AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Studio projektowe 1

Benchmark solverów prover9 oraz SPASS

29 marca 2021

Autorzy:

Mateusz Grzeliński

Przemysław Michałek

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Spis treści

1 Wprowadzenie

Celem projektu jest zbadanie wydajności automatyczych metod dowodzenia twierdzeń Prover9 oraz SPASS w kontekście formuł żywotnościowych i bezpieczeństwa, przedstawionych jako problem SAT (boolean satisfiability problem), a dokładniej FOF (First Order Formula). Na początku zostaje wygenerowna formuła SAT, która zostaje rozwiązana przez badane provery. Badany jest czas wykonania, rezultat (czy SAT jest spełnialny), użycie pamięci RAM. Generowana formuła SAT jest modyfikowana ze względu na między innymi długość formuły, ilość zmiennych.

Provery traktowane są jako czarne skrzynki (blackbox), ich parametry są modyfikowane z poziomu linii komend.

1.1 Żywotność i bezpieczeństwo formuł

Żywotność systemu to cecha, która zapewnia, że coś dobrego na pewno w końcu się wydarzy. Formuła żywotnościowa gwarantuje, że istnieje co najmniej jedno wydarzenie, dla którego formuła będzie spełniona.

Bezpieczeństwo systemu to cecha, która zapewnia, że nic złego nigdy się nie stanie. Formuła bezpieczeństwa gwarantuje, że dla każdego wydarzenia, formuła bezpieczeństwa nigdy nie zostanie pogwałcona.

Żywotność oraz bezpieczeństwo mogą zostać wyrażone na przykład w postaci formuły logicznej pierwszego rzędu, czyli problemu SAT. W formacie TPTP formuły żywotnościowe i bezpieczeństwa można przedstawić jako:

- kwantyfikatory (FOF) - w tptp kwantyfikator uniwersalny to `!`, a kwantyfikator egzystencjalny `?`
- kaluzule CNF (Conjunctive Normal Form) - powstaje po przekonwertowaniu formuły z kwantyfikatorem w klazulę

```
fof(simple_exists, axiom,
 ? [W,Z] :  p(W, Z) | p(a, b)
```

```
  ).

% converted with TPTP2X, otter algorithm
cnf(simple_exists_1,axiom,
    ( p(sk1,sk2) | p(a,b) )).
```

```
fof(simple_for_all, axiom,
 ! [W,Z] :  p(W, Z) | p(a, b)
  ).

% converted with TPTP2X, otter algorithm
cnf(simple_for_all_1,axiom,
    ( p(A,B) | p(a,b) )).
```

```
% dla każdego X, Y operacja lesseq, to to samo co less lub równość
fof(this_is_obvious, axiom,
  ! [X,Y] : ( $lesseq(X,Y) <=> ( $less(X,Y) | X = Y ) )
  ).

% converted with TPTP2X, otter algorithm
cnf(this_is_obvious_1,axiom,
    ( ~ $lesseq(A,B) | $less(A,B) | A = B )).

cnf(this_is_obvious_2,axiom,
    ( ~ $less(A,B) | $lesseq(A,B) )).

cnf(this_is_obvious_3,axiom,
    ( A != B | $lesseq(A,B) )).
```

```
fof(combined, axiom,
 ? [W,Z] : ( ! [X, Y] : p(W, Z, X)  | d(Y) )
  ).

% converted with TPTP2X, otter algorithm
cnf(combined_1,axiom,
    ( p(sk1,sk2,A) | d(B) )).
```

1.1.1 Przykład: zdawanie egzaminu

Problem: zalicz egzamin, aby zdać kurs

Warunek bezpieczeństwa: jeżeli podejmujesz się egzaminu, zalicz go

Warunek żywotnościowy: kiedyś musisz podejść do egzaminu

1.1.2 Przykład: światła na skrzyżowaniu

Problem: samochody chcą przejechać przez skrzyżowanie

Warunek bezpieczeństwa: tylko jedno światło powinno być zielone

Warunek żywotnościowy: każde światło powinno kiedyć zmienić się na zielone

2 Benchmark

Problem benchmarku w strategii blackbox sprowadza sie do wykonania podprogramu z odpowiednimi opcjami z poziomu linii komend. Wejście oraz testy benchmarka ustawiane są poprzez plik konfiguracyjny, sekcja **??**. Wejściem testu jest zbiór formuł SAT zapisanych na dysku w formacie TPTP w postaci pliku tesktowego. Test polega na podaniu pliku TPTP do provera. W razie potrzeby nastąpi automatyczna konwersja do odpowiedniej składni za pomocą dostępnych translatorów. Dla każdego provera dostępne są statystyki:

- czas wykonania
- użycie pamięci RAM
- spełnialność formuły

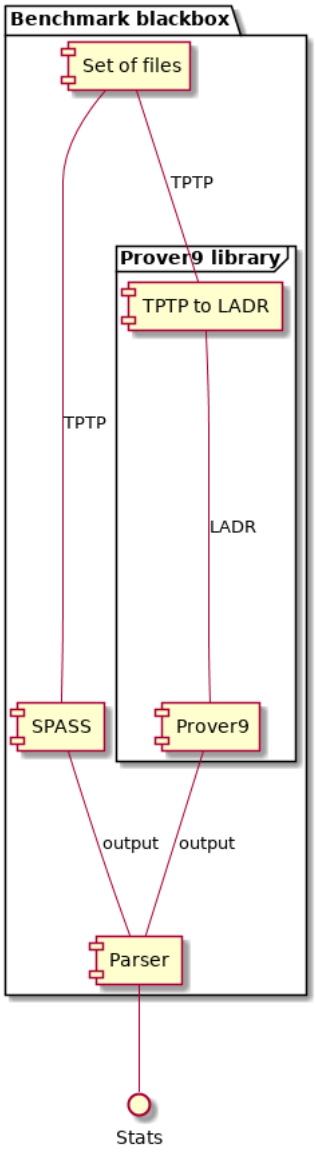
Więcej statystyk może zostać zebrany przez parser, który bada wyjście provera (sekcja **??**). Zbadane zostaną statystyki proverów ze względu na następujące właściwości formuły SAT:

- SAT type (CNF, FOF)

- number of clauses (CNF) - w składni TPTP jest to liczba użytych słów `cnf`
- number of atoms - ilość literałów połączonych operatorem *lub*
- maximal clause size
- number of predicates - predykat ustanawia n-argumentową relację między argumentami
- number of functors
- number of variables - ilość atomów, które zaczynają się dużą literą. Jeśli zmienna X wystąpi w dwóch różnych klauzulach, traktowana jest jako 2 różne zmienne
- maximal term depth

Example in tptp syntax:

```
% total in this example: 2 clauses, 6 atoms, 4 predicates, 1 functor, 2 variables

% clause: 4 atoms, 4 predicates, 0 functors, 1 variable
cnf(predicates_examples, axiom,
  ( 'p 1' % arity 0
  | p2
  | 'p 3'(X) % arity 1, variable X
  | p4(y) % arity 1, fact y
  )).

% clause: 2 atoms, 1 predicate, 1 functor, 1 variable
cnf(clause2, negated_conjecture,
    ( p4(X)
    | p4(f(X)) % f(X) is a functor
    )).
```



Rysunek 1: Diagram komponentów systemu benchmarka

2.1 Prover9

Prover9 jest to zautomatyzowane narzędzie udowadniające dla logiki pierwszego rzędu stworzone przez Williama McCune'a.

Prover9 dostępny jest jako plik wykonywalny, przyjmuje pliki w formacie LADR. Dla uniwersalności zostanie zastosowany konwerter TPTP do LADR dostępny wraz z Proverem9

jako osobny plik wykonywalny. Ilość opcji dostępnych z lini komend jest minimanlna, jedyną ważną opcją z punktu widzenia benchmarka jest `-x - set(auto2). (enhanced auto mode)`

Oficjalna strona internetowa https://www.cs.unm.edu/~mccune/mace4/

Listing 1: Przykład pliku wejściowego w składni LADR

```
formulas(sos).

  e * x = x.
  x' * x = e.
  (x * y) * z = x * (y * z).

  x * x = e.

end_of_list.

formulas(goals).

  x * y = y * x.

end_of_list.
```

