane elementy logiki pierwszego rzędu oraz podstawy algorytmiczne systemów dowodzenia twierdzeń. Opisano sposób implementacji wszystkich modułów stworzonego programu. Przedstawiono wyniki pomiarów wydajności oraz wynikające z nich wnioski.

Abstract

The goal of this thesis was to analyse the efficiency of Prover9 and SPASS Prover - automated theorem provers for first order logic, as well as to create software for generating random logical formulas. As a result, a program was created, designed for generation of random logical formulas, which represent one of eight predefined problems, that reflect the conditions that may occur in the process of modeling real-life systems as satisfiability problems. The formulas are generated in Clause Normal Form (or Conjunctive Normal Form in particular cases) and are comprised of safety and liveness clauses, that adequately convey the typical requirements of real-life systems. The software also provides the functionality of saving generated formulas and translating them to Prover9 and SPASS Prover compatible formats, saving the translations as well. In addition, it includes a module, which allows the user to run the provers using the saved formulas files or any other formulas files and retrieve statistics about satisfiability of formulas, as well as time and memory efficiency of the provers, from output files.

In order to analyse the efficiency of automated provers, over 1200 tests were conducted on more than 200 formulas. To make it possible in a reasonably short time, it was necessary to create an additional module of the program, which goal is to automate the generation of formulas, the running of the provers and the results retrieval. It allows the user to create test scenarios, which describe the parameters of the formulas to be generated and tested. In addition, it handles any predictable errors and saves the tests results into a CSV file.

The analysis of the obtained results made it possible to specify the situations in which each prover is more efficient than the other, however it did not prove any of the ATPs to be explicitly better. As it works better with formulas defined by average parameters, SPASS Prover proved itself to be more useful for proving real-life problems, than Prover9, but for formulas defined by parameters that represent extrema, which includes short formulas that are quite often present in real-life problems models, it was Prover9 that achieved higher efficiency.

This paper outlines the elements of first order logic that were used in the program, as well as the basis of algorithms utilised in automated theorem provers. It describes the implementation of all the modules of created software. It also presents the results of provers efficiency measurements and the arising conclusions.

Wstęp

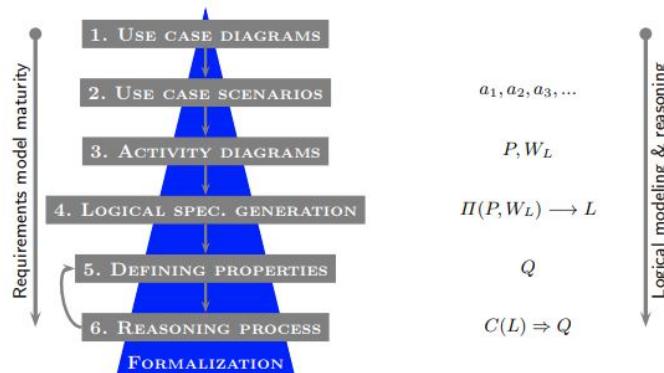
Niemal od początków swojego istnienia ludzie szukali sposobów na ułatwienie i automatyzację swoich zadań. W efekcie tych dążeń, na rzekach powstały młyny, pozwalające na zautomatyzowanie procesu mielenia zboża, ale też na napędzanie maszyn, których mechanizmy wykonywały pracę za człowieka. Jednym z pierwszych kroków rewolucji przemysłowej, która odmieniła oblicze cywilizacji było stworzenie mechanicznych maszyn tkackich. Wydarzeniem odpowiadającym za obecny stan branży motoryzacyjnej oraz sposób działania większości fabryk była automatyzacja linii produkcyjnej w zakładach Forda. Szczytowym osiągnięciem automatyzacji są jednak komputery. Umożliwiły one, przynajmniej częściową, automatyzację niemal wszystkich dziedzin życia, a w szczególności, od samego początku ich rozwój napędzany był przez równoległy rozwój matematyki. Dynamiczny rozwój nauk matematycznych spowodował, że twierdzenia tworzone przez uczonych osiągnęły taki poziom złożoności, iż, choć znaleźć można było wiele przesłanek wskazujących na ich prawdziwość, niemożliwym stało się dla człowieka udowodnienie ich w przeciągu jednego życia. Odpowiedzią na powstały problem było zaprojektowanie komputerów zdolnych do wykonywania obliczeń wiele tysięcy razy szybciej od człowieka [1]. Pozwoliło to na stworzenie algorytmów specjalizujących się w dowodzeniu formuł logicznych – solverów. Początkowo były to proste algorytmy, generujące wszystkie możliwe przypisania zmiennych, jednak z czasem dodawano do nich kolejne elementy, co pozwoliło na osiągnięcie imponującej wydajności. Obecnie na rynku dostępnych jest wiele solverów, organizowane są nawet konkursy, w których programiści konkurują, aby stworzyć algorytm, który najwydajniej poradzi sobie z udowodnieniem formuł z przygotowanego zbioru testowego.

Oprócz twierdzeń matematycznych, można dowodzić właściwie dowolne problemy, jak również zbiory wymagań występujących w życiu codziennym, konieczne jest jedynie przedstawienie ich w formie wyrażeń logicznych, dostosowanych do wybranego algorytmu. Poza solverami generalnego użytku, zaczęto tworzyć takie, które są zoptymalizowane pod rozwiązywanie konkretnych problemów, jak sudoku, dowodzenie poprawności programów, analiza gramatyczna tekstów, czy dowodzenie twierdzeń w logice pierwszego rzędu. Te ostatnie nazywa się systemami dowodzenia twierdzeń (ang. *automated theorem provers*), lub w skrócie proverami.

Cele i założenia pracy

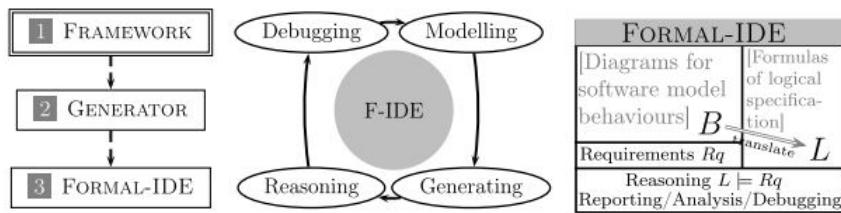
Niniejsza praca ma dwa główne cele. Pierwszym z nich jest opracowanie generatora formuł logicznych dla logiki pierwszego rzędu. Formuły mają być generowane losowo, na podstawie podanych przez użytkownika lub środowisko testowe parametrów. Drugim celem jest analiza wydajności czasowej oraz pamięciowej systemów dowodzenia twierdzeń: Prover9 oraz SPASS Prover, przy pomocy plików z formułami, utworzonych przez generator.

Istnieje rosnące zapotrzebowanie na coraz wydajniejsze rozwiązania programowe pozwalające na analizę i inżynierię wymagań (ang. *Requirements Engineering*). Tworzenie efektywnych, poprawnie działających programów, jak również całych systemów składających się z urządzeń i ich oprogramowania, wymaga spójnego, kompletnego przedstawienia zbioru wymagań. W tym celu wymagania przedstawione w języku naturalnym są dzielone na odseparowane od siebie elementy, co może odbywać się na przykład przy pomocy diagramów UML, a następnie te elementy tłumaczone są na zdania, czy klauzule logiczne i stopniowo udoskonalane [rysunek 1]. Proces modelowania jest jednak długi i wymaga wiedzy specjalistycznej zarówno z informatyki, jak i z dziedziny, której dotyczą modelowane wymagania. Aby móc testować programy odpowiedzialne za proces udoskonalania modelu, w tym programy dowodzące jego spełnialności, istnieje więc potrzeba możliwości szybkiego wygenerowania sztucznych wymagań w postaci formuły, co będzie zadaniem tworzonego generatora.



Rys. 1. Grafika przedstawia kolejne etapy procesu inżynierii wymagań. Źródło: [2]

Konieczność analizy wybranych systemów dowodzenia twierdzeń oraz samego testowania programów odpowiedzialnych za proces udoskonalania modeli wymagań powiązana jest z prowadzonymi obecnie na Wydziale Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej Akademii Górnictwo-Hutniczej badaniami nad stworzeniem środowiska pozwalającego na automatyzację procesu zilustrowanego na grafice [rysunek 1]. Analiza wydajności pozwoli na wybranie proverów, które zostaną wykorzystane w projekcie do realizacji kroku 6 - sprawdzania spełnialności utworzonego zestawu wymagań. Testowanie jest natomiast nieodzownym elementem rozwoju całego projektu, co pokazuje środkowy schemat na grafice [rysunek 2].



Rys. 2. Po lewej: trzy kolejne fazy powstawania projektu. **Pośrodku:** ilustracja etapów procesu powstawania projektu; punktem wejściowym jest modelowanie. **Po prawej:** zaplanowany wygląd okna środowiska (B - model zachowań systemu, L - tłumaczenie modelu na język logiki, Rq - utworzona formuła logiczna, która ma być poddana testom spełnialności). Źródło: [3]

Docelowo środowisko ma operować na logice temporalnej, jednak na rynku brakuje w pełni poprawne działających i rozwijanych systemów dowodzenia dla logiki temporalnej. Można jednak podstawowe założenia logiki temporalnej w prosty sposób zamodelować w logice pierwszego rzędu (dla której istnieją odpowiednie programy), co zostanie w tej pracy pokazane.

Projekt generatora został zrealizowany w języku Python3. Sprawdzenie wydajności systemów dowodzenia twierdzeń zostało przeprowadzone poprzez wykonanie ponad 1200 testów na ponad 200 losowo wygenerowanych formułach, reprezentujących typy problemów, odzwierciedlających wybrane aspekty formuł, jakie powstają przy modelowaniu wymagań. Konieczność przetestowania dużej ilości długich formuł oraz, dla zwiększenia dokładności wyników, testowania każdej formuły trzykrotnie, stworzyła potrzebę automatyzacji tego procesu - utworzenia prostego środowiska testowego, które, mając podany plik z kombinacjami parametrów wejściowych, uruchamia generator wielokrotnie i zapisuje wszystkie wyniki. Ostatnim etapem pracy jest analiza uzyskanych wyników oraz wyciągnięcie wniosków i ocena, do jakich zadań nadają się testowane systemy dowodzenia twierdzeń.

Struktura pracy

W kolejnym rozdziale nakreślona zostanie zasada działania i historia rozwoju solverów. Przedstawiony zostanie rynek proverów. Omówiona będzie charakterystyka logiki pierwszego rzędu oraz jej reprezentacji w systemach dowodzenia twierdzeń i sposób reprezentacji założenie logiki temporalnej przy pomocy logiki pierwszego rzędu.

Rozdział drugi podzielony jest na trzy części, z których każda odpowiada innemu etapowi działania zrealizowanego programu. Kolejno jest to: generowanie losowych formuł logicznych z uwzględnieniem wprowadzonych parametrów, translacja stworzonych formuł do formatów akceptowanych przez Prover9 oraz SPASS prover i uruchomienie proverów na gotowych formułach oraz pomiar wydajności pamięciowej i czasowej ich działania.

W rozdziale trzecim opisany zostanie sposób działania środowiska testowego, służącego do automatyzacji pomiarów oraz przedstawione i przeanalizowane będą wyniki przeprowadzonych pomiarów, a następnie wyciągnięte zostaną wnioski płynące z całokształtu wyników.

1. Dowodzenie formuł logicznych

W pierwszej sekcji tego rozdziału przedstawione zostaną podstawowe założenia logiki pierwszego rzędu oraz sposób jej reprezentacji w systemach dowodzenia twierdzeń. W drugiej sekcji omówiona zostanie zasada działania najprostszych solverów oraz historia ich późniejszego rozwoju. W trzeciej sekcji zaprezentowany zostanie rynek systemów dowodzenia twierdzeń, ze szczególnym zwróceniem uwagi na Prover9 oraz SPASS Prover, będące przedmiotem pomiarów przedstawionych w rozdziale "Testy wydajności" [rozdział 3].

1.1. Logika pierwszego rzędu

1.1.1. Podstawowe pojęcia

Alfabetem używanego na potrzeby tej pracy języka logiki pierwszego rzędu będzie (utworzony na podstawie definicji w [4, 5]) zbiór symboli:

- Przeliczalny zbiór symboli predykatów n -argumentowych, gdzie $n \geq 0$, przy czym predykaty 0-argumentowe są nazywane zdaniami elementarnymi.
- Przeliczalny zbiór symboli funkcji n -argumentowych, gdzie $n \geq 0$, przy czym funkcje 0-argumentowe są nazywane stałymi.
- Przeliczalny zbiór symboli zmiennych.
- Symbole logiki klasycznej: $true$, $false$, \neg , \vee , \wedge , \Rightarrow , \Leftrightarrow .
- Dwuargumentowy symbol predykatu równości $=$.
- Symbole kwantyfikatorów: \forall , \exists .
- Symbole interpunkcyjne: ")", "(", ",".

* Dodatkowo używany będzie symbol predykatu słabej większości \geq , nienależący do języka logiki pierwszego rzędu (w jednym z testowanych systemów dowodzenia jest on mimo wszystko zdefiniowany, w drugim trzeba go dodatkowo zdefiniować).

Atomami będziemy nazywali symbole predykatów - na potrzeby tworzonego programu: 0- i 1-argumentowe.

Literał to atom lub negacja atomu.

Klauzula to dysjunkcja dowolnej liczby literałów (lub implikacja pomiędzy dwoma takimi dysjunkcjami), do której zaaplikowano jeden lub więcej kwantyfikatorów (więcej o rodzajach klazul i ich budowie w następnym punkcie).

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

Taka definicja formuły sprawia, że przyjmuje ona formę Clause Normal Form, a przy pomocy prostych przekształceń, funkcjonalność zwana clausifier, która przeważnie jest częścią systemów dowodzenia twierdzeń [6], może zamienić poszczególne jej klauzule na w pełni zgodne z Conjunctive Normal Form. Clause Normal Form jest formą najlepiej nadającą się do reprezentacji wymagań, ponieważ składa się z koniunkcji odosobnionych zdań logicznych, co ma bezpośrednie przełożenie na listę odosobnionych wymagań dla poszczególnych elementów modelowanego systemu.

1.1.2. Rodzaje klauzul

Niżej przedstawione rodzaje klazul odnoszą się do własności bezpieczeństwa (*ang. safety*) i żywotności (*ang. liveness*), które należą do domeny logiki temporalnej (LT) [4]. Idea bezpieczeństwa polega na założeniu, że nigdy nie dojdzie do złego, niepożądanego zdarzenia. Idea żywotności reprezentuje stwierdzenie, że z pewnością dojdzie do jakiegoś pożdanego zdarzenia lub, że dojdzie do niego pod warunkiem, iż wcześniej będzie miało miejsce inne zdarzenie. Obie te idee doskonale odwzorowują rodzaje wymagań stawianych przed rzeczywistymi systemami. Choć są częścią logiki temporalnej, mogą być odwzorowane w logice pierwszego rzędu, poprzez zastąpienie operatorów logiki temporalnej odpowiednim złożeniem kwantyfikatorów oraz używanie 1- zamiast 0-argumentowych atomów. Argumentem atomów jest wtedy pojedyncza zmienna, będąca zmienną kwantyfikowaną, dla wygody zapisywana jako t lub t_i (gdzie i to liczba naturalna), nie musi jednak reprezentować czasu, może reprezentować dowolny inny parametr.

Klauzule bezpieczeństwa

W logice temporalnej służą do opisania, że do danego zdarzenia nigdy nie dojdzie. W uogólnieniu na logikę pierwszego rzędu określają, że pewne zdanie logiczne dla każdej wartości parametru będzie nieprawdziwe:

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

Gdzie t_0 to wartość minimalna kwantyfikowanego parametru (w LT jest to moment zerowy), a $B(t)$ to opisywane zdanie logiczne.

Mogą być używane między innymi do opisywania częściowej poprawności i poprawnego zachowania systemów oraz wzajemnego wykluczania i zapobiegania zakleszczeniom procesów [4].

Klauzule żywotności

W logice temporalnej służą do opisania, że dane zdarzenie na pewno się wydarzy. W logice pierwszego rzędu określają, że istnieje taka wartość parametru, dla której pewne zdanie logiczne jest prawdziwe:

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

Gdzie t_0 to wartość minimalna kwantyfikowanego parametru (w LT jest to moment zerowy), a $Q(t)$ to opisywane zdanie logiczne. Ta wersja klauzuli żywotności będzie dalej określana postacią ogólną.

Klauzule te mogą dodatkowo uwzględniać konieczność wystąpienia jakiegoś warunku i mają wtedy postać (nazywaną dalej warunkową):

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

Gdzie $P(t_1)$ to zdanie logiczne stanowiące warunek.

Mogą być używane między innymi do opisywania całkowitej poprawności systemów oraz dostępności i zapobiegania zagłodzeniu procesów [4].

1.2. SAT solvery

1.2.1. Problem spełnialności

Problem spełnialności to jeden z najbardziej podstawowych problemów informatyki. Polega na określaniu, czy dla danej formuły logicznej istnieje takie przypisanie zmiennych, dla którego formuła ta jest zawsze prawdziwa. Równoważnie można ten problem sformułować jako badanie, czy negacja formuły jest tautologią. Jest on jedną z instancji problemu spełniania ograniczeń (*ang. CSP - Constraint Satisfaction Problem*).

Zagadnienie to jest, zgodnie z twierdzeniem Cooka-Levina, problemem NP-zupełnym. Oznacza to, że każdy problem klasy NP może zostać do niego zredukowany w czasie wielomianowym. W związku z tym, również zadanie wypełnienia wymagań może być rozwiązywane jako problem spełnialności. Znalezienie takiego algorytmu, który pozwoliłby na jego rozwiązanie w $N^{O(1)}$ krokach, gdzie N to rozmiar formuły, byłoby jednoznaczne z udowodnieniem, że klasa problemów P jest równa klasie problemów NP [7].

Najprostszym sposobem rozwiązywania problemu spełnialności jest wygenerowanie wszystkich możliwych przypisań zmiennych, a następnie sprawdzanie ich prawdziwości, jednak sposób ten ma wykładniczą złożoność obliczeniową i nie nadaje się do działań na formułach z dużą ilością zmiennych, a i przy mniejszych formułach działa wolno.

1.2.2. Algorytm DLL

Nazwa algorytmu pochodzi od nazwisk jego twórców: Martina Davisa, George'a Logemanna oraz Donalda W. Loveland [8], choć bywa on również nazywany DPLL, ponieważ jest rozwinięciem stworzonego wcześniej algorytmu DP (Davis-Putnama). Stanowi wzbogacenie wspomnianego wcześniej najprostszego podejścia o chronologiczne nawroty i propagację ograniczeń boolowskich (*ang. BCP - Boolean Constraint Propagation*).

Algorytm generuje przypisania kolejnych zmiennych i sprawdza, czy nie stworzył w ten sposób sprzeczności, jeśli tak by się stało, to przypisanie, które doprowadziło do sprzeczności zostanie zanegowane i ta negacja zostanie dołączona do formuły jako kolejna klauzula (poprzez koniunkcję), a algorytm wykona nawrót do poprzedniego rozgałęzienia w drzewie decyzyjnym (nie koniecznie chodzi w tym miejscu o faktyczne drzewo, istnieją implementacje tego algorytmu, które nie budują drzewa, a więc określenie to oznacza tu zapamiętany w jakiś sposób stan algorytmu w momencie podejmowania decyzji) i podejmie przeciwną decyzję, jeśli nie została jeszcze sprawdzona lub nawrót o kolejny poziom, w innym przypadku. Dodatkowo algorytm w każdym kroku szuka formuł jednostkowych i je również zamienia na propagowane ograniczenia, a ponadto, jeżeli jakiś atom występuje w formule tylko z jedną polaryzacją (jedynie w formie zanegowanej lub jedynie w formie niezanegowanej), to jego przypisanie zostaje zapamiętane, a on sam zostaje wyłączony z dalszego poszukiwania przypisań. Te mechanizmy pozwalają na ograniczenie ilości przypisań oraz uproszczenie formuły, używanej do poszukiwania, już na samym początku działania algorytmu. Jednak, pomimo że takie podejście jest znaczco szybsze od zwykłego generowania wszystkich przypisań, to wciąż jego złożoność obliczeniowa jest wykładnicza.

1.2.3. Metoda CDCL

Przełomem w oprogramowaniu dowodzącym spełnialności był zaproponowany w 1996 algorytm GRASP, którego autorami byli J. P. Marques-Silva i K. A. Sakallah [9]. Stanowił początek nowego podejścia do poszukiwania spełnialności - CDCL (*ang. Conflict-Driven Clause Learning*). Algorytmy oparte o to podejście wykonują w pętli, dopóki dla wszystkich zmiennych nie zostanie wygenerowane przypisanie bez wystąpienia konfliktów lub też nie będzie możliwe wygenerowanie przypisania kolejnej zmiennej bez wywołania konfliktu, następujące kroki:

1. Wybierane jest kolejne przypisanie zmiennej.
2. Następuje propagacja ograniczeń boolowskich.
3. Jeśli na którymś etapie weszły konflikt, to diagnozowana jest przyczyna konfliktu.
 - a) Jeśli konflikt był wynikiem ostatniego wygenerowanego przypisania, to zostaje wygenerowane zamiast niego przeciwnie przypisanie.
 - b) Jeśli konflikt był wynikiem przypisania na wcześniejszym etapie (na niższym poziomie decyzyjnym), to wygenerowana na jego podstawie zostaje nowa klauzula z odpowiednimi ograniczeniami i zostaje wykonany niechronologiczny nawrót na właściwy poziom decyzyjny.

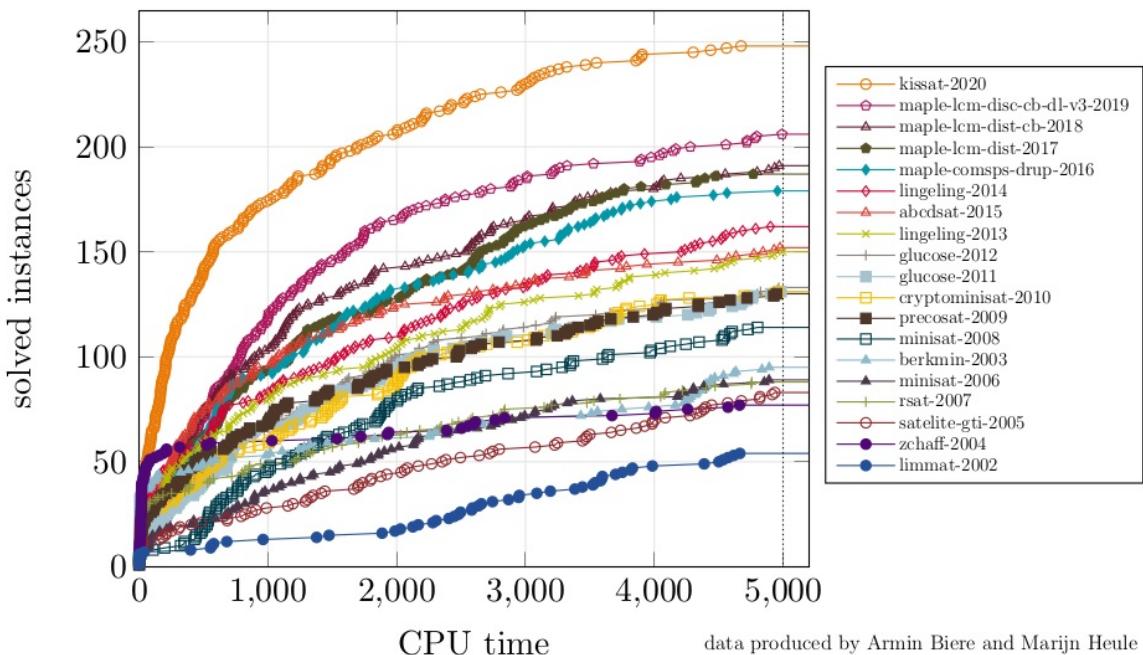
Programy takie w trakcie działania budują graf implikacji - strukturę pomocniczą, w której wierzchołki zawierają informacje o przypisaniu wartości do zmiennej i poziomie decyzyjnym tego przypisania, a krawędzie odpowiadają klauzulom, na podstawie których przeprowadzona została dedukcja.

Z uwagi na sposób, w jaki przeprowadzane jest wnioskowanie i generowanie nowych przypisań, formuła wejściowa dla algorytmów CDCL musi być podana w postaci CNF.

1.2.4. Usprawnienia i kierunki rozwoju

Obecnie algorytmy rozwijające ideę CDCL są nazywane SAT solvers. Osiągają olbrzymią wydajność, a ich twórcy konkurują ze sobą, testując swoje dzieła na specjalnie przygotowanych zbiorach.

SAT Competition Winners on the SC2020 Benchmark Suite



Rys. 1.1. Wyniki konkursu SAT Competition z roku 2020. Instancje, będące jednostką osi y , to formuły, z których największe mają miliony zmiennych i klauzul. Źródło: [10].

Takie wyniki są możliwe dzięki wzbogaceniu algorytmów o między innymi [11]:

- Używanie leniwych struktur danych (*ang. lazy data structures*) do reprezentacji formuł.
- Heurystyki o niskiej złożoności obliczeniowej, aktualizujące się przy nawrotach, służące do wyboru kolejnych nowych zmiennych i przypisań dla nich.
- Okresowe restartowanie przeszukiwania, przy zachowaniu wygenerowanych nowych klauzul.
- Warunkowe usuwanie dodanych klauzul.

- Wydajne implementacje propagacji ograniczeń.
- Usprawnione metody analizy konfliktów.

Rozwój programów rozwiązujących problem spełnialności, oprócz dalszego usprawniania SAT solverów, działających na prostym rachunku zdań, objął także powstanie nowych gałęzi, operujących na innych logikach. Te wyspecjalizowane odmiany algorytmów pozwalają na rozwiązywanie formuł z klauzulami o różnych wagach, mogą służyć do dowodzenia w programowaniu matematycznym, algebrze komputerowej lub też działać na innych rachunkach, jak chociażby, omawiany w tej pracy, rachunek zdań pierwszego rzędu.

1.3. Systemy dowodzenia twierdzeń

1.3.1. Nowe wyzwania w logice pierwszego rzędu

W generalnym ujęciu, spełnialność w logice pierwszego rzędu jest jedynie częściowo rozstrzygalna [12], co sprawia, że systemy dowodzenia twierdzeń mogą nie kończyć swojego działania (przydatna staje się w związku z tym funkcjonalność dostępna w większości z nich, umożliwiająca przerwanie poszukiwań po określonym czasie). Problem ten nie występuje jednak w przypadku formuł monadycznych, czyli takich, w których nie występują symbole funkcyjne, a symbole predykatowe są jedynie 1-argumentowe, a właśnie takie są na potrzeby tej pracy używane (wyjątkiem jest tu predykat słabej wielkości, który jest 2-argumentowy, ale jest używany w taki sposób, że nie wywołuje nierożstrzygalności problemu).

Prover9 dążą do udowodnienia formuły poprzez dowiedzenie sprzeczności jej negacji. W tym celu używają technik znanych już z SAT solverów, jednak, aby było to możliwe, dodatkowo wykorzystują algorytmy pozwalające im upraszczać lokalnie rachunek pierwszego rzędu do rachunku zdań, poprzez między innymi unifikację oraz wprowadzanie nowych zmiennych podstawieniowych.

Spośród dostępnych na rynku systemów dowodzenia twierdzeń, jako potencjalnie możliwe do wykorzystania w projekcie środowiska automatyzacji analizy wymagań wybrane zostały Prover9 oraz SPASS Prover. Stało się tak, ponieważ są to programy stale lub relatywnie niedawno wspierane, co wyróżnia je spośród większości dostępnych, których rozwój został szybko porzucony. Ponadto są proste w obsłudze i, względem innych proverów, wykazują się dużą niezawodnością.