Listing 2: Przykład wyjścia Provera9

```
============================== Prover9 ===============================
Prover9 (32) version Apr-2006A, Apr 2006.
Process 28593 was started by mccune on cleo.thornwood,
Fri May  5 09:04:05 2006
The command was "../bin/prover9 -f x2.in".
============================== end of head ===========================

============================== INPUT =================================

% Reading from file x2.in


clauses(sos).
e * x = x.
x ' * x = e.
(x * y) * z = x * (y * z).
```

```
x * x = e.
end_of_list.

clauses(goals).
x * y = y * x.
end_of_list.

============================== end of input ==========================

============================== PROCESS GOALS ==========================

% Each goal clause was negated; the result (to be placed in sos):

clauses(negated_goals).
c2 * c1 != c1 * c2.
end_of_list.

============================== end of process goals ==================

============================== PROCESS INITIAL CLAUSES ===============

% Clauses before input processing:

clauses(usable).
end_of_list.

clauses(sos).
1 e * x = x.  [input].
2 x ' * x = e.  [input].
3 (x * y) * z = x * (y * z).  [input].
4 x * x = e.  [input].
5 c2 * c1 != c1 * c2.  [clausify].
end_of_list.

clauses(demodulators).
end_of_list.

Predicate elimination: (none).

Term ordering decisions:
Relation symbol precedence:  lex([ = ]).
Function symbol precedence:  lex([ e, c1, c2, *, ' ]).
```

```
After inverse_order: Function symbol precedence:  lex([ e, c1, c2, *, ' ]).
Unfolding symbols: (none).

Auto inference settings:
  % set(paramodulation).  % (positive equality literals)
    % set(paramodulation) -> set(back_demod).

Reasonable options, based on the structure of the clauses:
  (nothing changed)

============================== end of process initial clauses ========

============================== CLAUSES FOR SEARCH ====================

% Clauses after input processing:

clauses(usable).
end_of_list.

clauses(sos).
6 e * x = x.  [input].
7 x ' * x = e.  [input].
8 (x * y) * z = x * (y * z).  [input].
9 x * x = e.  [input].
10 c2 * c1 != c1 * c2.  [clausify].
end_of_list.

clauses(demodulators).
6 e * x = x.  [input].
7 x ' * x = e.  [input].
8 (x * y) * z = x * (y * z).  [input].
9 x * x = e.  [input].
end_of_list.

clauses(denials).
end_of_list.

============================== end of clauses for search =============

============================== SEARCH ================================

% Starting search at 0.00 seconds.
```

```
given #1 (wt=5): 6 e * x = x.  [input].

given #2 (wt=6): 7 x ' * x = e.  [input].

given #3 (wt=11): 8 (x * y) * z = x * (y * z).  [input].

given #4 (wt=5): 9 x * x = e.  [input].

given #5 (wt=7): 10 c2 * c1 != c1 * c2.  [clausify].

given #6 (wt=8): 11 x ' * (x * y) = y.  [para(7(a,1),8(a,1,1)),demod(6(2)),flip(a)].

given #7 (wt=4): 19 x ' = x.  [back_demod(15),demod(17(4))].

given #8 (wt=5): 20 x * e = x.  [back_demod(17),demod(19(1))].

given #9 (wt=7): 12 x * (x * y) = y.  [para(9(a,1),8(a,1,1)),demod(6(2)),flip(a)].

given #10 (wt=9): 13 x * (y * (x * y)) = e.  [para(9(a,1),8(a,1)),flip(a)].

given #11 (wt=11): 21 x * (y * (x * (y * z))) = z.
↪  [back_demod(16),demod(19(2),8(4))].

given #12 (wt=7): 23 x * (y * x) = y.
↪  [para(13(a,1),12(a,1,2)),demod(20(2)),flip(a)].

% Operation * is associative-commutative; redundancy checks enabled.

============================== PROOF =================================

% Proof 1 at 0.00 (+ 0.00) seconds.
% Length of proof is 15.
% Level of proof is 7.
% Maximum clause weight is 11.
% Given clauses 12.

6 e * x = x.  [input].
7 x ' * x = e.  [input].
8 (x * y) * z = x * (y * z).  [input].
9 x * x = e.  [input].
10 c2 * c1 != c1 * c2.  [clausify].
```

```
11 x ' * (x * y) = y.  [para(7(a,1),8(a,1,1)),demod(6(2)),flip(a)].
12 x * (x * y) = y.  [para(9(a,1),8(a,1,1)),demod(6(2)),flip(a)].
13 x * (y * (x * y)) = e.  [para(9(a,1),8(a,1)),flip(a)].
15 x ' ' * e = x.  [para(7(a,1),11(a,1,2))].
17 x ' * e = x.  [para(9(a,1),11(a,1,2))].
19 x ' = x.  [back_demod(15),demod(17(4))].
20 x * e = x.  [back_demod(17),demod(19(1))].
23 x * (y * x) = y.  [para(13(a,1),12(a,1,2)),demod(20(2)),flip(a)].
28 x * y = y * x.  [para(23(a,1),12(a,1,2))].
29 $F.  [resolve(28,a,10,a)].

============================== end of proof ==========================

============================== STATISTICS ============================

Given=12. Generated=118. Kept=23. proofs=1.
Usable=8. Sos=3. Demods=12. Denials=0. Limbo=2, Disabled=14. Hints=0.
Weight_deleted=0. Literals_deleted=0.
Forward_subsumed=95. Back_subsumed=0.
Sos_limit_deleted=0. Sos_displaced=0. Sos_removed=0.
New_demodulators=21 (0 lex), Back_demodulated=9. Back_unit_deleted=0.
Demod_attempts=683. Demod_rewrites=156.
Res_instance_prunes=0. Para_instance_prunes=0. Basic_paramod_prunes=0.
Nonunit_fsub_feature_tests=0. Nonunit_bsub_feature_tests=0.
Megabytes=0.03.
User_CPU=0.00, System_CPU=0.00, Wall_clock=0.

============================== end of statistics =====================

============================== end of search =========================

THEOREM PROVED

Exiting with 1 proof.

Process 28593 exit (max_proofs) Fri May  5 09:04:05 2006
```

2.1.1 TPTP to LADR

W bibliotece LADR, która jest załączona wraz ze źródłami Provera9, dostępny jest translator składni TPTP do LADR. Jest to plik wykonywalny. Na standardowe wejście przyjmuje TPTP, na standardowy wyjście podaje LADR.

Alternatywą do konwertera dostarczanego razem z prover9 jest program `tptp2X` dostarczanego przez TPTP.

2.2 SPASS

SPASS Theorem Prover jest narzędziem do automatycznego dowodzenia twierdzeń, należących do rachunku predykatów pierwszego rzędu.

SPASS nie korzysta z zewnętrznych bibliotek, dostępny jest jako plik wykonywalny. Akceptuje pliki w składni TPTP lub swojej własnej. SPASS udostępnia wiele opcji z poziomu lini komend. Z punktu widzenia benchmarka istotnymi są:

- TODO

Wszystkie opcje linii komend https://webspass.spass-prover.org/help/options.html
Oficjana strona internetowa https://webspass.spass-prover.org/

Listing 3: Przykład pliku wejściowego w składni SPASS

```
begin_problem(Sokrates1).

list_of_descriptions.
name({*Sokrates*}).
author({*Christoph Weidenbach*}).
status(unsatisfiable).
description({* Sokrates is mortal and since all humans are mortal, he is mortal too.
↪  *}).
end_of_list.

list_of_symbols.
 functions[(sokrates,0)].
```

```
 predicates[(Human,1),(Mortal,1)].
end_of_list.

list_of_formulae(axioms).

formula(Human(sokrates),1).
formula(forall([x],implies(Human(x),Mortal(x))),2).

end_of_list.

list_of_formulae(conjectures).

formula(Mortal(sokrates),3).

end_of_list.

end_problem.
```

Listing 4: Przykład wyjścia SPASS

```
--------------------------SPASS-START-----------------------------
Input Problem:
1[0:Inp] ||  -> Human(sokrates)*.
2[0:Inp] || Mortal(sokrates)* -> .
3[0:Inp] || Human(u)* -> Mortal(u).
 This is a monadic Horn problem without equality.
 This is a problem that has, if any, a finite domain model.
 There are no function symbols.
 This is a problem that contains sort information.
 The conjecture is ground.
 The following monadic predicates have finite extensions: Human.
 Axiom clauses: 2 Conjecture clauses: 1
 Inferences: IEmS=1 ISoR=1 IORe=1
 Reductions: RFMRR=1 RBMRR=1 RObv=1 RUnC=1 RTaut=1 RSST=1 RSSi=1 RFSub=1 RBSub=1
 ↪  RCon=1
 Extras    : Input Saturation, Always Selection, No Splitting, Full Reduction,
 ↪  Ratio: 5, FuncWeight: 1, VarWeight: 1
 Precedence: Mortal > Human > sokrates
 Ordering  : KBO
Processed Problem:
```

```
Worked Off Clauses:

Usable Clauses:
1[0:Inp] ||  -> Human(sokrates)*.
2[0:Inp] || Mortal(sokrates)* -> .
3[0:Inp] Human(u) ||  -> Mortal(u)*.
SPASS V 3.9
SPASS beiseite: Proof found.
Problem: input
SPASS derived 1 clauses, backtracked 0 clauses, performed 0 splits and kept 4
↪  clauses.
SPASS allocated 85013 KBytes.
SPASS spent     0:00:00.03 on the problem.
                0:00:00.01 for the input.
                0:00:00.01 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.

-------------------------SPASS-STOP------------------------------
```

2.3 TPTP

TPTP - *ang. (Thousands of Problems for Theorem Provers)* - to biblioteka problemów wykorzystywanych do testowania systemów ATP (Automated Theorem Proving). Jednocześnie jest to nazwa formatu, w którym zapisywane są te testy. TPTP udostępnia te problemy na oficjalnej stronie intenetowej. Razem z TPTP istnieje TSTP (*ang. Thousands of Solutions from Theorem Provers*) - biblioteka rozwiązań problemów. Te problemy są sklasyfikowane przez domeny (3 literowe skróty), przykładowo LCL - Logic Calculi, COL - Combinatory Logic

W formacie TPTP można zapisywać TPI (TPTP Process Instruction), THF (Typed Higher-order Logic), TFF (Typed First-order Logic), FOF, CNF. Celem benchmarka jest badanie proverów logiku pierwszego rzędu, więc interesują nas CNF, TFF, FOF (TFF/FOF with external clausifiers).

2.3.1 Wybrane elementy składni TPTP

- The syntax for atoms is that of Prolog: variables start with upper case letters, atoms and terms are written in prefix notation, uninterpreted predicates and functors either start with lower case and contain alphanumerics and underscore, or are in 'single quotes'.
- Each logical formula is wrapped in an annotated formula structure of the form `language(name,role,formula,source,[useful_info])`
  - role gives the user semantics of the formula, one of `axiom`, `hypothesis`, `definition`, `assumption`, `lemma`, `theorem`, `corollary`, `conjecture`, `negated_conjecture`, `plain`, `type`, and `unknown`. Axiom-like formulae are those with the roles `axiom`, `hypothesis`, `definition`, `assumption`, `lemma`, `theorem`, and `corollary`. They are accepted, without proof, as a basis for proving conjectures in THF, TFF, and FOF problems. In CNF problems the axiom-like formulae are accepted as part of the set whose satisfiability has to be established. `conjecture` occur in only THF, TFF, and FOF problems, and are to all be proven from the axiom(-like) formulae. A problem is solved only when all conjectures are proven. TPTP problems never contain more than one conjecture. `negated_conjectures` are formed from negation of a `conjecture`, typically in FOF to CNF conversion.
  - The `useful_info` field of an annotated formula is optional, and if it is not used then the `source` field becomes optional. The `source` field is used to record where the annotated formula came from, and is most commonly a file record or an inference record.
- The language also supports interpreted predicates and functors. These come in two varieties: defined predicates and functors, whose interpretation is specified by the TPTP language, and system predicates and functors, whose interpretation is ATP system specific. The defined predicates recognized so far are `$true` and `$false` `=` and `!=` `$distinct` (only TFF language) and arithmetic predicates (only TFF and THF). Interpreted predicates and functors are syntactically distinct from uninterpreted ones - they are `=` and `!=`, or start with a $, a ", or a digit. Non-variable symbols can be given a type globally, in the formula with role type. The defined types are `$o` - the Boolean type, `$i` - the type of individuals, `$real` - the type of reals, `$rat` - the type of rational, and `$int` - the type of integers. New types are introduced in formulae with the type role, based on `$tType` - the type of all types.