1.3.2. Charakterystyka Prover9

Prover9 został stworzony przez Williama McCune jako następca cieszącego się dużą popularnością provera Otter. Jest oparty o rezolucję binarną i jednostkową. Jego cechą szczególną jest wysoce czytelny dla użytkownika język wprowadzanych formuł oraz zwracanego dowodu. Dzięki wykorzystaniu paramodulacji i superpozycji jest w stanie zagwarantować rozstrzygalność formuł zawierających 2-argumentowy predykat równości.

1.3.3. Charakterystyka SPASS Prover

SPASS został stworzony przez naukowców Max-Planck-Institut fur Informatik jako zarówno system dowodzenia twierdzeń, jak i platforma do rozwijania tego typu systemów. Jego kluczowym elementem jest metoda unifikacji zwana contextual rewriting, usprawniona o dodatkowe reguły eliminacji. Jego format wejściowy jest ustrukturyzowany i czytelny dla człowieka, chociaż aspekt ten jest zaburzony przez prefiksową notację operatorów logicznych. Ponadto prover ten obsługuje format TPTP, popularnej bazy problemów służących do testowania systemów dowodzenia.

2. Implementacja programu

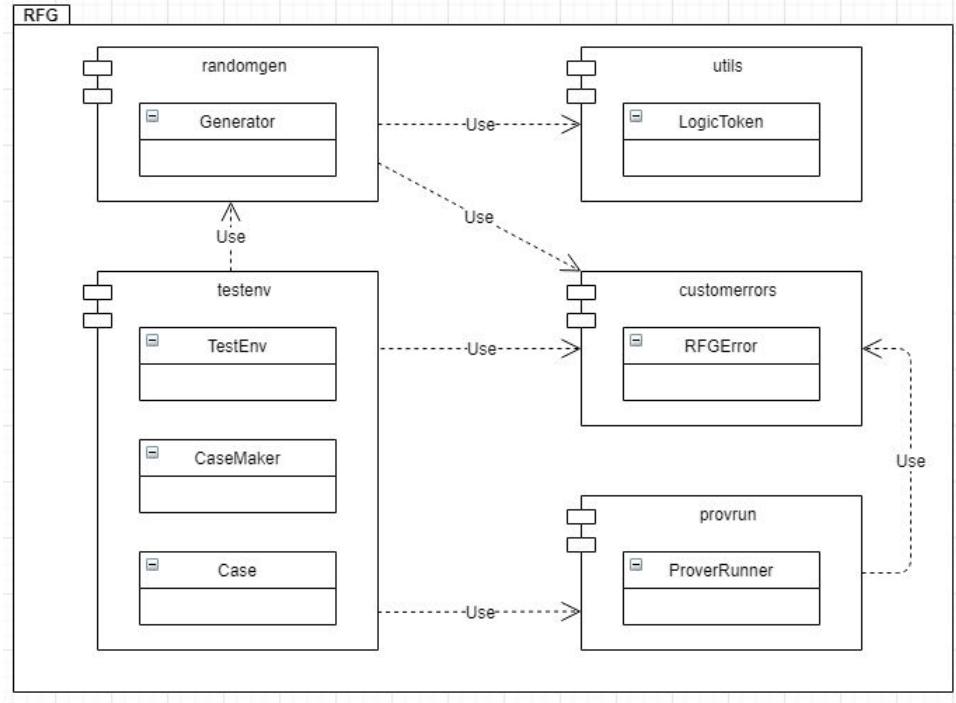
W rozdziale tym przedstawiony zostanie schemat działania programu. Wyjaśniony zostanie sposób implementacji generatora losowych formuł logicznych, zawartej w nim funkcjonalności, pozwalającej na tłumaczenie wygenerowanych formuł do formatów akceptowanych przez testowane w tej pracy systemy dowodzenia twierdzeń oraz skryptu służącego do pozyskiwania wyników z plików wyjściowych utworzonych przez prover. Poszczególne sekcje będą zawierać szczegółowy opis użytkowania programu - dostępne parametry, problemy możliwe do wygenerowania, towarzyszące im obostrzenia itp., dodatkowo na końcu zostanie jeszcze raz, w uproszczonej formie, omówiona instrukcja użytkowania krok po kroku. W rozdziale tym, jak również w całej pracy, opisana jest wersja 1.0.0 programu, a jeśli powstaną nowsze wersje, to zmiany w nich wprowadzone zostaną również naniesione w dokumencie z instrukcją użytkownika.

2.1. Podział logiczny programu

Aplikacja składa się z dwóch głównych modułów: generatora formuł logicznych (zwanego dalej generatorem) oraz menedżera pracy systemów dowodzenia twierdzeń (zwanego dalej prover runnerem lub po prostu runnerem). Oprócz nich w skład aplikacji wchodzą również dwa moduły pomocnicze: jeden z nich zawiera klasy i metody użytkowe, uniwersalne, nieprzynależne wyłącznie do jednego modułu, drugi definiuje nową klasę wyjątków, przystosowaną do potrzeb diagnostycznych programu. Ponadto dostępny jest moduł implementujący środowisko testowe umożliwiające automatyzację testów, jednak zostanie on w tej sekcji omówiony jedynie pobiędzie, szczegółowy jego opis jest tematem sekcji "Środowisko testowe" w rozdziale "Testy wydajności" [sekcja 3.1]. Zależności między poszczególnymi modułami pokazano na diagramie [rysunek 2.1].

Oprócz własnych modułów, program wykorzystuje również pakiety udostępniane z instancjami języka Python3 oraz elementy pakietu SciPy. Większość modułów wykorzystuje pakiet *os* do pozyskiwania informacji o ścieżkach dostępu do plików. Moduł *randomgen* wykorzystuje pakiet *random*, do generowania wszystkich losowych elementów formuły, oraz moduł *stats* z pakietu SciPy, aby utworzyć rozkład Poissona. Moduł *provrund* używa pakietu *subprocess*, aby uruchamiać systemy dowodzenia twierdzeń w osobnych procesach. Moduł *customerrors* wykorzystuje pakiet *datetime*, żeby zapisywać czas wystąpienia błędu. Moduł *testenv* używa pakietu *statistics* do wyliczania średniej i dominanty wyników proverów, pakietu *csv* do zapisywania tych wyników w plikach typu CSV oraz pakietu *subprocess*

do końca pracy proverów, które nie wyłączyły się na skutek błędu w trakcie poszukiwania rozwiązań formuły.



Rys. 2.1. Diagram przedstawiający zależności pomiędzy poszczególnymi modułami oraz zawarte w nich klasy.

2.2. Implementacja modułów

Działanie programu (szerzej opisane w sekcji "Skrócona instrukcja użytkownika" [sekcja 2.3]) można podzielić na 3 główne etapy: generowanie formuły logicznej na podstawie parametrów, tłumaczenie jej na formaty kompatybilne z systemami dowodzenia twierdzeń oraz uruchamianie tych systemów na przetłumaczonej formule, połączone z pomiarem wydajności. W tej sekcji każdemu z wymienionych etapów zostanie poświęcona osobna subsekcja, opisująca dokładnie jego sposób implementacji i wynikające z niego możliwości oraz ograniczenia.

2.2.1. Generowanie formuł logicznych

Za generowanie losowych formuł logicznych oraz translację wygenerowanych formuł odpowiada 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, poczawszy od konstruktora wywołującego metodę *generate()*, kończąc, na przykład, na metodzie *getRandomAtomList()*. W dalszej części tej sekcji, najpierw opisane zostaną typy problemów, jakich możliwość wygenerowania została przewidziana w programie, a potem kolejno opisane zostaną poszczególne metody klasy, aby zaprezentować generację formuły od ogólnego zarysu do szczegółowych elementów.

Możliwe do wygenerowania problemy

Program udostępnia możliwość wygenerowania formuły reprezentującej jeden z ośmiu predefiniowanych problemów, które ilustrują różne, wymagające dla systemów dowodzenia twierdzeń, aspekty wymagań modelowanych na podstawie rzeczywistych problemów. 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 (w scenariuszu testowym z rozdziału 3 [rozdział 3] jest to kolejno 50, 100, 200, 500, 1000 oraz 2000) oraz lista długości klauzul (w scenariuszu testowym jest to [2,3,4,6,8,10]). 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, który lepiej odzwierciedla rzeczywiste zbiory wymagań. Pozwala na wybranie, poprzez specyfikację parametru lambda, jakiej długości

klauzul będzie w formule najwięcej, jednocześnie charakteryzuje się względnie powolnym zmniejszaniem się prawdopodobieństwa w miarę oddalania się od wartości lambda (w przeciwnieństwie do na przykład rozkładu normalnego). W scenariuszu testowym została wykorzystana wartość lambda 3.5, ponieważ wymagania w rzeczywistych problemach najczęściej składają się z 3 lub 4 zmiennych. 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 (w scenariuszu testowym są to kolejno 2, 3, 4, 5 i 10). 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 (scenariusz testowy przewiduje kolejno długości 2, 3, 4 oraz 5). 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 (w scenariuszu testowym są to 1, 5, 10, 20) 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%. W scenariuszu testowym wszystkie rozkłady są testowane po kolei. 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 (w scenariuszu testowym są to: 10, 20, 35, 50, 65, 80, 90). 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. Ze względu na duże obciążenie proverów, scenariusz zawiera jedynie testy na formułach składowych długości 50, 100 i 200.

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

- Problem8a: $(\neg F1 \vee \neg F2) \wedge (F1 \vee F2)$ – koniunkcja implikacji oraz implikacji przeciwej (spełnialna gdy obie zmienne mają te samą wartość, czyli gdyoba pliki charakteryzuje taka sama spełnialność)
- Problem8b: $\neg(\neg F1 \wedge \neg F2)$ – implikacja przedstawiona za pomocą negacji koniunkcji
- Problem8c: $(\neg F1 \vee F2) \wedge \neg(\neg F2 \vee F1)$ – koniunkcja implikacji i implikacji odwrotnej (zawsze niespełnialna)

Ze względu na duże obciążenie proverów, scenariusz zawiera testy na formułach składowych długości 50, 100, 200, 500, 1000, nie testuje formuł o długości 2000.

Konstruktor klasy Generator

Konstruktor może przyjmować wszystkie parametry potrzebne w każdym z typów problemów, jednak jedynie sam typ problemu jest parametrem wymaganym, reszta ma ustalone 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_safty_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 klazul, 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 klazul, reprezentujące 4 formuły (formula, formula2, formula3 oraz formulaR), choć większość problemów będzie wykorzystywać jedynie pierwszą z nich. Wylicza liczbę atomów, mnożąc ilość klazul 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 kolej 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_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).
```

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.

Metoda generate

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.

Metoda generateProblem1

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.

Algorytm 2.1: Generowanie formuły reprezentującej problem typu 1.

```

1   begin
2     formula = []
3     ilosc_klauzul_kazdej_dlugosci = zaokr(ilosc_klauzul / ilosc_rznych_dlugosci_klauzul)
4     ilosc_klauzul_bezpiecz_jednej_dlugosci = zaokr(ilosc_klauzul_kazdej_dlugosci * procent
5       ↪ klauzul_bezpieczenia)
6     foreach dlugosc in lista_dlugosci_klauzul
7       for numer_klauzuli_biezacej_dlugosci in (0, ilosc_klauzul_kazdej_dlugosci)
8         if dlugosc_formuly < ilosc_klauzul
9           if numer_klauzuli_biezacej_dlugosci < ilosc_klauzul_bezpiecz_jednej_dlugosci
10             klauzula = generuj_klauzule_bezpieczenia(dlugosc)
11           else
12             klauzula = generuj_klauzule_zywotnosci(dlugosc)
13             formula = formula + [klauzula]
14
15     niedomiark_klauzul = ilosc_klauzul - dlugosc_formuly
16     niedomiark_klauzul_bezpiecz = zaokr(niedomiark_klauzul * procent_klauzul_bezpieczenia)
17     for licznik in (0, niedomiark_klauzul)
18       if licznik \leq niedomiark_klauzul_bezpiecz.
19         klauzula = generuj_klauzule_bezpieczenia(losuj(lista_dlugosci_klauzul))
20       else
21         klauzula = generuj_klauzule_zywotnosci(losuj(lista_dlugosci_klauzul))
22         formula = formula + [klauzula]
23
24   docelowa_formula = formula
25 end

```

Metoda jako parametry otrzymuje listę długość klauzul, ilość klauzul oraz możetrzymać procent klauzul bezpieczeństwa - domyślnie jest to 0,5 i numer docelowej formuły - domyślnie jest to 1. 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 pozostały. 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.

Metoda generateProblem2

Metoda ta implementuje wzorcowy algorytm dla metod używających rozkładu Poissona. Główną różnicą między nim, a algorymem dla problemu 1 [algorytm 2.1] 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 lambda 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 lambda, 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.

Metoda generateProblem3

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()*.

Metoda generateProblem4

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.

Metoda generateProblem5

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.

Metoda generateProblem6

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.

Metoda generateProblem7

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.

Metoda generateProblem8

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.

Metoda generateFormulaR

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.

Metoda generateSafetyClause

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

Metoda generateLivenessClause

Ta metoda wywołuje metodę *generateClause*, przekazując jej żądaną 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.

Metoda generateClause

W metodzie tej tworzone są dwie listy: literałów oraz relacji, które są następnie łączone naprzemianie 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.

Metoda getRandomAtomList

Wynikiem działania tej metody jest lista tokenów typu ATOM, które są dokładniej opisane w sekcji o module utils [subsekcja 2.2.4], a składają 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.

Metoda `getRandomRelation`

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.

Metoda `replaceORwithIMP`

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.

Korekta uzyskanej formuły

Po wygenerowaniu całej formuły, konieczne jest zastosowanie dwóch korekt, zaimplementowanych w metodzie `cleanup()`.

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 kluczów, 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. Operacja ta nie jest możliwa, 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.

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.

Zapisanie formuły do pliku

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 [rysunek 2.2]. 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  |

```

Rys. 2.2. 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.

2.2.2. Translacja do formatów zgodnych z systemami dowodzenia

Tłumaczenie odbywa się poprzez wywołanie metody generatora *translateAndSave()*. W przypadku większości typów problemów, jej działanie 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 oddzielnego 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.