- The universal quantifier is `!`, the existential quantifier is `?`, and the lambda binder is `^`. Quantified formulae are written in the form `Quantifier [Variables] : Formula`
- The binary connectives are infix `|` for disjunction, infix `&` for conjunction, infix `<=>` for equivalence, infix `=>` for implication, infix `<=` for reverse implication, infix `<~>` for non-equivalence (XOR), infix `~|` for negated disjunction (NOR), infix `~&` for negated conjunction (NAND), infix `@` for application. The only unary connective is prefix `~` for negation
- Arithmetic system are used in only the THF and TFF languages. This includes: `$real` (real number) `$rat` (rational) `$to_int` (cast to int) `$to_rat` `$to_real` `$is_int` `$is_rat` `$is_real`, unary operators: `$floor` `$round` `$ceiling` `$truncate`, comparison of 2 numbers: `=` `$less` `$lesseq` `$greater` `$greatereq` `$uminus` `$sum` `$difference` `$product` `$quotient` `$quotient_e` (e for Euclidean quotient) `$quotient_t` (t for truncate) `$quotient_f` (f for floor) `$distinct` `$remainder_e` `$remainder_t` `$remainder_f`

Oficjalna strona internetowa http://www.tptp.org
Pełny spis domen http://www.tptp.org/cgi-bin/SeeTPTP?Category=Documents&File=THFSynopsis
BNF składni TPTP http://www.tptp.org/TPTP/SyntaxBNF.html
Tutorial składni TPTP znajduje się pod linkiem http://www.tptp.org/TPTP/TR/TPTPTR.shtml
Aby poznać jak zapisywana jest formuła, polecam przeczytać *The Formulae Section*

2.3.2 Narzędzia dodatkowe TPTP

TPTP4X (napisane w c), TPTP2X (napisane w prologu) for reformatting, transforming, and generating TPTP problem files. Nie są używane w tym projekcie ale warto o nich wspomnieć, dostarczają wiele funkcjonalności. Przykładowe możliwości:

- konwertowanie FOF do CNF
- konwertowanie TPTP do składni prover9, dimacs, otter, dfg i więcej
- optymalizacja FOF, CNF za pomocą różnych algorytmów
- zmień porządek formuły CNF

2.4 Parser

Zadaniem parsera jest wydobycie dodatkowych informacji statystycznych o przebiegu działana proverów, na podstawie ich wyjścia.
Każdy prover podaje inne dane na wyjściu, dostępne statystyki podane są w tabeli poniżej.
Statystyki zostaną podane w formacie json.

Tablica 1: Dostępne statystyki dla różnych proverów

Prover	SPASS	Prover9
SAT spełnialny	dostępny	dostępny
TODO		

2.5 Użycie i konfiguracja

Ze względu na mnogośc opcji, większość opcji zawarta jest w pliku konfiguracyjnym **??** w formacie *toml*.

Najpierw definiowana jest lista wejść (`testInput`). Wejście to zbiór plików, które można jednoznacznie zidentyfikować za pomocą nazwy (`name`). Następnie definiowana jest lista zestawów testowych (`testSuite`). Zestaw testowy definiuje parametry wspólne dla kilku przypadków testowych (`testCase`) np. ścieżka do pliku wykonywalnego. Każdy zestaw testowy posiada listę przypadków testowych. Każdy przypadek testowy definiuje w jakim formacie oczekuje wejście. Jeśli formaty są różne, konflikt jest rozwiązywany przy pomocy dostępnych translatorów. Opcje do testowania są definiowane jako lista. Plik wejściowy może zosać podany przez standardowe wejście, przez opcję lini komend lub jako ostatni argument w komendzie.

```
testSuite.executable testSuite.options testSuite.testCase.options [input_after_option
↪  file_path] [file_path]
```

2.5.1 Wspierane funkconalności

- ścieżka do pliku wykonywalnego może być podana w pliku konfiguracyjnym, lub może być zawarta w zmiennyj środowiskowej `PATH` (ścieżka w pliku ma pierwszeństwo)

- definiowanie opcji linii komend do testowania pliku wykonywalnego
- definiowanie listy źródeł do testów. Źródłem do testów mogą być tylko pliki tesktowe
- definiowanie które wejścia mają byś przetestowane w przypadku testowym
  - testuj tylko wymienione - opcja `include_only`,
  - testuj wszystkich oprócz - opcja `exclude`,
  - testuj wszystkie zdefiniowane wejścia - nie podając żadnej z opcji
- pozycja nazwy pliku źródłowego może być ustawiona w następujący sposób
  - domyślnie plik podawany jest na standardowe wejście
  - podaj plik jako ostatni argument `input_as_last_argument`
  - podaj plik jako argument po opcji `input_after_option`
- TODO: wyniki zapisywane są jako plik *json* do katalogu wyściowego zdefiniowanego w pliku konfiguracyjnym.

2.5.2 Ograniczenia

- podanie kilku plików wejściowych naraz dla jednej testownej komendy nie jest możliwe, np. `-o file1.p -o file2.p`

Listing 5: Przykład pliku konfiguracyjnego benchmarka

```
[general]
# generate output to
output_dir="benchmark-output.json"
test_case_timeout = 300

[[translators]]
from_format="TPTP"
to_format="LADR"
extension="in"
PATH="../provers/LADR-2009-11A/bin"
executable="tptp_to_ladr"
options=[]
#input_after_option="-TPTP"
```

```
#input_as_last_argument=false
#output_after_option=""

[[testInputs]]
# for specifying include/exclude in testCase
name="tptp_files"
# path to files
path="tests/data/tptp"
# wildcards supported
files=["*.p"]
# currently only TPTP
format="TPTP"

#[[testInputs]]
## for specifying include/exclude in testCase
#name="prover9_files"
## path to files
#path="tests/data/prover9"
## wildcards supported
#files=["*.in"]
## currently only TPTP
#format="LADR"

# PROVER9
[[testSuites]]
name="prover9 tests"
# name available in PATH
executable="prover9"
# append to path env variable, comment or leave empty to disable
PATH="../provers/LADR-2009-11A/bin"
version="2009-11A"
# options always appended to executable, comment or leave empty to disable
#options=[""]

[[testSuites.testCases]]
# metadata
name="Prover9 with selected test files"
# what file format is accepted by prover, input will be automatically translated if
↪  possible
format="LADR"
# mutually exclusive options input_as_last_argument, input_after_option
#input_as_last_argument=true
```

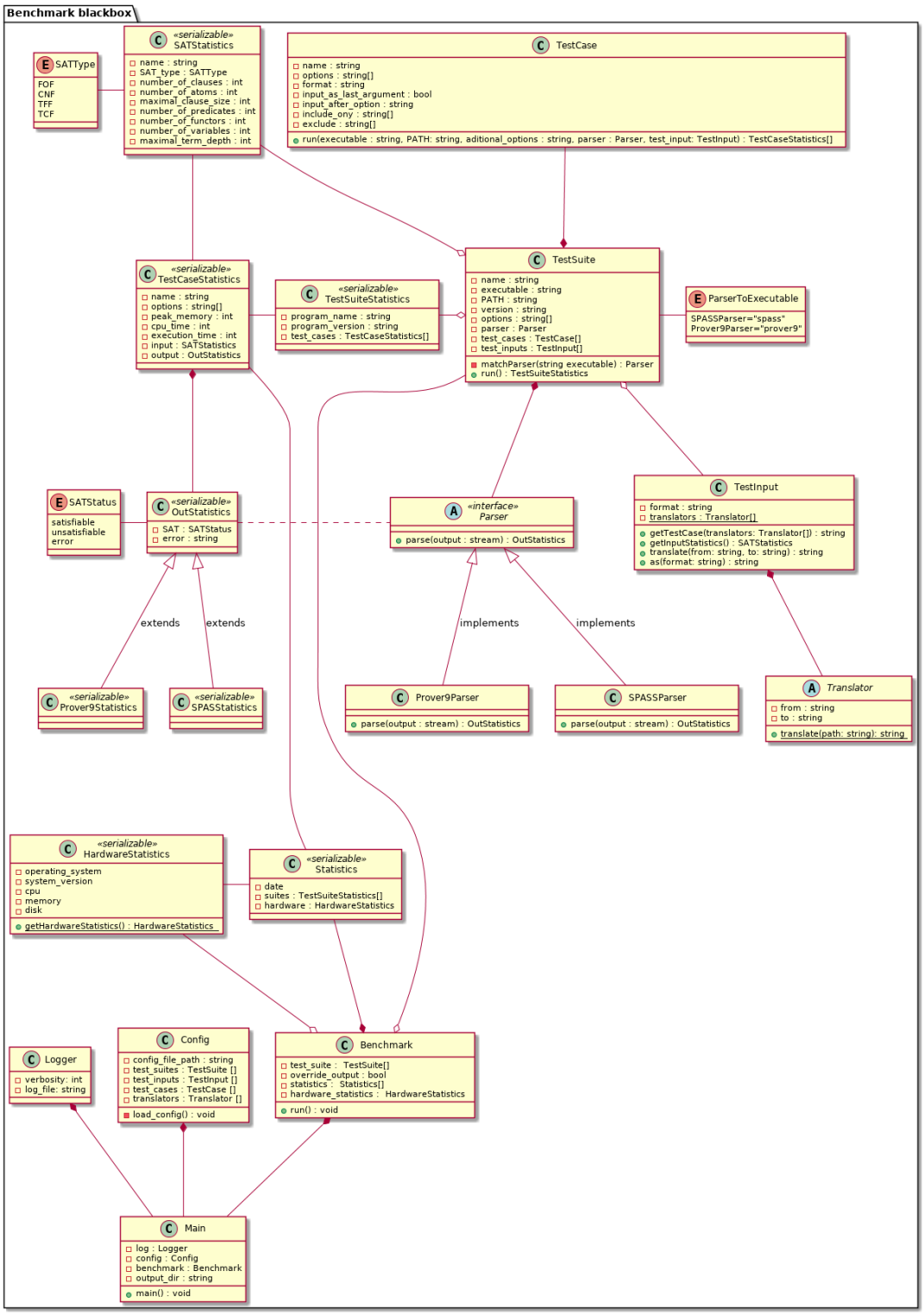
```
#input_after_option="-o"

# mutually exclusive: include or exclude inputs for test. Comment both to test all
#include_only=["prover9_files"]
#exclude=[""]

# list of options to test,
# set to [] or add empty string as option to run testcase without options
options=[]

# SPASS
[[testSuites]]
name="spass tests"
executable="SPASS"
PATH="../provers/spass39/"
version="3.9"

[[testSuites.testCases]]
name="SPASS with selected test files"
format="TPTP"
options=[]
input_after_option="-TPTP"
```