Przed przystąpieniem do omawiania poszczególnych metod, warto nadmienić, że metody dla poszczególnych proverów są analogiczne w założeniach, mają takie same parametry wejściowe i wszystkie zwracają ścieżki do plików. Metody typu translate przyjmują parametry:

- raw - zmienna logiczna, określająca, czy formuła ma zostać przetłumaczona do "surowego" formatu, to znaczy nie zawierającego nagłówków, opisów i podobnych elementów, a jedynie samą formułę. Domyślnie jest ustawiona na False.

- file_name_appendix - jest to człon, który zostanie wstawiony między opis parametrów generatora, a nazwę provera w nazwie pliku wyjściowego. Domyślnie jest to “_”, ale w przypadku plików z formułami składowymi, zamiast tego zawiera numer formuły.
- source_formula - formuła, która ma zostać przetłumaczona. Domyślnie jest to główna formuła, bo w większości problemów występuje tylko ona.

Natomiast metody typu join:

- file_list - lista nazw plików, które mają zostać złączone.
- pattern - wzorzec, według którego ma nastąpić połączenie. Jest to lista elementów ze zbioru: [”FILE*”, ”NOT”, ”LP”, ”RP”, ”OR”, ”AND”], które oznaczają kolejno: zawartość pliku z formułą (* to numer formuły pomniejszony o 1), negację, nawias otwierający, nawias zamykający, dysjunkcję, koniunkcję. Przykładowo, formuła z problemu 7a: $\neg(F1 \vee F2 \vee F3) \vee R$; będzie reprezentowana przez wzorzec: [”NOT”, ”LP”, ”FILE0”, ”OR”, ”FILE1”, ”OR”, ”FILE2”, ”RP”, ”OR”, ”FILE3”].
- remove - zmienna logiczna, określająca, czy pliki składowe mają, po utworzeniu połączonego pliku, zostać usunięte. Domyślnie jest ustawiona na True.

Format Prover9

Format Prover9 jest bardzo zbliżony do prostego matematycznego zapisu formuł logicznych. W tym miejscu zostaną omówione tylko te jego elementy, które są wykorzystywane w programie, a szerszy opis wszystkich możliwości można znaleźć na stronie provera [13].

Na początku pliku można umieścić komendy zmieniające wartości zmiennych i flag provera. W plikach generowanych przez metodę *translateToProver9()* wykorzystywana jest komenda ”*assign (max_megs, 1024) .*”, która powoduje zwiększenie limitu pamięci z 200 MB na 1024 MB. Wartość taka została wybrana jako wystarczająco duża, żeby większość dowodów, możliwych do wyprodukowania w trakcie 300 sekund, jej nie przekroczyło oraz na tyle mała, aby wymagania sprzętowe programu były spełnialne przez niemal wszystkie relatywnie nowe komputery.

Formuła jest przedstawiana w najprostszej dostępnej w Prover9 formie - jako pojedyncza lista założeń, co jest reprezentowane poprzez otoczenie jej klamrą w postaci napisu ”*formulas (sos) .*” na początku i ”*end_of_list .*” na końcu. Słowo ”*sos*” jest tu skrótem od sformułowania *space of search*, które w języku angielskim oznacza przestrzeń poszukiwań i może być również zastąpione słowem ”*assumptions*”, czyli założenia.

Cała formuła jest reprezentowana w postaci Clause Normal Form, a więc jest listą klauzul, z których każda jest wpisana w osobnej linii oraz jest zakończona kropką. W trakcie działania provera poszczególne linie takiej listy są traktowane, jakby był między nimi znak koniunkcji.

W obrębie klauzul obowiązuje następująca reprezentacja symboli:

- ”all” - odpowiada kwantyfikatorowi uniwersalnemu.

- ”exists” - odpowiada kwantyfikatorowi egzystencjalnemu.
- ”-” - odpowiada negacji.
- ”|” - odpowiada dysjunkcji.
- ”&” - odpowiada koniunkcji.
- ”->” - odpowiada implikacji.
- ”>=” - odpowiada predykatowi słabej większości.
- Ciągi znaków zaczynające się od małej litery i zakończone wyrażeniem w nawiasach oznaczają funkcje.
- Ciągi znaków zaczynające się od wielkiej litery i zakończone wyrażeniem w nawiasach oznaczają predykaty.
- Ciągi znaków zaczynające się od małej litery z przedziału [u-z] oznaczają zmienne.
- Pozostałe ciągi znaków są traktowane jako stałe.

W powyższych definicjach sformułowanie ”ciągi znaków” odnosi się do ciągów nie będących słowami kluczowymi.

Klauzule bezpieczeństwa mają więc postać: ”`all u (-([F])) .`”, gdzie ”u” zastępuje użyte w definicjach [sekcja 1.1] ”t” z uwagi na wymogi dotyczące nazw zmiennych, a [F] oznacza dysjunkcję: ”`Var1(u) | Var2(u) | ...`”, gdzie liczby w nazwach predykatów są przykładowe, każdy z predykatów może być poprzedzony negacją, a wielokropki symbolizuje dowolną długość takiej dysjunkcji.

Klauzule żywotności mają postać: ”`all u exists u1 ((u >= u1) & (([F]) -> ([F])) .`”, gdzie oznaczenia są analogiczne, jak w przypadku klauzul bezpieczeństwa, a każda z dysjunkcji [F] jest inna.

Translację przykładowej formuły z poprzedniej sekcji [rysunek 2.2] na format Prover9 pokazano na rysunku [rysunek 2.3].

```

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.

```

Rys. 2.3. Przykładowa formuła przetłumaczona na format Prover9.

Format SPASS

SPASS Prover cechuje się formatem o większej ilości metadanych oraz o prefiksowej formie zapisu symboli logicznych. Pełny opis formatu znajduje się w dokumencie przygotowanym przez twórców [14].

Plik wejściowy należy rozpocząć od linii "begin_problem(nazwa_problemu)." i zakończyć linią "end_problem." Pomiędzy tymi liniami, treść problemu jest podzielona na listy, z których każda rozpoczyna się linią "list_of_[typ_listy].", a kończy linią "end_of_list.".

Pierwsza, obowiązkowa lista - list_of_descriptions - służy do opisu problemu. Tworzony w ramach pracy program zapisuje minimalną wersję tej listy, zawierającą jedynie obowiązkowe elementy, do których należą: name - nazwa problemu, author - autor problemu, status - spośród wartości {satisfiable, unsatisfiable, unknown}, description - opis problemu. Wszystkie z nich zapisuje się w osobnych liniach, w formie "[element] ([wartość]).", a wszystkie wartości, oprócz statusu, to napisy, które muszą być ujęte w klamrę "{}". W generowanych plikach nazwa problemu to opisująca parametry część nazwy pliku, element author zawiera imię i nazwisko autora pracy, status jest zawsze ustawiony jako unknown, ponieważ formuły są losowe i na etapie zapisywania ich do pliku program nie posiada wiedzy o ich spełnialności, natomiast opis problemu zawsze zawiera napis "{The problem was generated randomly}".

Kolejnym blokiem jest lista symboli - list_of_symbols, która może zawierać deklaracje symboli funkcyjnych i predykatowych. W liście generowanej przez program będą deklarowane jako symbole funkcyjne 0-argumentowe dwie stałe, służące później zdefiniowaniu relacji słabej większości, a jako symbole predykatowe, wszystkie zmienne występujące w formule (1-argumentowe) oraz symbol słabej większości (2-argumentowy).

Najbardziej obszerną z list jest list_of_formulae(axioms), na którą składają się wszystkie klauzule formuły, a więc, podobnie jak w przypadku Prover9, formuła jest reprezentowana w postaci Clause Normal Form. Każda klauzula zajmuje osobną linię i jest zamknięta w znaczniku "formula().", który, wbrew nazwie, istotnie reprezentuje to, co w pracy nazywane jest klauzulą - następuje tu niespójność oznaczeń między formatem provera, a wykorzystywany w pracy źródłami.

Oprócz zadeklarowanych w list_of_symbols, wykorzystywane w klauzulach są następujące symbole:

- "forall()" - odpowiada kwantyfikatorowi uniwersalnemu. Jest to symbol 2-argumentowy - pierwszym argumentem jest lista zmiennych kwantyfikowanych, a drugim zdanie logiczne.
- "exists()" - odpowiada kwantyfikatorowi egzystencjalnemu. Jest to symbol 2-argumentowy - pierwszym argumentem jest lista zmiennych kwantyfikowanych, a drugim zdanie logiczne.
- "not()" - odpowiada negacji, jest symbolem 1-argumentowym.
- "or()" - odpowiada dysjunkcji, jest symbolem o dowolnej liczbie argumentów, nie mniejszej od 1.
- "and()" - odpowiada koniunkcji, jest symbolem o dowolnej liczbie argumentów, nie mniejszej od 1.

- "implies()" - odpowiada implikacji, jest symbolem 2-argumentowym, z których pierwszy jest poprzednikiem, a drugi następnikiem.
- Ciągi "T" oraz "T1" są wykorzystywane jako oznaczenia zmiennych, aczkolwiek SPASS Prover nie przydziela znaczenia symbolom na podstawie sposobu ich zapisu, jak to miało miejsce w Prover9, toteż w ogólności zmienne mogą być dowolnymi ciągami liter, cyfr oraz znaku "_".

Wszystkie wymienione powyżej symbole prefiksowe można zagnieździć i łączyć.

Klauzule bezpieczeństwa będą w tym formacie przedstawiane w postaci: "formula(forall([T], not({or(literały(T))}))).", gdzie {or(literały(T))} to reprezentujący literal, pojedynczy predykat z argumentem T lub dysjunkcja, zawierająca porozdzielane przecinkami predykaty odpowiadające literałom, z których każdy ma T jako argument.

Klauzule żywotności będą postaci: "formula(forall([T], exists([T1], and(GTE(T, T1), implies({or(literały(T))}, {or(literały(T)}))))).", gdzie oba ciągi {or(literały(T))} zastępują analogiczne, jak w przypadku klauzul bezpieczeństwa, dysjunkcje, różniące się od siebie.

Tłumaczenie przykładowej formuły [rysunek 2.2] na format SPASS przedstawiono na rysunku [rysunek 2.4].

```

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.

```

Rys. 2.4. Przykładowa formuła przetłumaczona na format SPASS.

Translacja do formatu Prover9

Pierwszym, a zarazem najważniejszym etapem działania metody *translateToProver9()* jest utworzenie listy klauzul, w formie stringów, w formacie Prover9, na podstawie wybranej, wygenerowanej przez generator formuły. 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 przechowującą 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, poprzedzony spacja. 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, rozzielona znakami koniunkcji i odpowiednio obłożona nawiasami, co ma umożliwić późniejszełączenie kilku takich formuł dowolnymi operatorami. W takim wypadku formuła przyjmuje postać Conjunctive Normal Form zamiast Clause Normal Form.

Translacja do formatu SPASS

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 literał), 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 literał). 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 żywotności 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 pozostało niezmienioną.

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 kolejnych wszystkie listy, a sama lista klauzul jest porozdzielana znakami nowej linii. Zwrócona zostaje ścieżka do zapisanego pliku.

Łączenie plików Prover9

Metoda *joinProver9FilesWithPattern* pobiera listę plików z formułami oraz wzorzec ich łączenia. Pierwszym krokiem jej działania jest otworzenie 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 klauzule 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, zespółona lista wynikowa oraz stopka. Jeśli parametr *remove* nie był wyzerowany, to pliki ze składowymi formułami zostają usunięte. Zwrócona zostaje ścieżka do nowo powstałego pliku.

Łączenie plików SPASS

Podobnie, jak metoda związana z Prover9, metoda *joinSPASSFilesWithPattern* pobiera listę plików z formułami oraz wzorzec ich łączenia. Pierwszym krokiem jej działania jest otworzenie 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 zapamiętywanych 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 ustaliona, 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.

2.2.3. Pomiar wydajności

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

Konstruktor

Konstruktor klasy ProverRunner jest prosty - oprócz utworzenia instancji, jedynie zapisuje przekazane mu parametry: nazwę provera ("prover9" lub "spass") oraz ścieżkę do pliku wejściowego, w polach klasy.

Metoda performMeasurements

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 wyłapuje 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.

Metoda runProver

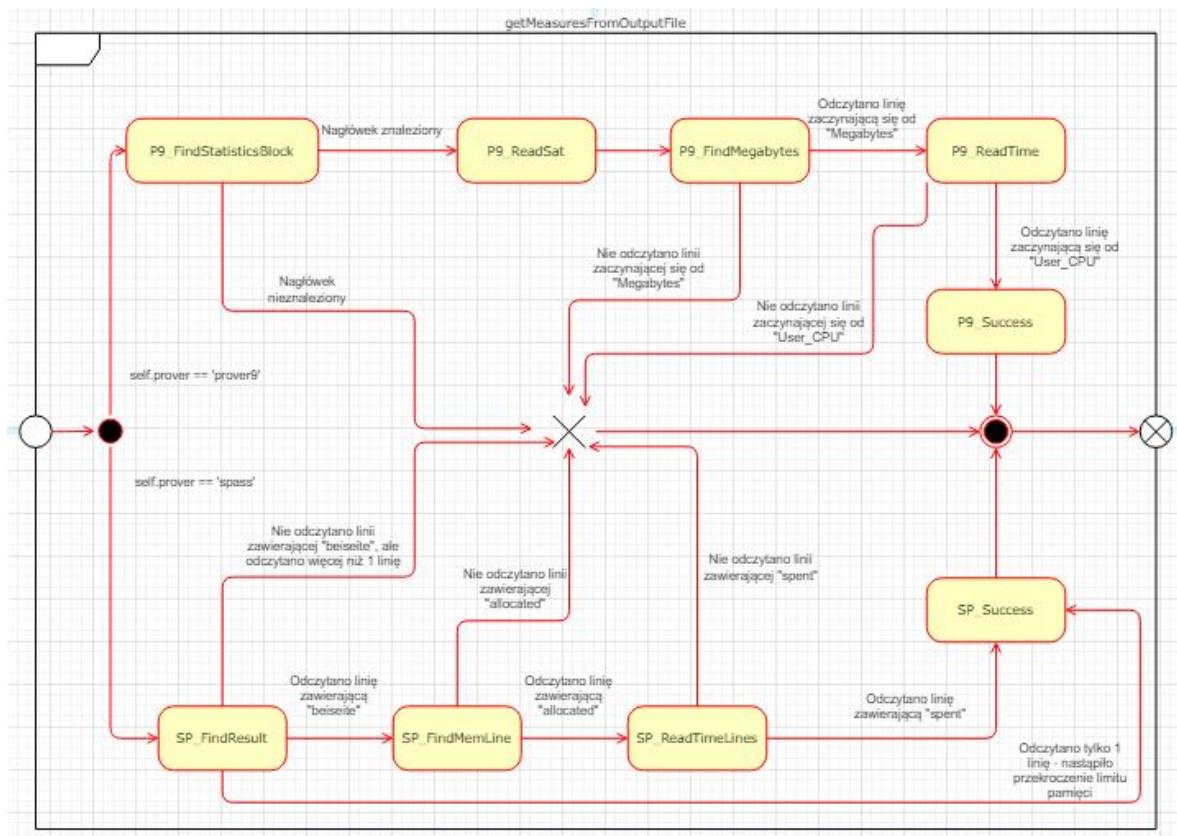
Zapisuje przekazany jej parametr ze ścieżką do pliku w polu klasy, a następnie buduje komendę terminala, służącą do uruchomienia systemu dowodzenia twierdzeń odpowiadającego wartości pola *prover* i uruchamia ją w nowym procesie przy pomocy funkcji *subprocess.call*. Komendy dla obu proverów zawierają limit czasowy, który domyślnie jest ustawiony na 300 sekund. 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 kilkadesiąt 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*.