Listing 6: Przykładowe komendy testowe

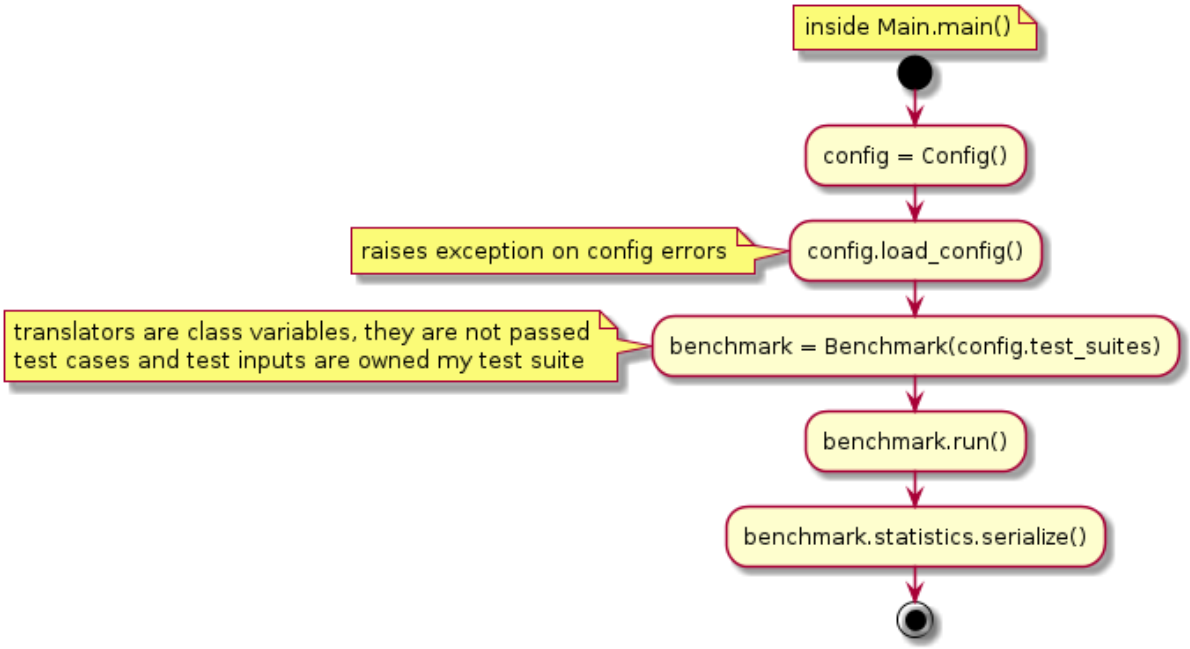
```
# ...
[[testSuites]]
executable="ls"
options=["-1"]
# ...

[[testSuites.testCases]]
options=[""]
# test cases:
# ls -1

[[testSuites.testCases]]
options=["", "-r -l"]
# test cases:
# ls -1
```

```
# ls -1 -r -l
```

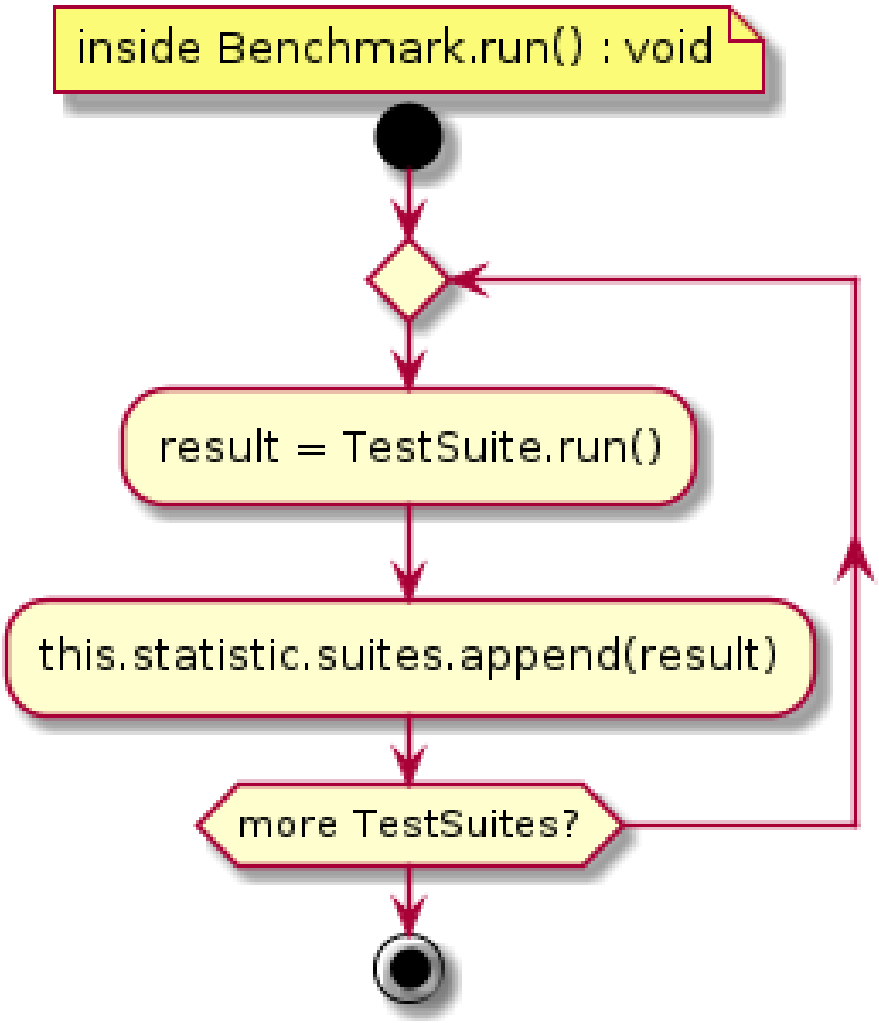
2.6 Diagramy



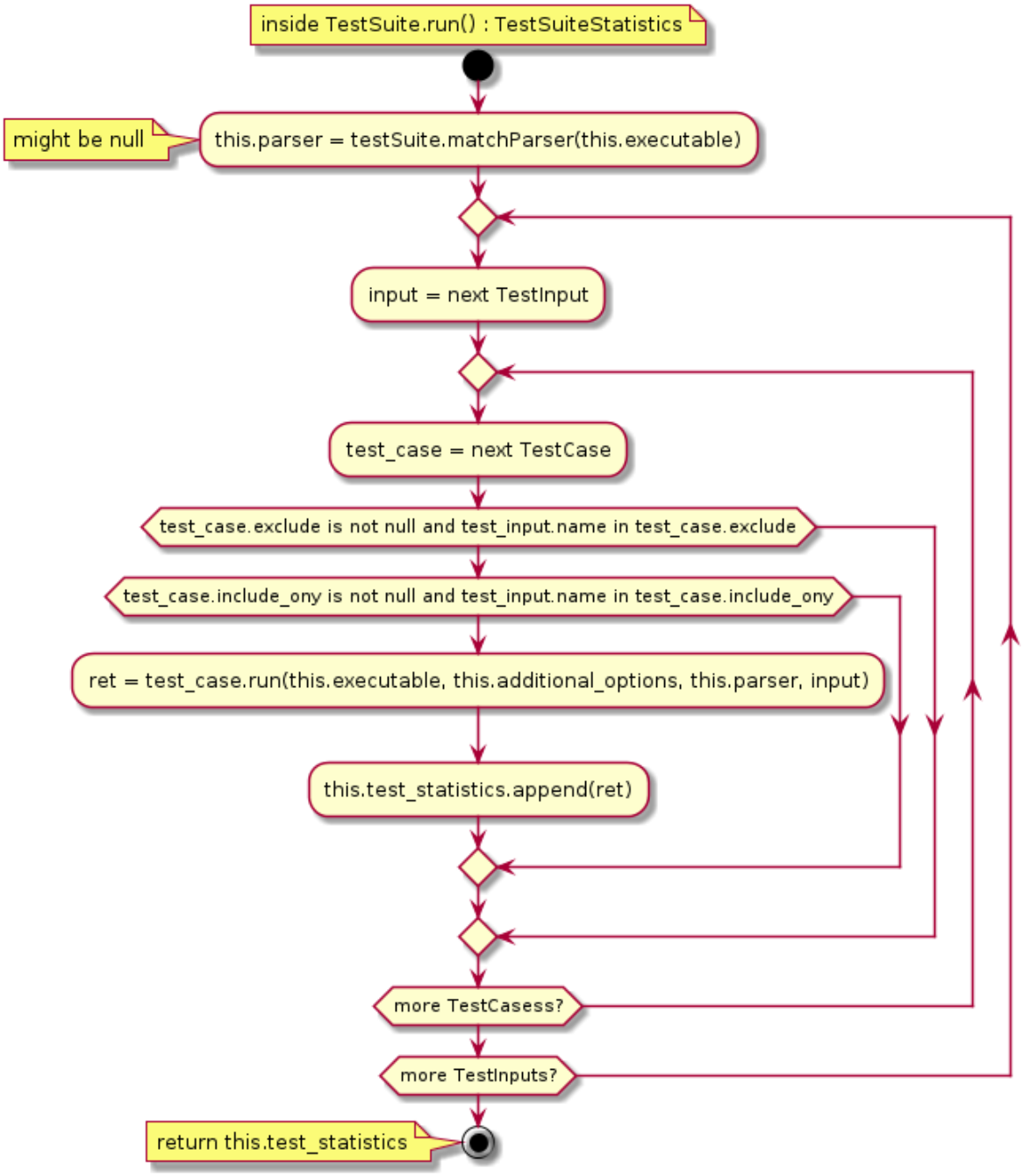
Rysunek 2: Diagram klas



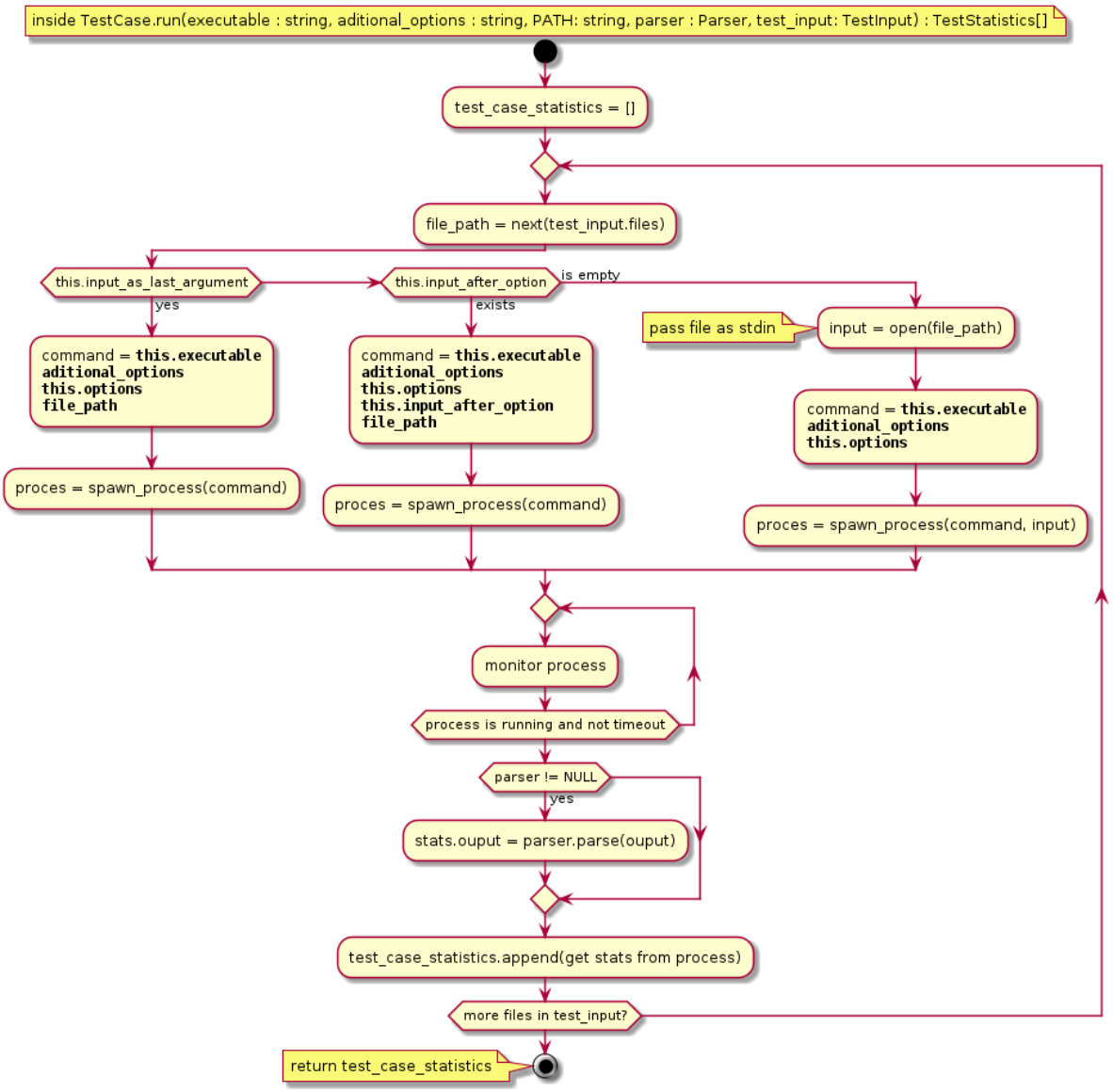
Rysunek 3: Diagram aktywności



Rysunek 4: Diagram aktywnośći



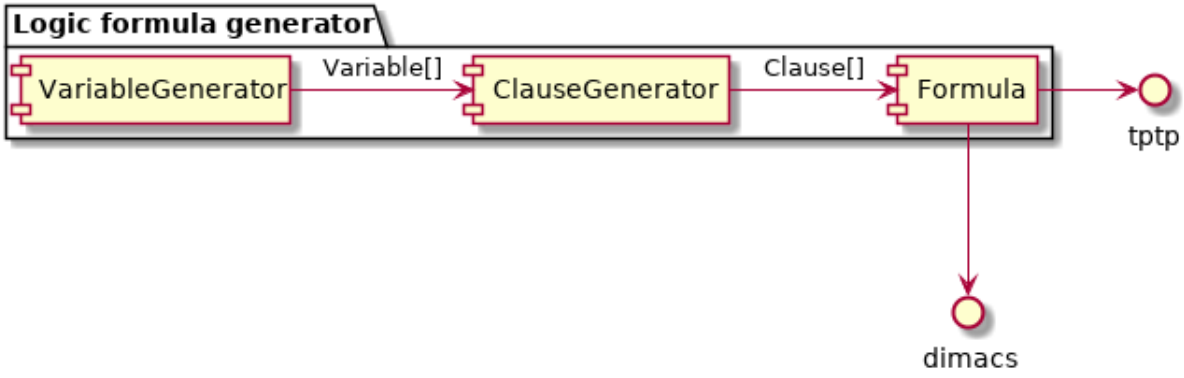
Rysunek 5: Diagram aktywnośći



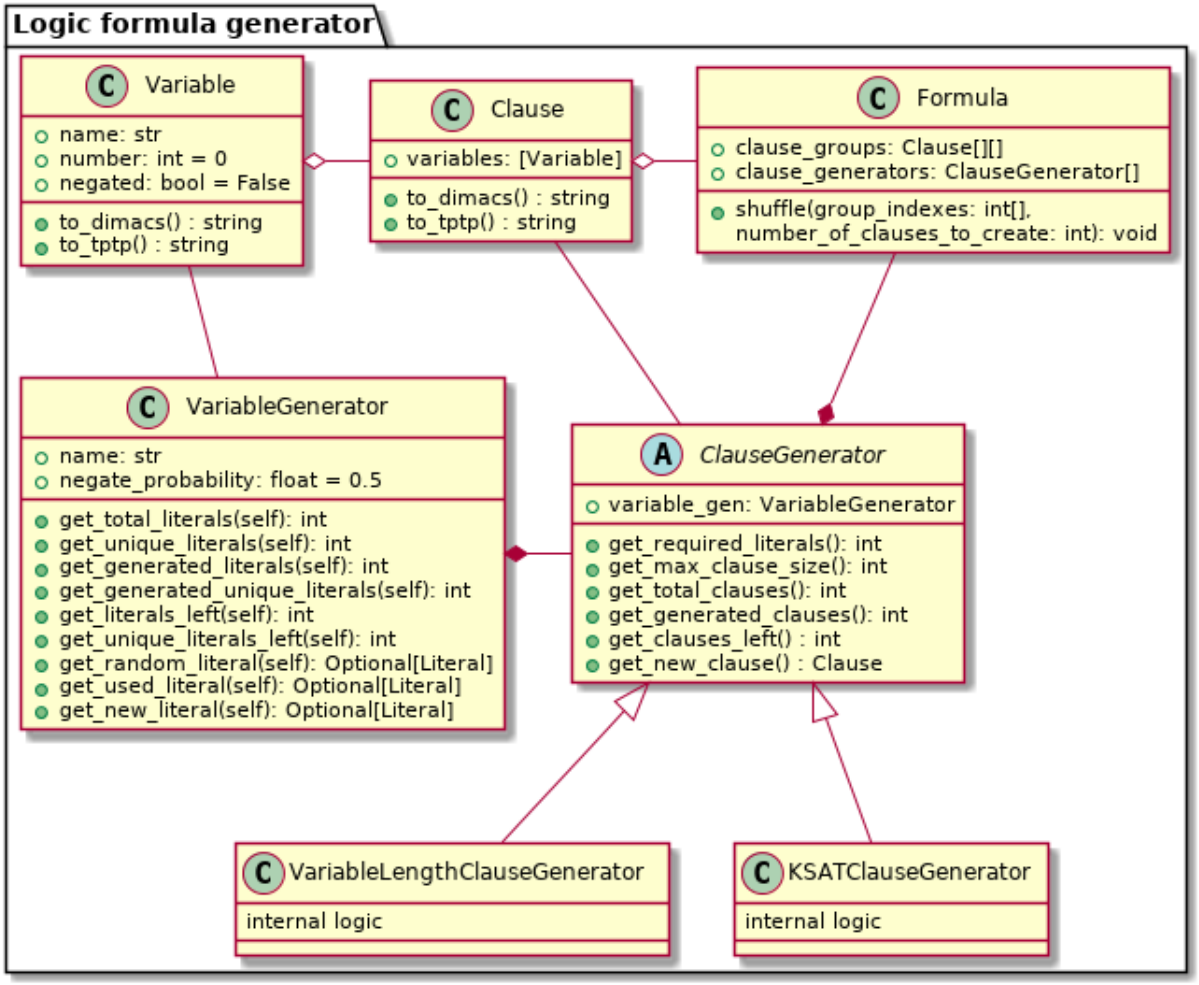
Rysunek 6: Diagram aktywnośći

3 Generator formuł logicznych

Losowy generator formuł SAT. Generator może generować formuły CNF oraz . Generator może generować formuły w formacie DIMACS oraz TPTP.



Rysunek 7: Komponenty generatora CNF



Rysunek 8: Diagram klas generatora CNF

Założenia ogólne:

- na jedną formułę, składa się w niezależnych grup kauzul (w z przedziału $[1, \infty]$). Wygenerowanie jednej grupy klauzul polega na ponownym uruchomieniu generatora z innymi parametrami. Przy czym:
  - grupa klauzul nie współdzieli zbioru zmiennych
  - grupa może posiadać całkowicie inne parametry sterujące
- po wygenerowaniu, grupa klazul może być wymieszana z inną grupą klauzul. Mieszanie polega na:
  - utworzeniu nowych klauzul bazując na zmiennych występujących w obu grupach - wprowadza to nowe zależności między istniejącymi już zmiennymi

Szczegóły implementacyjne:

- logika generatora (backend) używa ograniczeń sztywnych. Na przykład, generator zmiennych musi wygenerować **dokładnie** n zmiennych, generator klauzul musi wygenerować **dokładnie** m klauzul itp.
- ograniecznia miękkie, tzn. wygeneruj formułę z **około** n zmiennych i **około** m klauzul, uzyskiwane są przez wygenerowanie dokładnych warości na frontendzie i uruchomienie generatora z dokładnymi wartościami
- generatory są obiektami tylko do odczytu

3.1 Generator zmiennych (*VariableGenerator*)

Generator zmiennych ma za zadanie podanie dokładnie n zmiennych, w tym m różnych.