Metoda getMeasuresFromFile

Jako że, zarówno Prover9, jak i SPASS Prover, umieszczają blok ze statystykami [rysunek 2.6 i rysunek 2.7] na końcu danych wynikowych, a jego elementy występują w ścisłej, z góry określonej kolejności, w tej metodzie do odczytu plików wyjściowych użyto maszyny stanów, której schemat obrazuje [rysunek 2.5].



Rys. 2.5. Diagram stanów metody getMeasuresFromFile().

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 nieznalezienie, 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 zakończane jest przeszukiwanie pliku.

```
=====
      STATISTICS
=====

Given=0. Generated=1. Kept=0. proofs=1.
Usable=0. Sos=0. Demods=0. Limbo=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
=====
```

Rys. 2.6. 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: " (co z języka niemieckiego można przetłumaczyć jako "żarty na bok"; nazwa programu: SPASS oznacza po niemiecku żarty lub zabawę), 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 ustawniona 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/RFG/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.
```

Rys. 2.7. 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.

2.2.4. Moduły pomocnicze

Moduł utils

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

Klasa LogicToken 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 literal, IMP - reprezentujący implikcję, 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.

Funkcja *zipToList()* 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.

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

Moduł customerrors

Moduł ten implementuje dwie klasy błędów: RFGError - dziedziczącą po RuntimeError oraz RFGTimeoutError - dziedziczącą po RFGError. Klasy te mają przeddefiniowane konstruktory i funkcje odpowiedzialne za konwersję na string, tak, aby oprócz wiadomości informującej o powodzie rzucenia wyjątku, zwracały również dokładną godzinę wystąpienia błędu i parametry przetwarzanego wtedy problemu.

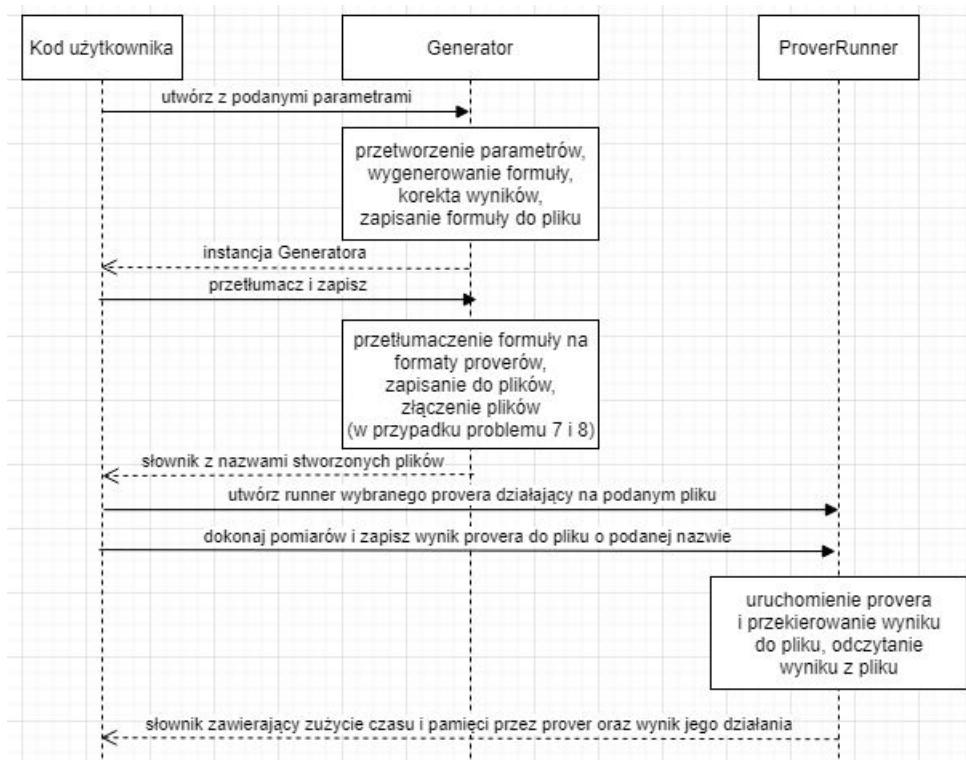
2.3. Skrócona instrukcja użytkownika

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 provera 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 tej pracy nie posiada wiedzy na temat jego występowania na innych dystrybucjach Linuxa) i prawdopodobnie nie zostanie rozwiążany 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.

Aby użyć programu, należy w pierwszej kolejności utworzyć instancję klasy Generator, przekazując do konstruktora pożądane parametry (ich opis znajduje się w pierwszym punkcie poprzedniej sekcji [subsekcja 2.2.1]). 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 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 schemat działania programu w formie diagramu sekwencji [rysunek 2.8] oraz przykładowe wywołania pozwalające na skorzystanie z jego funkcjonalności [rysunek 2.9].



Rys. 2.8. Diagram sekwencji przedstawiający działanie programu bez automatyzacji wywołań.

```

main.py
1  from provrun import ProverRunner
2  from randomgen import Generator
3
4  generator = Generator('problem3', clauses_num=2000, atoms_num_coeff=3)
5  files_dict = generator.translateAndSave()
6  runner = ProverRunner("prover9", input_file=files_dict["prover9_input"])
7  # remove extension ".in" and replace it with ".out"
8  prover_output_file = files_dict["prover9_input"][:-2] + "out"
9  results_dict = runner.performMeasurements(prover_output_file)
10 print(results_dict)
11

```

Rys. 2.9. Przykładowe użycie programu, bez wykorzystania środowiska testowego, dla problemu 3.

Pełna instrukcja użytkownika, zawierająca informacje zebrane z tej sekcji, sekcji o implementacji oraz rozdziału o środowisku testowym, a ponadto dodatkowe szczegóły, przedstawiona zostanie w osobnym pliku, aby łatwiej można było odnaleźć w niej konkretne fragmenty.

3. Testy wydajności

Jako że założenia pracy wymagały utworzenia około 210 instancji klasy Generator, a każda z wygenerowanych formuł miała być testowana trzykrotnie dla każdego systemu dowodzenia twierdzeń (co łącznie daje około 1260 uruchomień proverów), konieczna była automatyzacja procesu testowania. W tym celu powstał moduł testenv, który zostanie opisany w pierwszej sekcji tego rozdziału. W sekcji drugiej przedstawione zostaną wyniki przeprowadzonych pomiarów, w formie wykresów, które zostaną krótko omówione. Sekcja trzecia będzie zawierać wnioski dotyczące wydajności działania Prover9 oraz SPASS Prover, jakie wynikają z całokształtu uzyskanych wyników.

3.1. Środowisko testowe

Na moduł testenv, implementujący środowisko testowe, składają się trzy klasy: TestEnv - będąca głównym trzonem środowiska, w którym wykonywana jest większość działań, CaseMaker - klasa pomocnicza, której zadaniem jest tworzenie pojedynczych obiektów testowych na podstawie list parametrów oraz Case - reprezentująca obiekty testowe i tworząca instancje klasy Generator na podstawie otrzymanych parametrów. Obsługa programu poprzez środowisko testowe jest znacznie uproszczona, jednak wymaga przygotowania pliku ze scenariuszem.

3.1.1. Przygotowanie scenariusza testowego

Scenariusz należy zapisać w zwyczajnym 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 oczekiwany 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.

Na grafice [rysunek 3.1] został przedstawiony plik scenariusza wykorzystany do przeprowadzenia testów wydajności opisanych w dalszej części tej pracy.

3.1.2. Odczyt scenariusza

Konstruktor klasy TestEnv wymaga podania ścieżki do pliku ze scenariuszem. Inicjuje listę przypadków testowych, a następnie wywołuje metodę *readTestConfigFile()*, odczytującą scenariusz z pliku.

Wspomniana metoda czyta plik 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 rozpoczęjących 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 usuwainstancję klasy i wraca do poszukiwania linii z typem problemu.

Metoda *makeCases()* klasy CaseMaker iteruje po wszystkich listach będących polami tej klasy, tworzy wszystkie możliwe przypadki testowe, jako obiekty klasy Case i zwraca je w liście.

```

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
30 problem4
31
32 clauses 50 100 200 500 1000 2000
33 safety_prec 50
34 lengths 2 3 4 5
35 n_coeff 0.5
36
37 end of problem
38
39 problem5
40
41 clauses 50 100 200 500 1000 2000
42 safety_prec 50
43 lengths 1 5 10 20
44 n_coeff 0.5
45 distribution even more_short more_long
46
47 end of problem
48
49 problem6
50
51 clauses 50 100 200 500 1000 2000
52 safety_prec 10 20 35 50 65 80 90
53 lengths 2 3 4 6 8 10
54 n_coeff 0.5
55
56 end of problem
57

58 problem6
59
60 clauses 50 100 200 500 1000 2000
61 safety_prec 10 20 35 50 65 80 90
62 lengths 2 3 4 6 8 10
63 n_coeff 0.5
64 poisson True
65
66 end of problem
67
68 problem7a
69
70 clauses 50 100 200
71 safety_prec 50
72 lengths 2 3 4 6 8 10
73 n_coeff 0.5
74
75 end of problem
76
77 problem7b
78
79 clauses 50 100 200
80 safety_prec 50
81 lengths 2 3 4 6 8 10
82 n_coeff 0.5
83
84 end of problem
85
86 problem8a
87
88 clauses 50 100 200 500 1000
89 safety_prec 50
90 lengths 2 3 4 6 8 10
91 n_coeff 0.5
92
93 end of problem
94
95 problem8b
96
97 clauses 50 100 200 500 1000
98 safety_prec 50
99 lengths 2 3 4 6 8 10
100 n_coeff 0.5
101
102 end of problem
103
104 problem8c
105
106 clauses 50 100 200 500 1000
107 safety_prec 50
108 lengths 2 3 4 6 8 10
109 n_coeff 0.5
110
111 end of problem
112

113 problem7a
114
115 clauses 50 100 200
116 safety_prec 50
117 lengths 2 3 4 6 8 10
118 n_coeff 0.5
119 poisson True
120
121 end of problem
122
123 problem7b
124
125 clauses 50 100 200
126 safety_prec 50
127 lengths 2 3 4 6 8 10
128 n_coeff 0.5
129 poisson True
130
131 end of problem
132
133 problem8a
134
135 clauses 50 100 200 500 1000
136 safety_prec 50
137 lengths 2 3 4 6 8 10
138 n_coeff 0.5
139 poisson True
140
141 end of problem
142
143 problem8b
144
145 clauses 50 100 200 500 1000
146 safety_prec 50
147 lengths 2 3 4 6 8 10
148 n_coeff 0.5
149 poisson True
150
151 end of problem
152
153 problem8c
154
155 clauses 50 100 200 500 1000
156 safety_prec 50
157 lengths 2 3 4 6 8 10
158 n_coeff 0.5
159 poisson True
160
161 end of problem
162

```

Rys. 3.1. Plik scenariusza użytego w testach wydajności.

3.1.3. Realizacja scenariusza i zapis wyników

Po załadowaniu scenariusza, aby uruchomić testy, należy wywołać metodę *makeTests()* na obiekcie klasy TestEnv. Rozpoczyna ona 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).
- "Percentage 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, która wywołuje konstruktor klasy Generator, przekazując mu listę parametrów i zwraca tak utworzony obiekt. 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 RFGErro, 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 zostajewyłapany i program kończy działanie.

Drugi blok zawiera wywołanie metody *runProver()* dla Prover9. 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. 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" ustalone na {"sat": "Timeout", "memory": 0, "time": 300}. Ponadto błędy typu RFGErro 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 *pkill*. Powstałe w trakcie wywoływania bloku błędy typu RFGErro są wyłupywane i logowane do pliku, pozostałe powodują przerwanie pracy programu.

3.1.4. Sposób użytkowania środowiska testowego

Środowisko testowe realizuje wszystkie zadania kodu użytkownika przedstawione na diagramie [rysunek 2.8]. Użytkownik musi więc jedynie stworzyć plik ze scenariuszem testowym, co zostało opisane w poprzednich punktach tej sekcji, 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 zajść nawet kilka godzin - realizacja pełnego scenariusza przedstawionego na rysunku [rysunek 3.1] zajmuje około 12 godzin. Przykładowe uruchomienie środowiska testowego ilustruje grafika [rysunek 3.2].