Parametry sterujące:

- nazwa zmiennej
- m - ilość różnych zmiennych

- n - ilość zmiennych do wygenerowania, ilość różnych zmiennych jest mniejsza lub równa ilości zmiennych
- prawdopodobieństwo zanegowania zmiennej

Ograniczenia:

1. prawdopodobieństwo zanegowania jest z zakresu $[0, 1]$
2. ilość zmiennych jest większa lub równa niż ilość różnych zmiennych $n >= m$

3.2 Generator klauzul (*ClauseGenerator*)

Bazując na zmiennych podanych przez generator zmiennych, generuje dokładnie k klauzul. Generator zmiennych musi zostać całkowicie wyczerpany. Klauzule które pochodzą z jednego generatora, nazywane są grupą klauzul.

Parametry sterujące:

- generator zmiennych
- ilość klauzuj do wygenerowania
- maksymalny rozmiar klauzuli

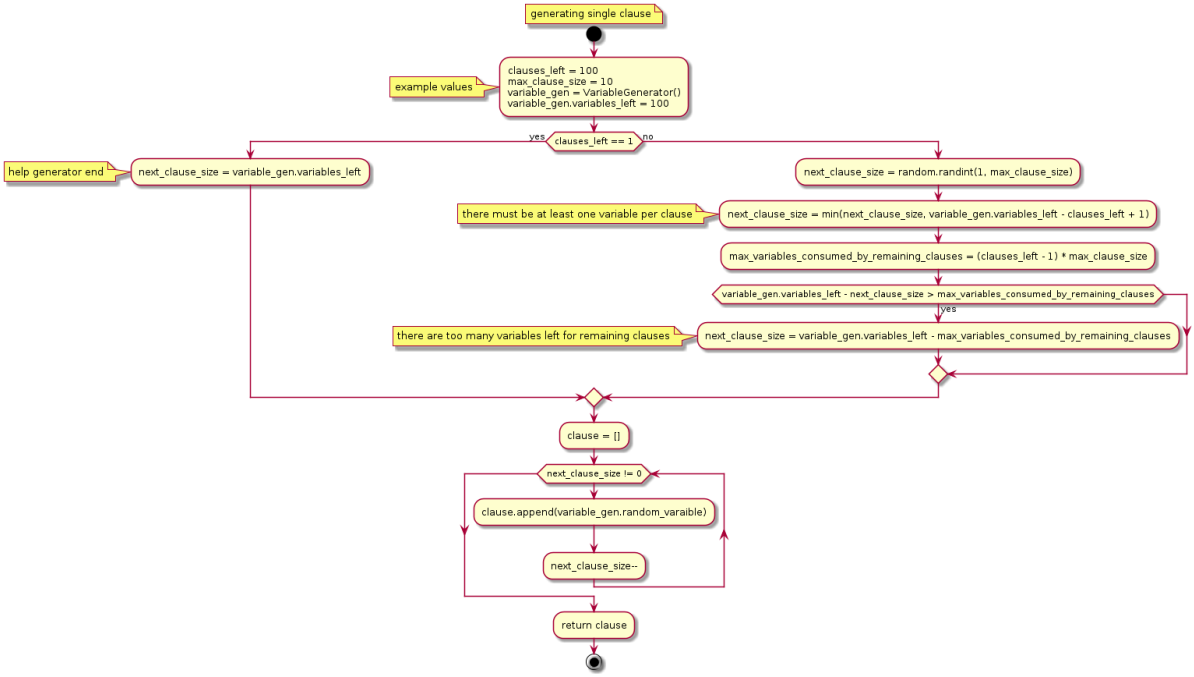
Ograniczenia:

1. na klauzulę przypada co najmniej jedna zmienna
2. ilość zmiennych wymaganych przez generator klauzul musi być większa niż ilość zmiennych dostaczanych przez generator zmiennych (cały generator zmiennych musi zostać skonsumowany)
3. każdy literał w pojedyńczej kaluzuli jest różny - nie może wystąpić: $(p1 \vee p1) \wedge (p1 \vee p2)$
4. każda klauzula jest różna - nie może wystąpić: $(p1 \vee p2) \wedge (p1 \vee p2)$

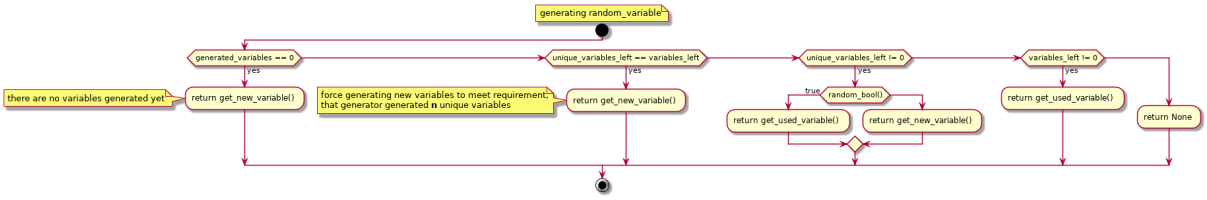
3.3 Formuła (*Formula*)

Formuła powstaje w wyniku kilkukrotnego uruchomiania generatora klauzul **??**. Zbiera ona wszystkie grupy klauzul i umożliwia mieszanie ich.

3.4 Algorytm generowania CNF



Rysunek 9: Algorytm generowania pojedyńczej klauzuli



Rysunek 10: Algorytm generowania losowej zmiennej

4 Rachunek zdań - nazewnictwo w kontekście DIMACS

Literał to zmienna lub jej zanegowanie

Zmienna to ciąg znaków alfanumerycznych

Klauzula to zbiór literałów połączonych logicznym znakiem $i \wedge$

```
c DIMACS CNF formula example 1
c
p cnf 3 2
1 -3 0
2 3 -1 0

c powyższa formuła składa się z:
c 2 klauzuli
c 3 zmiennych: [1,2,3]
c 5 literałów
```

5 Logika pierwszego rzędu - nazewnictwo w kontekście TPTP

Term to zmienna, stała lub wynik działania funktorów na zmiennych i stałych

Atom to wyrażenie logiczne, które nie może zostać rozbite na składowe

Literał to atom lub jego zaprzeczenie

Zmienna to atom, który zaczyna się z dużej litery. Zmienna ma zasięg klauzulu. Tzn. jeśli zmienna A pojawi się w jednej klazuli kilkukrotnie jest to dalej jedna zmienna.

Zmienna singletonowa to zmienna użyta tylko raz w klauzuli

Unit clause to klauzula, która posiada tylko jeden atom

Horn clause to klaulula, która posiada co najmniej jeden pozytywny literał

RR clause - ??

Predykat jest to operator logiczny, który zwraca prawdę lub fałsz. Predykat operuje na określonej liczbie termów. Liczba ta jest stała i nazywana argumentowością predykatu (*arity*).

Funktor oprator logiczny, który zwraca term. Funktor posiada określoną argumentowość.

Constant functor to funktor o argumentowości 0

Klauzula jest to zbiór termów połączonych dysjunkcją ($\vee$)

```
% TPTP CNF formula example 1
cnf(simple_clause_1, axiom,
    ( p(f,f) | ~p(a,b) | p(X, V) | pp(X) )).

% powyższa formuła składa się z:
% 1 klauzuli: w tym 0 jednostkowa, 1 Horn
% 4 literały: [p(f,f), ~p(a,b), p(X, V), pp(X)]
% 4 atomy: [p(f,f), p(a,b), p(X, V), pp(X)]
% 2 predykatów: [p, pp] o argumentowości 1: [pp] i 2: [p]
% 3 funktorów: [f, a, b] o argumentowości 0 - funktory stałe
% 2 zmiennych: [X, V] w tym 1 zmienna singletonowo
```

```
% TPTP CNF formula example 2
cnf(simple_clause_1, axiom,
    ( p(f,f) | ~p(a,b) | p(X, V) | pp(X) )).

cnf(simple_clause_2, axiom,
    ( pp(f) | pp(X) )).

cnf(simple_clause_3, axiom,
    ( ppp )).

% powyższa formuła składa się z:
% 3 klauzuli, w tym 1 klauzula jednostkowa, 3 Horn
% 4 literały: [p(f,f), ~p(a,b), p(X, V), pp(X)]
% 4 atomy: [p(f,f), p(a,b), p(X, V), pp(X)]
% 3 predykatów: [p, pp, ppp] o argumentowości 0: [ppp], 1: [pp] i 2: [p]
% 3 funktorów: [f, a, b] o argumentowości 0 - funktory stałe
% 3 zmiennych: [X, V] w tym 2 zmienne singletonowe
```

```
% TPTP CNF formula example 3
cnf(simple_clause_1, axiom,
    ( p(f,f) | ~p(f,f) )).

% powyższa formuła składa się z:
% 1 klauzuli, w tym 0 jednowtkowa, 1 Horn
% 2 literały: [p(f,f), ~p(f,f)]
% 1 atomy: [p(f,f)]
% 1 predykatu: [p] o argumentowości 2
% 1 funktorów: [f] o argumentowości 0 - funktor stały
% 0 zmiennych
```

```
% TPTP CNF formula example 4
cnf(simple_clause_1, axiom,
    ( p(f,f) | ~p(f,f) )).

cnf(simple_clause_2, axiom,
    ( ~p(f,f) )).

% powyższa formuła składa się z:
% 2 klauzuli, w tym 1 jednostkowa, 1 Horn
% 2 literały: [p(f,f), ~p(f,f)]
% 1 atomy: [p(f,f)]
% 1 predykatu: [p] o argumentowości 2
% 1 funktorów: [f] o argumentowości 0 - funktor stały
% 0 zmiennych
```

6 Zestaw testowy

Problem: Jak ilość atomów wpływa na czas wykonania?

Zestaw to formuły CNF różnej długości

- liczba klauzul to 100, 200, 500, 1000, 2500, 5000
- stosunek atomów do klauzul wynosi 2, 3, 4, 5, 10 (2 oznacza, że jeśli liczba klauzul to 100, wtedy liczba atomów to 200)
- maksymalna długość klauzuli to ilość wszystkich zmiennych / ilość klauzul

Problem: Jak stosunek atomów bezpieczeństwa do atomów żywotnościowych wpływa na czas wykonania?

Zestaw to formuły CNF różnej długości

- liczba klauzul to 100, 200, 500, 1000, 2500, 5000
- stosunek atomów bezpieczeństwa do atomów żywotnościowych wynosi 2, 3, 4, 5, 10
- maksymalna długość klauzuli to ilość wszystkich zmiennych / ilość klauzul

Problem: Jak udział k-SAT i zmiennych wpływa na czas wykonania?
Hipoteza:

1. obecność co najmniej jednej klauzul k-SAT, gdzie k dąży do 1, sprzyja szybkiemu rozwiązaniu problemu
2. obecność co najmniej jednej klauzul k-SAT, gdzie k dąży do ∞, znacząco spowalnia rozwiązanie problemu
3. im większa obecność klazul z dużym k, tym większy czas wykonywania

Zestaw skupia się na k-SAT:

- liczba klauzul to 100, 200, 300, 400, 500, 1000, 2000, 3000, 4000, 5000

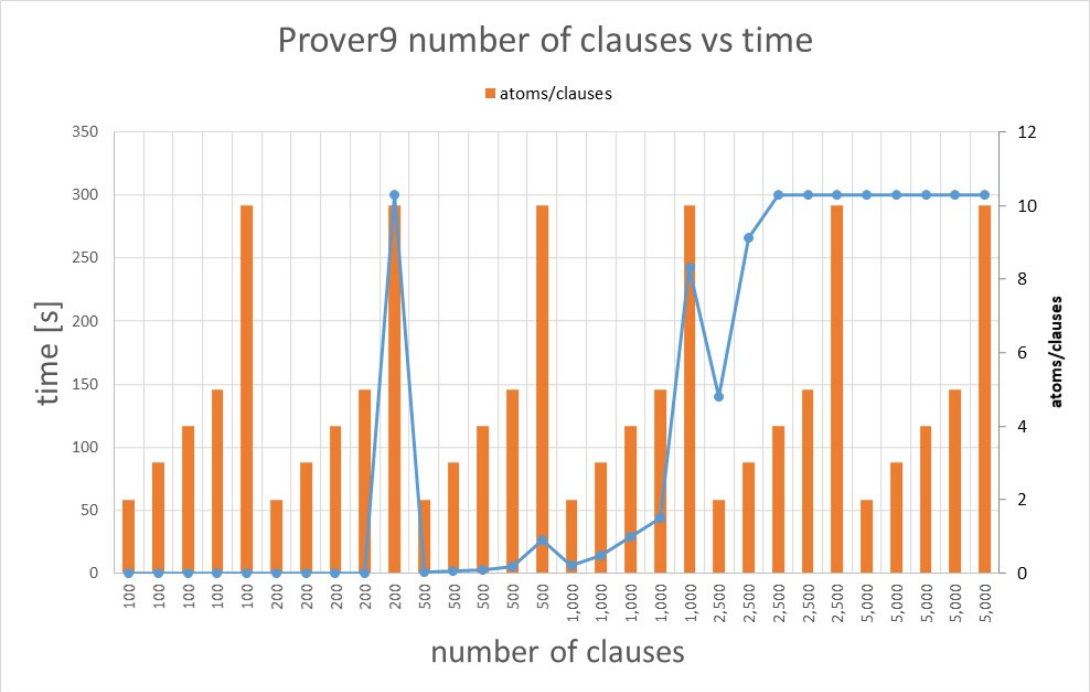
- klauzule są postaci:
  - 1,5,10,20-SAT – kada z nich stanowi 25% wszystkich klauzul (po równo)
  - 1,5,10,20-SAT – 1-SAT to 1% (minimum 1), 5,10,20-SAT po równo
  - 1,5,10-SAT – po równo
  - 1,5,10,20-SAT – 20-SAT to 1% (minimum 1), 1,5,10-SAT po równo
  - 5,10,20-SAT – po równo

7 Wnioski

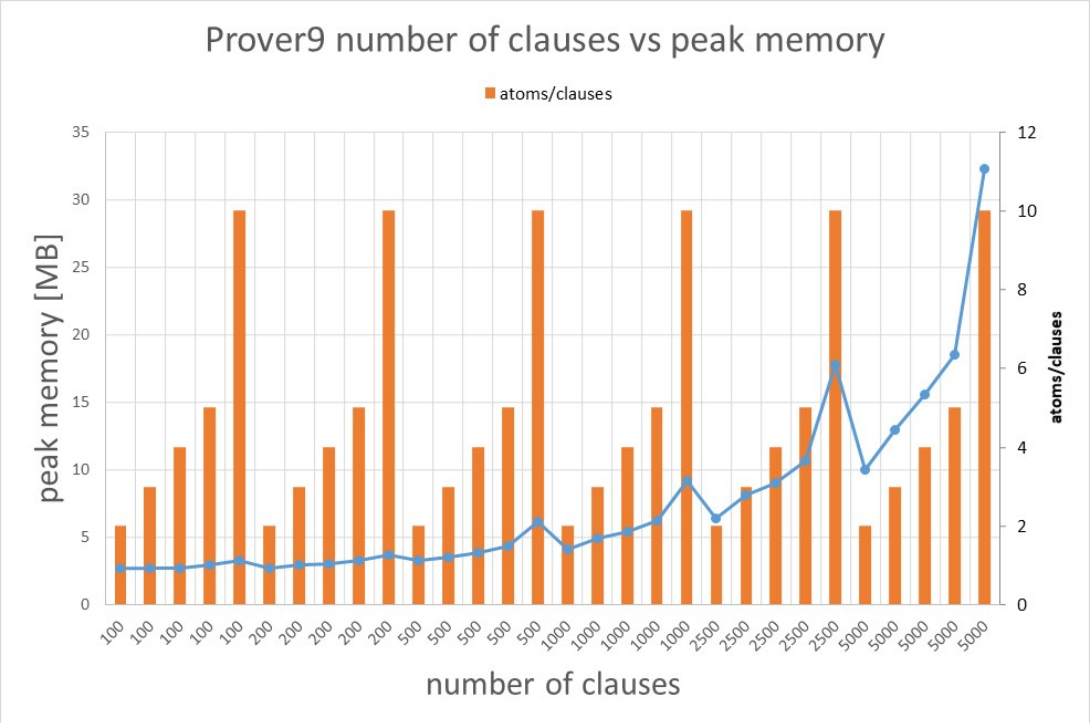
Celem projektu było zbadanie które z czynników formuł logicznych wpływają najbardziej na działanie proverów SPASS oraz Prover9. Aby odpowiedzieć na to pytanie napisany został generator formuł o zadanych parametrach które następnie były przepuszczane przez provery z użyciem strategii blackbox. Na wyjściu otrzymane zostały pliki JSON z parametrami wejścia takimi jak: liczba klauzul, atomów, predykatów, funktorów, zmiennych oraz parametrami wyjścia: czasem wykonania, maksymalnym zużyciem pamięci (peak memory) oraz statusem (spełnialna, niespełnialna, timeout). W celu ograniczenia czasu wykonywania się benchmarków wprowadzono górną granicę czasu na każdy test case: 300 sekund.

7.1 Zestaw 1

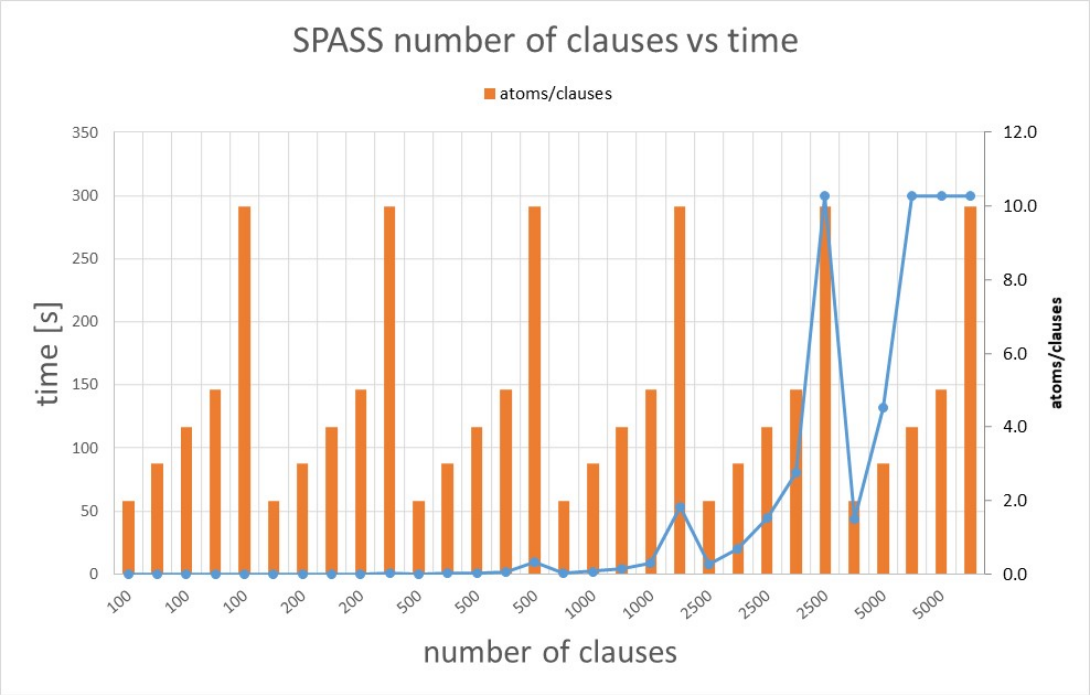
Jak widać na wykresach załączonych poniżej według oczekiwań największy wpływ na czas wykonywania/używaną pamięć największy wpływ ma ilość klauzul. Zarówno na czas wykonania oraz na zużytą pamięć zauważalny wpływ ma również stosunek liczby atomów do klauzul, pierwszy taki skok możemy zauważyć w Proverze9 dla test case'ów o 200 klauzulach, gdzie test case o 2000 atomach zakończył się timeout'em przy niewielkim wzroście pamięci. Zwiększenie się czasu wykonywania oraz zużytej pamięci widać bardzo dobrze dla test case'ów Provera9 o 500 oraz 1000 klauzulach. Dla 2500 klauzul i stosunku atomy/klauzule większego lub równego od 4 oraz dla wszystkich testcase'ów o 5000 klauzulach benchmark kończył się timeoutem. Warto zauważyć coraz bardziej stromy wzrost zużytej pamięci dla większych ilości klauzul dla Provera9. Dla SPASS już od najmniejszych pod względem ilości klauzul test case'ach zużycie pamięci jest znacząco większe nawet od największych pod względem ilości klauzul test case'ach w Proverze9 co przemawia na korzyść tego provera.