```

from testenv import TestEnv
import os

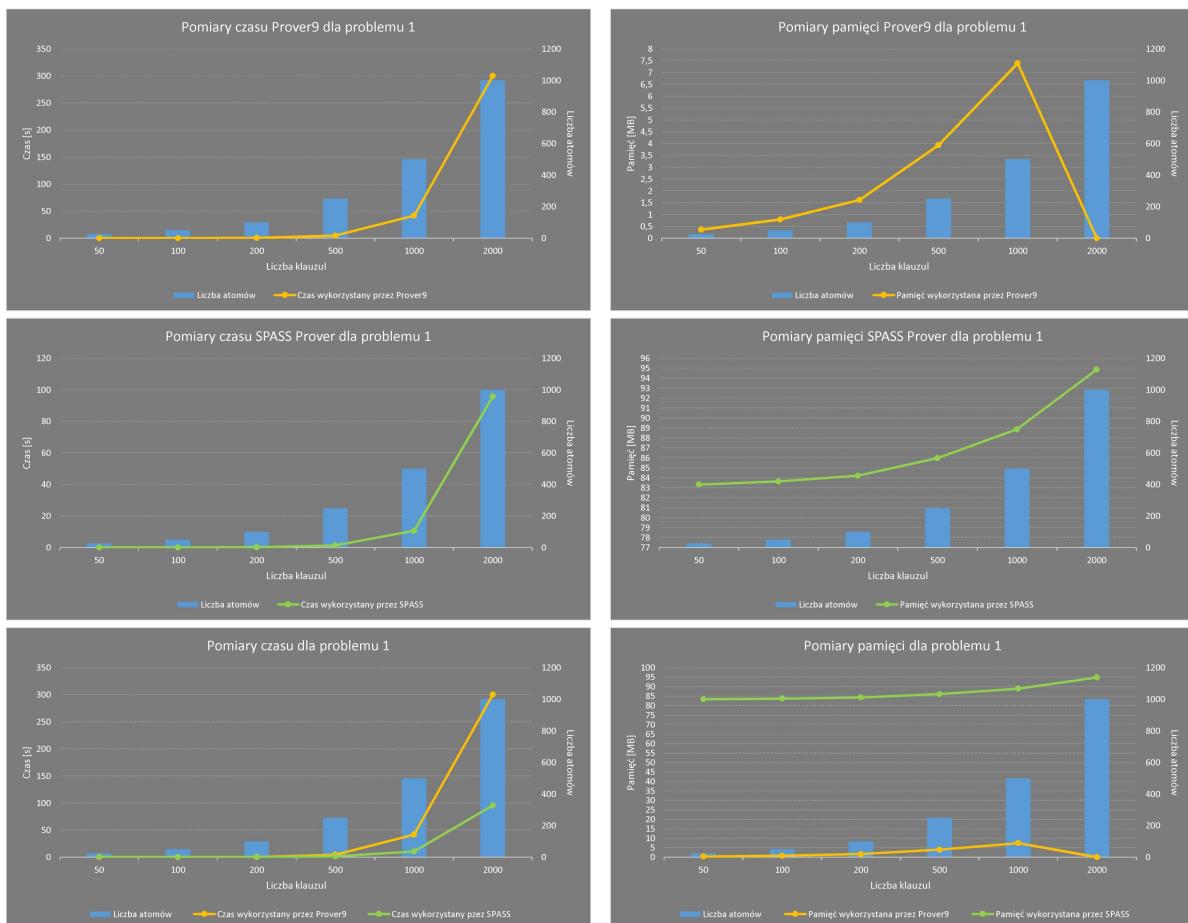
enviroment = TestEnv(os.path.dirname(__file__) + "/dane_testowe/scenario")
enviroment.makeTests()

```

Rys. 3.2. Skrypt uruchamiający środowisko testowe z utworzonym wcześniej scenariuszem.

3.2. Wyniki pomiarów

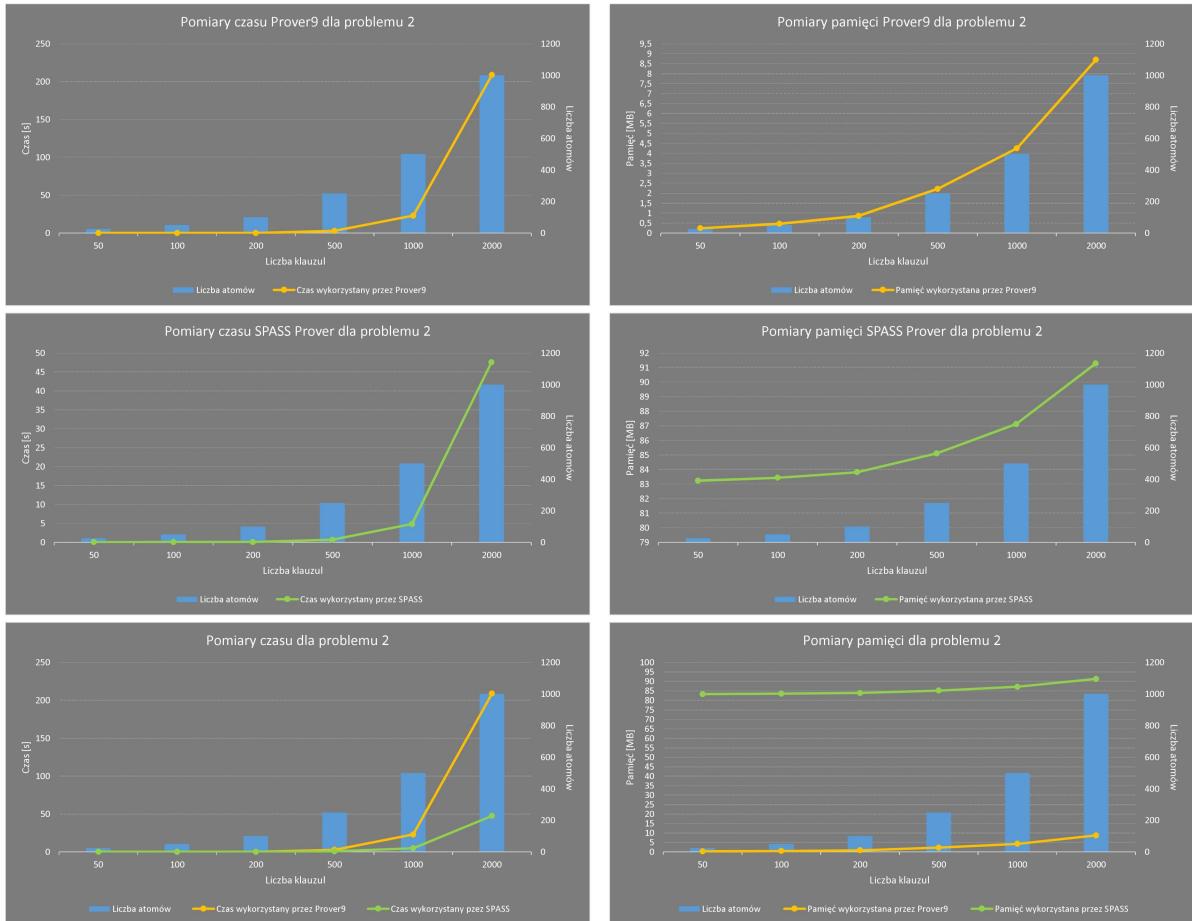
W tej sekcji przedstawione zostaną wykresy ilustrujące wyniki przeprowadzonych pomiarów, wraz z krótką ich analizą. Ogólne wnioski z pomiarów zawarte zostaną w sekcji następnej.



Rys. 3.3. Wyniki pomiarów na formułach reprezentujących problem 1.

Już przy testach na najprostszym z problemów - problemie 1 [rysunek 3.3], dało się zaobserwować znaczące różnice między działaniem obu systemów dowodzenia twierdzeń. SPASS Prover poradził sobie zdecydowanie lepiej czasowo od Prover9, nie przekraczając granicy 100 sekund, podczas gdy Prover9 przekroczył limit czasowy przy formule składającej się z 2000 klauzul. Przy dwóch najmniejszych długościach klauzul to jednak Prover9 okazał się szybszy. Ponadto wykorzystuje on o wiele mniej pamięci.

Można zauważyc, że choć wzrost używanej pamięci jest dla obu proverów dość podobny, SPASS Prover za każdym razem zajmuje około 80 MB więcej (zerowy wynik Prover9 dla najdłuższej klauzuli to efekt przekroczenia limitu czasu - pamięć nie jest wtedy mierzona).



Rys. 3.4. Wyniki pomiarów na formułach reprezentujących problem2.

Analizując wyniki dla problemu 2, wykorzystującego rozkład Poissona [rysunek 3.4], zauważyc można, że systemy radzą sobie lepiej pamięciowo i o wiele lepiej czasowo z takimi formułami, co będzie widoczne również w dalszych testach. Żaden z proverów nie przekroczył limitów. Ponownie zaobserwować można, że SPASS Prover o wiele szybciej radzi sobie z długimi formułami, niż Prover9, jednak przy formułach krótszych wypada nieznacznie wolniej. W wynikach pomiaru pamięci, podobnie jak po przednio, widać niemal identyczny trend obu proverów oraz dodatkowe 80 MB wykorzystywane przez SPASS Prover.

Wyniki dla problemu 3 [rysunek 3.5] pozwalają zauważyc, że, w ogólnym ujęciu, długość formuły ma większy wpływ na wydajność systemów, niż ilość atomów, jednak jeżeli stosunek ilości atomów do ilości klauzul jest duży - czyli duża jest również szansa, że żaden atom nie wystąpi w formule dwukrotnie, to obciążenie zarówno pamięciowe, jak i czasowe proverów jest znaczco większe, niż przy formułach o tej samej długości, ale mniejszej ilości atomów. Jeśli chodzi o wydajność czasową, to SPASS Prover okazuje się lepszy już przy formułach o długości 100 klauzul, a przy długich formułach jego przewaga

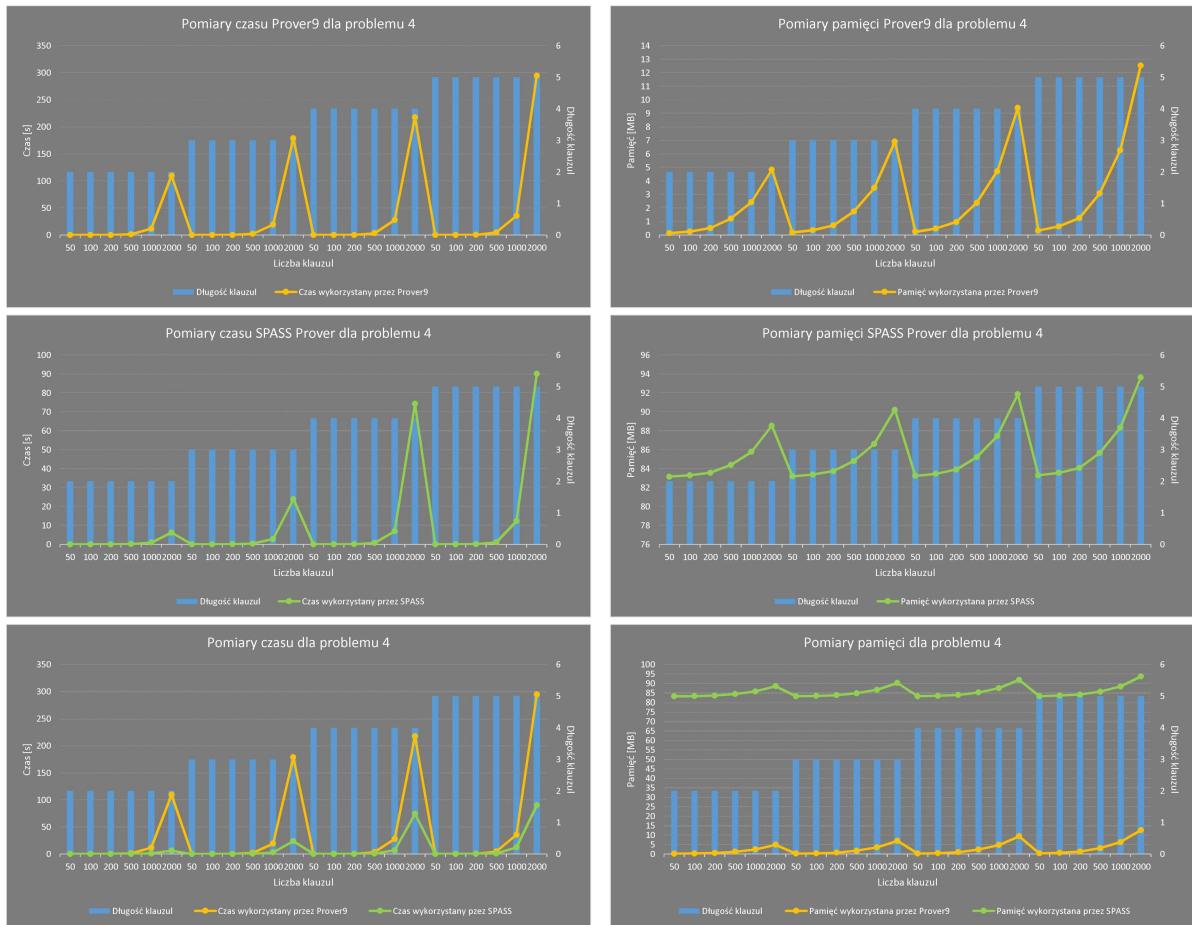


Rys. 3.5. Wyniki pomiarów na formułach reprezentujących problem3.

znacząco się pogłębia, choć dla największej formuły osiągnął wynik jedynie nieznacznie lepszy od Prover9. W kwestii wydajności pamięciowej kolejny raz lepszy okazał się Prover9 o około 80 MB w każdej próbie.

W wynikach dla problemu 4 [rysunek 3.6] potwierdzają się wszystkie wcześniejsze obserwacje: wstępna przewaga czasowa Prover9, późniejsza znacząca przewaga czasowa SPASS Prover oraz stała przewaga pamięciowa Prover9, wynosząca około 80 MB. Dodatkowo zaobserwować można, że długość klauzul znacząco wpływa zarówno na wykorzystywany czas, jak i pamięć, w przypadku obu proverów. Interesujące może również być spostrzeżenie, że Prover9 przy wszystkich klauzulach długości 5 osiąga podobne wyniki czasowe, co w problemie 1, przy klauzulach o różnych długościach, natomiast dla krótszych klauzul osiąga wyniki lepsze. Dla SPASS Prover granica wypada z kolei na klauzulach o długości 4 - formuły z krótszymi klauzulami przetwarzają szybciej, a z dłuższymi wolniej, niż formuły o różnych długościach klauzul.

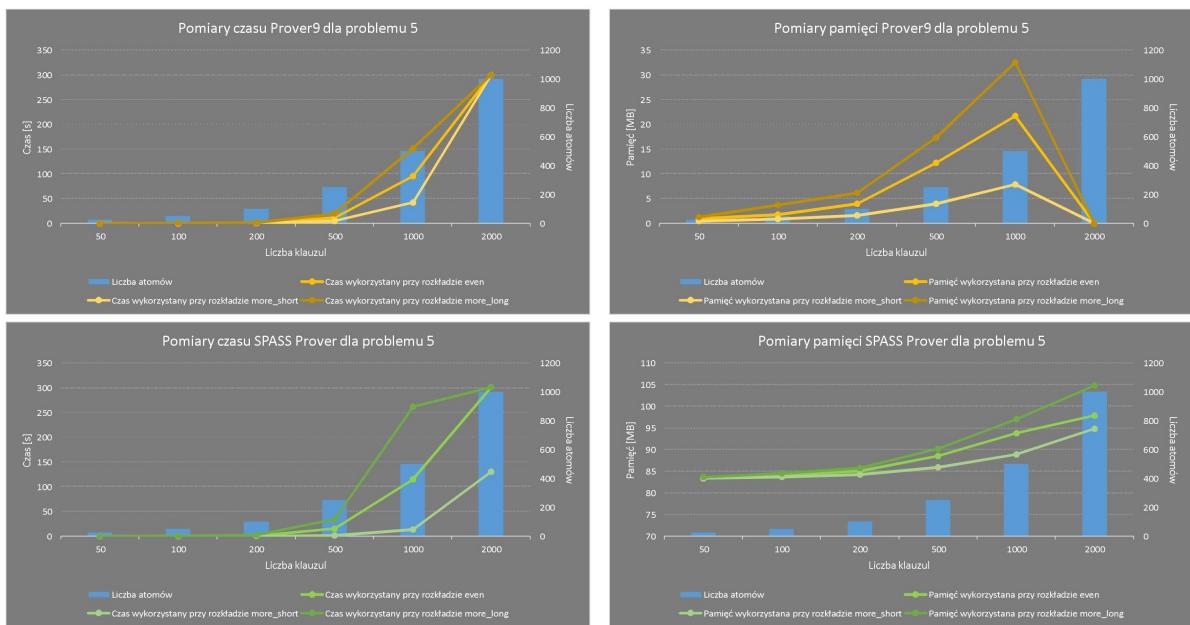
Przed przystąpieniem do omawiania wyników dla problemu 5 należy zwrócić uwagę na fakt, że problem ten wykorzystywał inny zestaw długości klauzul, niż większość pozostałych problemów: [1, 5, 10, 20], a więc występowały w nim klauzule bardzo krótkie, ale również bardzo długie i średnia długość



Rys. 3.6. Wyniki pomiarów na formułach reprezentujących problem4.

klauzul była znacznie wyższa, niż przy innych problemach. Analizując różnice pomiędzy poszczególnymi rozkładami długości w obrębie wyników z jednego provera [rysunek 3.7] można zauważać, że oba systemy radzą sobie tym lepiej, im więcej jest krótszych, a mniej dłuższych klauzul. Porównanie wyników pomiędzy proverami [rysunek 3.8] pozwala natomiast na obserwacje, które nie były możliwe przy krótszych klauzulach: SPASS Prover z formułami o wysokiej średniej długości klauzul osiąga zauważalnie gorsze wyniki czasowe, niż Prover9. Porównanie wyników pamięciowych prowadzi do analogicznych wniosków, jak w przypadku poprzednich problemów.

Wyniki pomiarów dla problemu 6 [rysunek 3.9] potwierdzają, że poza pojedynczymi odstępstwami, oba systemy dowodzenia twierdzeń lepiej radzą sobie z formułami, w których długości klauzul cechuje rozkład Poissona. Prover9 uzyskuje najgorsze wyniki czasowe dla formuł o równej ilości klauzul bezpieczeństwa i żywotności - im większa różnica między ilościami klauzul, tym wyniki czasowe są lepsze, a ponadto mniej czasu zajmuje dowodzenie formuł składających się głównie z klauzul żywotności, niż tych składających się głównie z klauzul bezpieczeństwa. Wykorzystanie pamięci przez Prover9 jest niemal identyczne dla formuł zawierających 35% lub więcej klauzul bezpieczeństwa, natomiast jest znacznie większe dla formuł zawierających 10% i 20% klauzul bezpieczeństwa. SPASS Prover osiąga tym lepsze wyniki, zarówno czasowe, jak i pamięciowe, im więcej większy procent formuły stanowią klauzule



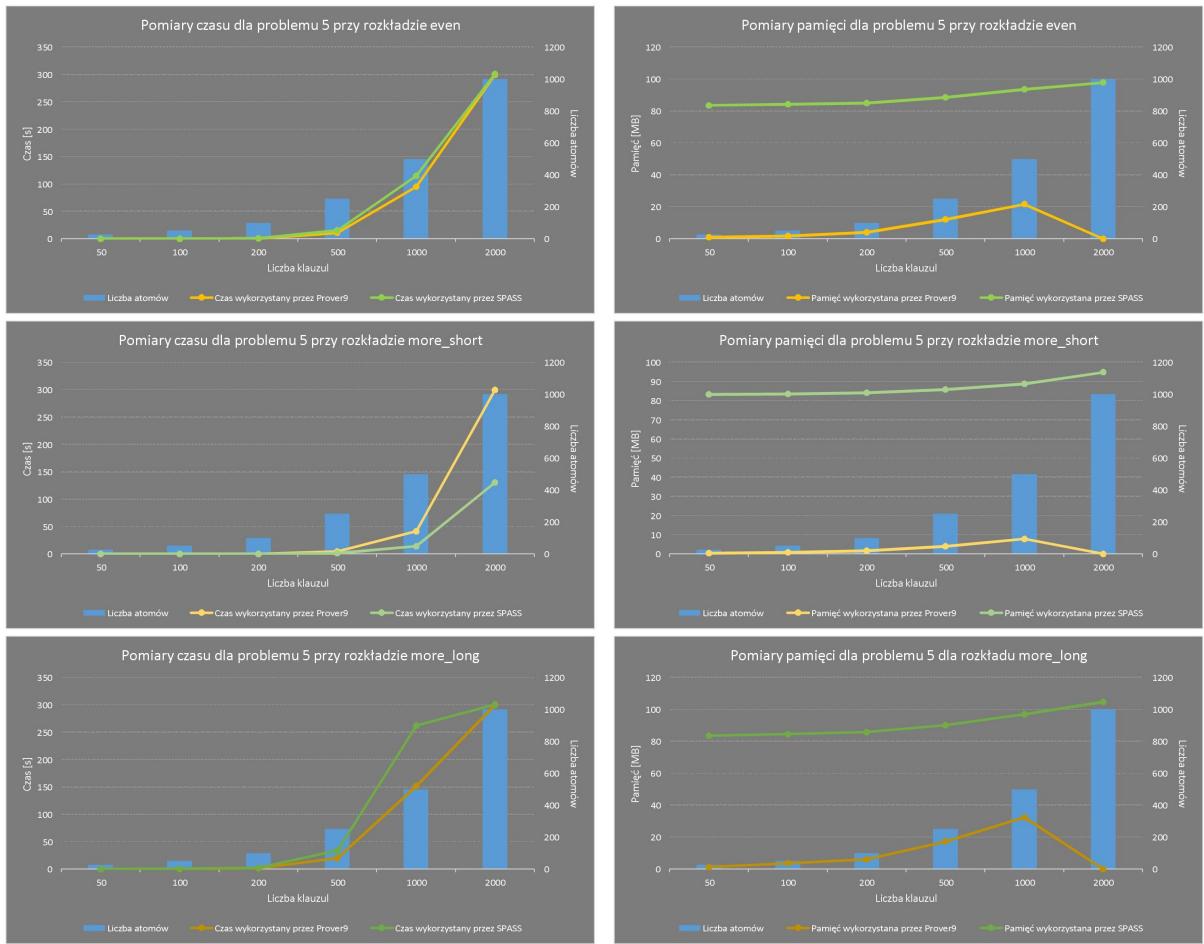
Rys. 3.7. Wyniki pomiarów na formułach reprezentujących problem 5.

bezpieczeństwa. Porównując wyniki obu proverów [rysunek 3.10], można zauważyc, że przy rozkładzie równomiernym SPASS Prover osiąga gorsze wyniki czasowe, niż Prover9 dla formuł o dużej przewadze ilości klauzuł żywotności nad ilością klauzuł bezpieczeństwa, natomiast znaczaco lepsze, gdy klauzul bezpieczeństwa jest więcej. Przy rozkładzie Poissona, niezależnie od stosunku ilości klauzuł różnych typów, SPASS Prover wykorzystuje o wiele mniej czasu, niż Prover9. Różnice w wykorzystaniu pamięci są analogiczne, jak we wcześniejszych problemach.

Dla wyników problemu 7 [rysunek 3.11] niemożliwe okazało się wyciągnięcie wiarygodnych informacji na temat działania Prover9, gdyż wszystkie próby zakończyły się przekroczeniem limitu czasu lub pamięci. SPASS Prover przeprowadził dowód dla wszystkich instancji problemu, a wyniki pozwalają stwierdzić, że formuły reprezentujące problem 7b powodują większe wykorzystanie czasu i pamięci, niż formuły reprezentujące problem 7a, a więc dysjunkcja formuł jest przetwarzana lepiej, niż koniunkcja, a ponadto potwierdzają obserwacje, że formuły o rozkładzie Poissona są mniej wymagające, niż te o rozkładzie równomiernym.

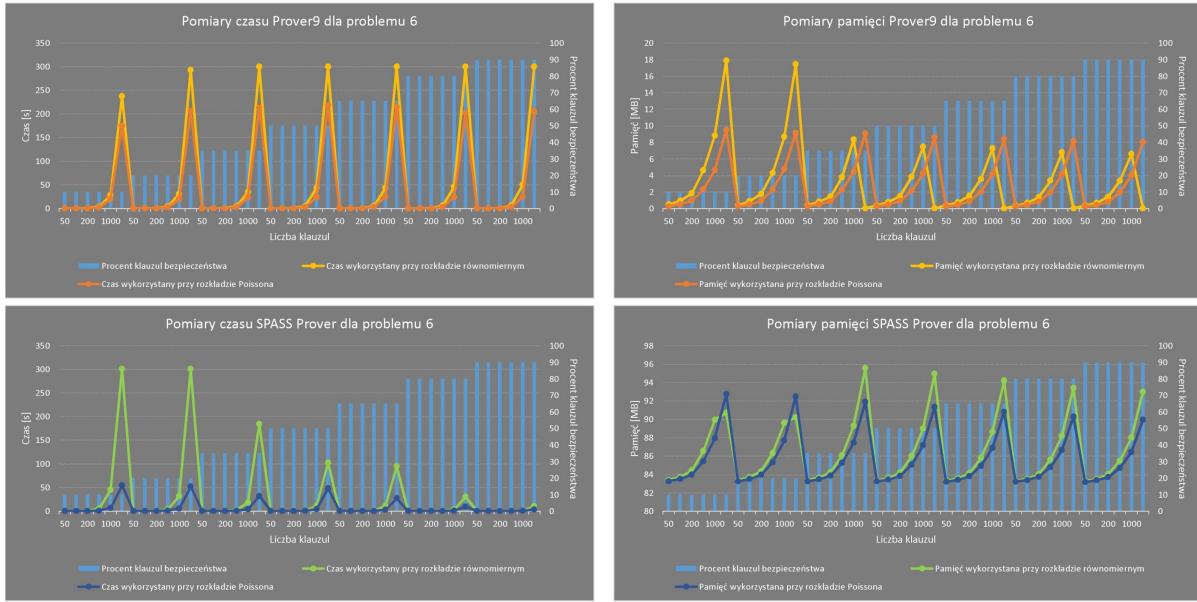
W przypadku problemu 8 [rysunek 3.12], wyniki dla Prover9 ponownie nie pozwalają na wyciągnięcie użytecznych wniosków z powodu błędów. Wyniki dla SPASS Prover również nie są kompletne, jednak można na ich podstawie odczytać, że najbardziej obciążającym czasowo podproblemem jest problem 8a, a najmniej problem 8c, natomiast najbardziej obciążającym pamięciowo jest również problem 8a, a najmniej problem 8b.

Zbiorcza analiza wyników zwracanych przez prover [rysunek 3.13] pozwala zaobserwować, że aż około 27% przygotowanych testów wywołało błąd przynajmniej jednego systemu. W 62% testów została dowiedziona niespełnialność formuły, co nie powinno dziwić, gdyż losowanych jest bardzo dużo klauzuł, czyli wiele różnych podzbiorów zbioru atomów, a klauzule bezpieczeństwa, jako negacje dysjunkcji,



Rys. 3.8. Porównanie wyników pomiarów na formułach reprezentujących problem 5.

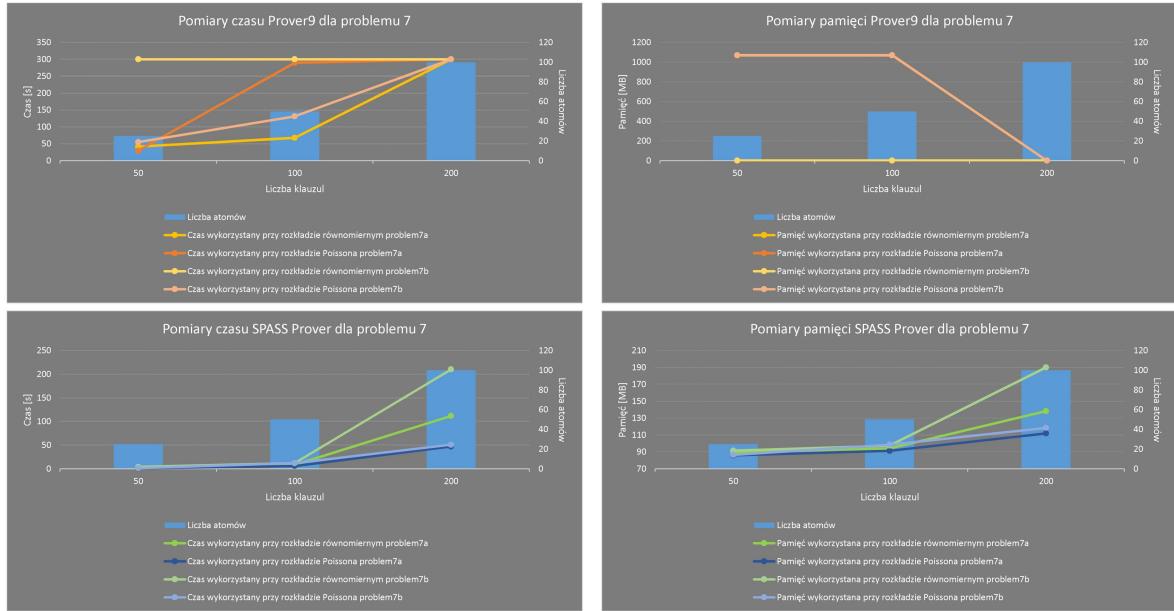
sprowadzają się w istocie do koniunkcji atomów, a więc już pojedyncza niezgodność pomiędzy klauzulami powoduje niespełnialność całej formuły. Pozostałe 11% testów wykazało spełnialność formuły. Były to wyłącznie formuły reprezentujące problem 3 i zarazem żadna formuła reprezentująca problem 3 nie okazała się niespełnialna. Wynika to z faktu, że wysoki stosunek ilości atomów do ilości klauzul w tych formułach powoduje, iż atomy rzadko się powtarzają, a więc mało jest możliwości sprzeczności.



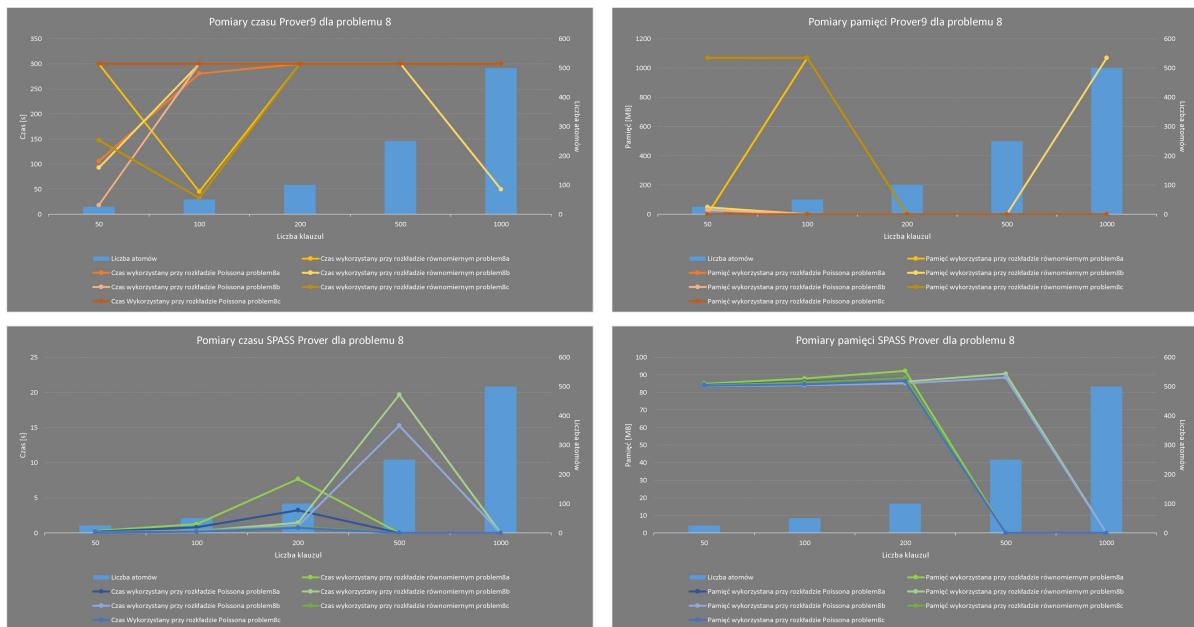
Rys. 3.9. Wyniki pomiarów na formułach reprezentujących problem6.



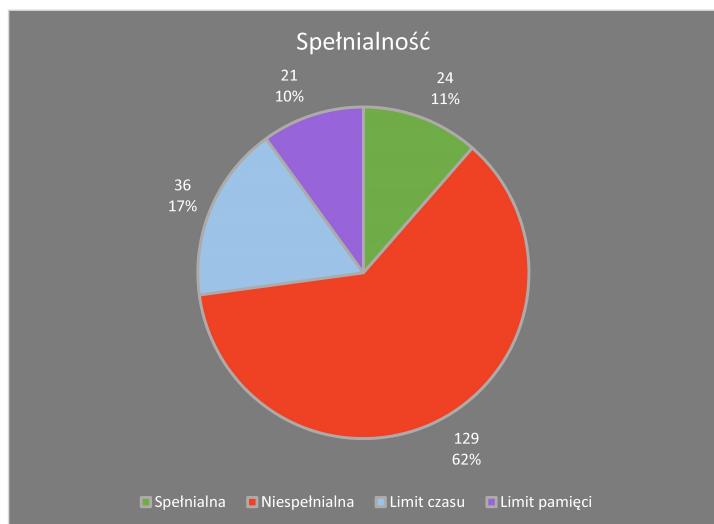
Rys. 3.10. Porównanie wyników pomiarów na formułach reprezentujących problem6.



Rys. 3.11. Wyniki pomiarów na formułach reprezentujących problem 7.



Rys. 3.12. Wyniki pomiarów na formułach reprezentujących problem 8.



Rys. 3.13. Udział poszczególnych wyników pomiarów w scenariuszu testowym.

3.3. Wnioski

Podsumowując wszystkie uzyskane wyniki można dojść do wniosku, że jedyną stałą różnicą w wydajności pamięciowej obu proverów jest dodatkowe około 80 MB wykorzystywane przez SPASS Prover. Ponieważ dla zdecydowanej większości współczesnych komputerów 80 MB pamięci nie stanowi niemal żadnej różnicy, uznać można, że oba systemy, poza najbardziej obciążającymi formułami, nie powodują przeciążenia sprzętu komputerowego i na podstawie samej analizy wydajności pamięciowej nie da się zdecydować, który z nich jest lepszy.

Analiza wykorzystania czasu pozwala stwierdzić, że do dowodzenia formuł składających się z co najwyżej 100 klauzul lepiej nadaje się Prover9, natomiast z formułami dłuższymi znaczco lepiej radzi sobie SPASS Prover. Duży wpływ na wydajność czasową ma średnia długość klauzul w formule i przy bardzo dużych jej wartościach Prover9 osiąga lepsze wyniki, niż Prover9. Formuły o rozkładzie Poissona są przetwarzane szybciej, niż te o rozkładzie równomiernym i prawdopodobnie ma to ponownie związek ze średnią długością klauzul w tych formułach. SPASS Prover osiąga najlepsze czasy dla formuł zawierających więcej klauzul bezpieczeństwa, niż żywotności. Prover9 natomiast lepiej radzi sobie z formułami, w których jest więcej klauzul żywotności, choć nieznacznie. Najgorsze wyniki osiąga przy formułach o równej ilości obu typów klauzul. Problemy polegające na łączeniu formuł okazały się zbyt wymagające dla Prover9.

Podsumowanie

Celem niniejszej pracy była analiza wydajności czasowej oraz pamięciowej systemów dowodzenia twierdzeń Prover9 i SPASS Prover poprzez implementacja generatora losowych, spараметryzowanych formuł logicznych. Zawarte w niej zostało krótkie przedstawienie wykorzystanych elementów logiki pierwszego rzędu, charakterystyka problemu spełnialności, objaśnienie algorytmów stanowiących podstawy działania systemów dowodzenia dla wszystkich rodzajów logiki oraz opis wymagań, jakie spełnić muszą systemy dowodzenia dla logiki pierwszego rzędu. W części praktycznej opisany został schemat działania stworzonego programu, sposób implementacji poszczególnych jego funkcjonalności oraz dworzona została skrócona wersja instrukcji użytkownika, umożliwiająca obsługę programu. Poszerzona instrukcja, zawierająca wiedzę pozwalającą na poszerzanie i modyfikację oprogramowania, została spisana w osobnym dokumencie.

W ramach pracy stworzony został program umożliwiający wygenerowanie i zapisanie do pliku oraz przetłumaczenie na format wejściowy Prover9 lub SPASS Prover losowej formuły logicznej, reprezentującej jeden z ośmiu przewidzianych problemów, umożliwiających przeprowadzenie testów obciążeniowych systemów dowodzenia twierdzeń. Ponadto stworzone oprogramowanie pozwala na uruchomienie Prover9 lub SPASS Prover, podając na wejście losowo wygenerowaną lub dowolną inną formułę, i automatyczne odczytanie wyniku działania tych systemów oraz statystyk dotyczących wykorzystanego czasu i pamięci.

Ponieważ pozyskanie wiarygodnych danych na temat działania wybranych systemów dowodzenia twierdzeń wymagało uruchomienia ponad 2000 testów, zaistniała potrzeba stworzenia dodatkowego modułu, którego zadaniem jest automatyzacja pracy głównego programu. Moduł ten pozwala na uruchomienie serii wielu testów i zebranie ich wyników poprzez utworzenie pliku tekstowego z zapisem scenariusza testowego i wywołanie zaledwie dwóch funkcji. Jego działanie oraz wyniki wykonanych testów zostały opisane w rozdziale trzecim pracy.

Analiza uzyskanych wyników pozwoliła na określenie sytuacji, w których lepiej sprawdzają się poszczególne prover, nie umożliwiła wyłonienia jednoznacznie lepszego systemu. Ze względu na lepsze działanie dla formuł o średnich parametrach, SPASS Prover znajdzie szersze zastosowanie przy dowodzeniu formuł reprezentujących rzeczywiste problemy, jednak do dowodzenia formuł, których parametry reprezentują ekstrema lepiej nadą się Prover9.

Stworzone w ramach pracy oprogramowanie w pełni realizuje jej cele i założenia, stanowi jednak zamknięte środowisko, którego rozszerzenie o kolejne typy problemów lub dostosowanie do testowania innych systemów dowodzenia twierdzeń wymaga dobrej znajomości kodu źródłowego oraz znaczającej jego modyfikacji. Dalszą drogą rozwoju programu mogłoby stanowić zastosowanie jeszcze bardziej obiektowego podejścia, co pozwoliłoby na zwiększenie modułowości kodu, a co za tym idzie, dodawanie nowych funkcjonalności poprzez jedynie tworzenie nowego kodu, bez modyfikacji już istniejącego. Pewnym ułatwieniem dla użytkownika mogłoby być stworzenie prostego interfejsu graficznego. Umożliwiłoby to łatwiejsze korzystanie również z samych proverów, które często nie posiadają własnego GUI. Ponieważ systemy dowodzenia twierdzeń nie poradziły sobie dobrze z większością formuł reprezentujących problemy 7 i 8, jeśli chciałoby się faktycznie mierzyć różnice pomiędzy poszczególnymi sposobami łączenia formuł, należało by poszukać sposobów, aby choć częściowo przekształcić wygenerowane złamania na Clause Normal Form - na przykład za pomocą transformacji Tseitina.

Bibliografia

- [1] Victor W. Marek Marijn J. H. Heule Oliver Kullmann. „Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer”. W: *Theory and Applications of Satisfiability Testing – SAT 2016* (2016), s. 228–245.
- [2] Radosław Klimek. „From Extraction of Logical Specifications to Deduction-Based Formal Verification of Requirements Models”. W: *Software Engineering and Formal Methods; 11th International Conference, SEFM 2013*. 2013, s. 61–75.
- [3] Radosław Klimek. „Pattern-based and composition-driven automatic generation of logical specifications for workflow-oriented software models”. W: *Journal of Logical and Algebraic Methods in Programming* 104 (2019), s. 201–226.
- [4] Radosław Klimek. *Wprowadzenie do logiki temporalnej*. Kraków: AGH. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, 1999. Rozd. 4 i 5.
- [5] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts in Computer Science. Springer New York, 2012. ISBN: 9781461223603.
- [6] Stuart Melville Geoff Sutcliffe. „The Practice of Clausification in Automatic Theorem Proving”. W: *South African Computer Journal* 18 (1996), s. 57–68.
- [7] Donald E. Knuth. *The Art of Computer Programming. Satisfiability*. T. 4. 2016. Rozd. 7.
- [8] Donald W. Loveland Martin Davis George Logemann. „A Machine Program for Theorem Proving”. W: *Communications of the ACM* 7.5 (1961), s. 394–397.
- [9] K. A. Sakallah J. P. Marques-Silva. „GRASP—A New Search Algorithm for Satisfiability”. W: *Proc. ICCAD 1996*.
- [10] Oficjalna strona Kissat solvera (stan na dzień 06.01.2021)
<http://fmv.jku.at/kissat/>.
- [11] Ines Lynce Joao Marques-Silva i Sharad Malik. „Conflict-Driven Clause Learning SAT Solvers”. W: *Handbook of Satisfiability* 185 (2009), s. 131–153.
- [12] John Harrison. *A Survey of Automated Theorem Proving*.
<https://www.cl.cam.ac.uk/~jrh13/slides/stpetersburg-28sep13/slides.pdf> (stan na dzień 06.01.2021). Intel Corporation, 2013.

- [13] *Oficjalna instrukcja Prover9 w wersji 2009-11A (stan na dzień 06.01.2021)*
<https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/>.
- [14] *Oficjalna instrukcja SPASS Prover w wersji 1.5 (stan na dzień 06.01.2021)*
<https://webspass.spass-prover.org/help/spass-input-syntax15.pdf>.

Dodatek A – lista załączników

- RFG-1.0.0.zip – archiwum z kodem programu. Jego zawartość jest opisana w instrukcji.
- RandomFormulaGenerator_Instrukcja.pdf – instrukcja użytkownika stworzonego programu.
- wyniki_testow_opracowanie.xlsx – arkusz kalkulacyjny zawierający wyniki przeprowadzonych testów i ich opracowanie.