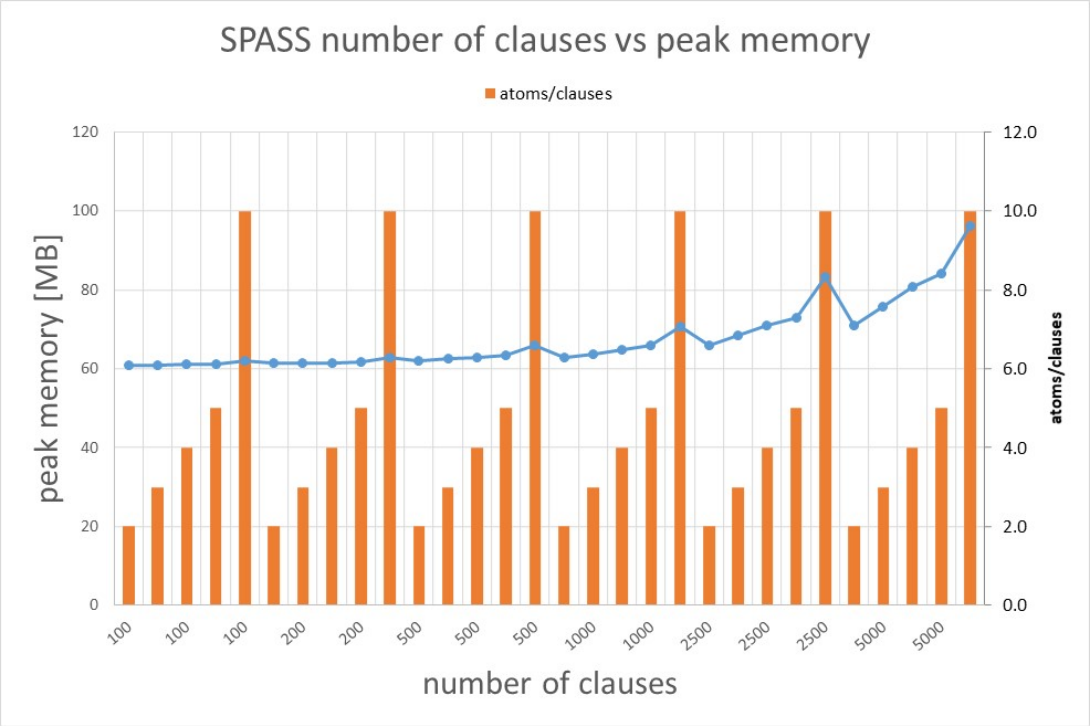
Rysunek 11: Prover9 zestaw 1 liczba klauzul vs czas



Rysunek 12: Prover9 zestaw 1 liczba klauzul vs pamięć



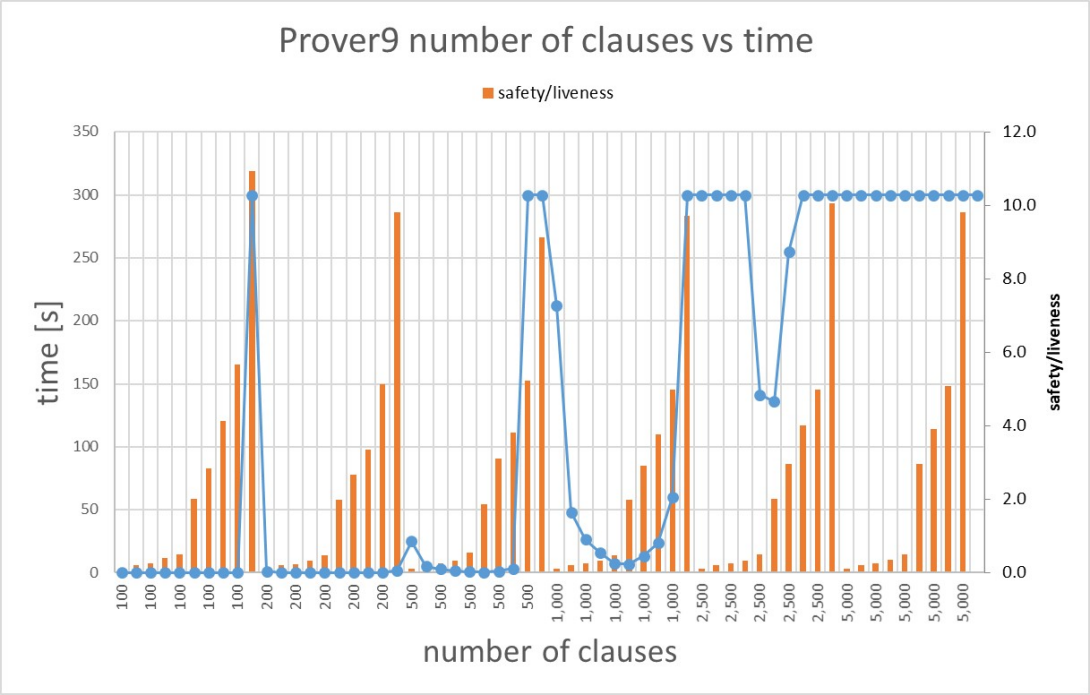
Rysunek 13: SPASS zestaw 1 liczba klauzul vs czas



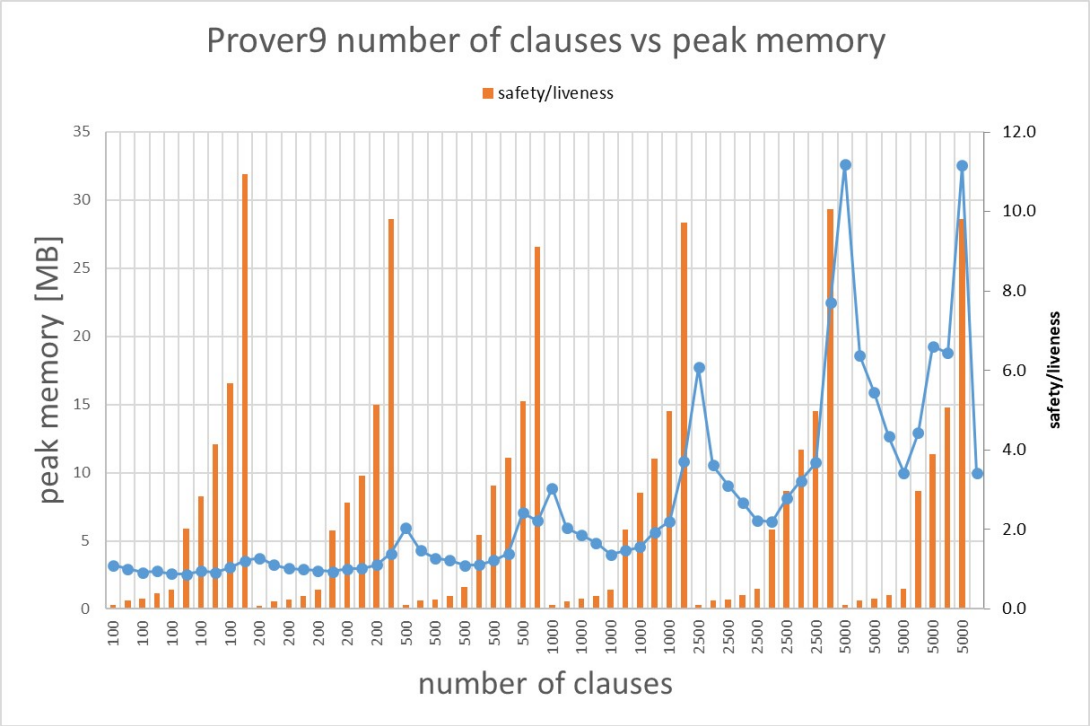
Rysunek 14: SPASS zestaw 1 liczba klauzul vs pamięć

7.2 Zestaw 2

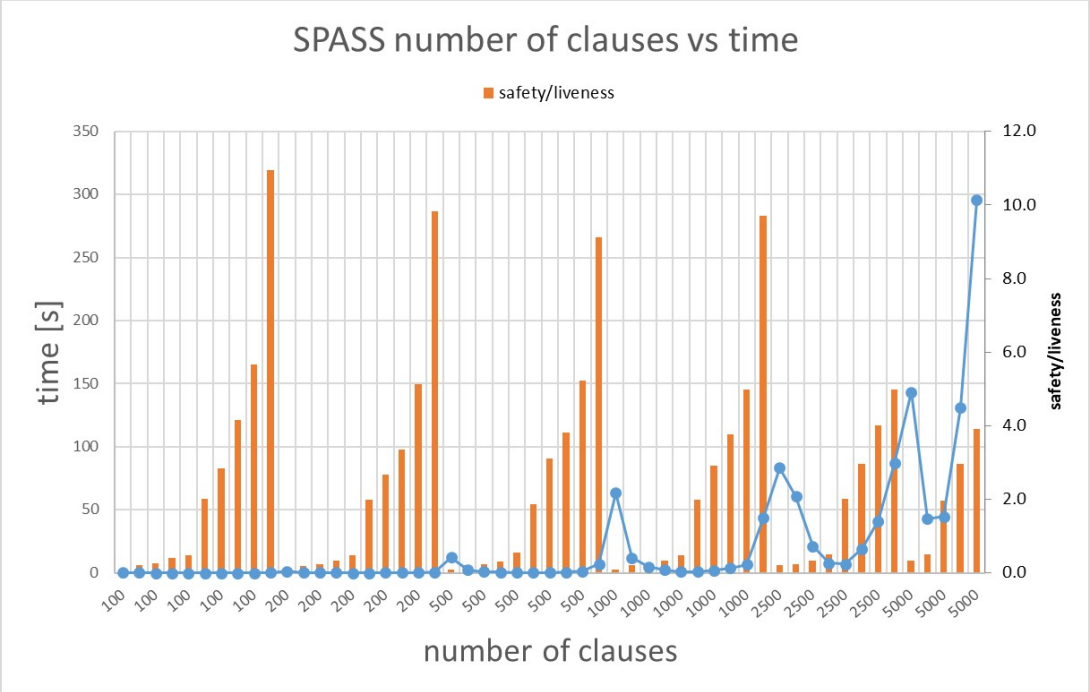
W zestawie 2 badano wpływ stosunku ilości atomów bezpieczeństwa do ilości atomów żywotności. W grupach test case'ów o tych samych ilościach klauzul zadano różne stosunki ilości atomów bezpieczeństwa do ilości atomów żywotności. Z otrzymanych wyników można wywnioskować, że zbyt nierówny stosunek tych liczb (odbiegający od 1 w obie strony) wpływa znacząco na czas wykonywania się benchmarku, wpływ ten jest większy dla Provera9 niż dla SPASS. Najlepiej widać tą zależność w Proverze9 dla test case'ów z 1000 klauzul. Tak samo jak dla zestawu 1 SPASS zużywa znacząco więcej pamięci niż Prover9 jednakże w zestawie 2 można zauważyć kilkukrotnie mniejszą ilość timeout'ów dla SPASS.



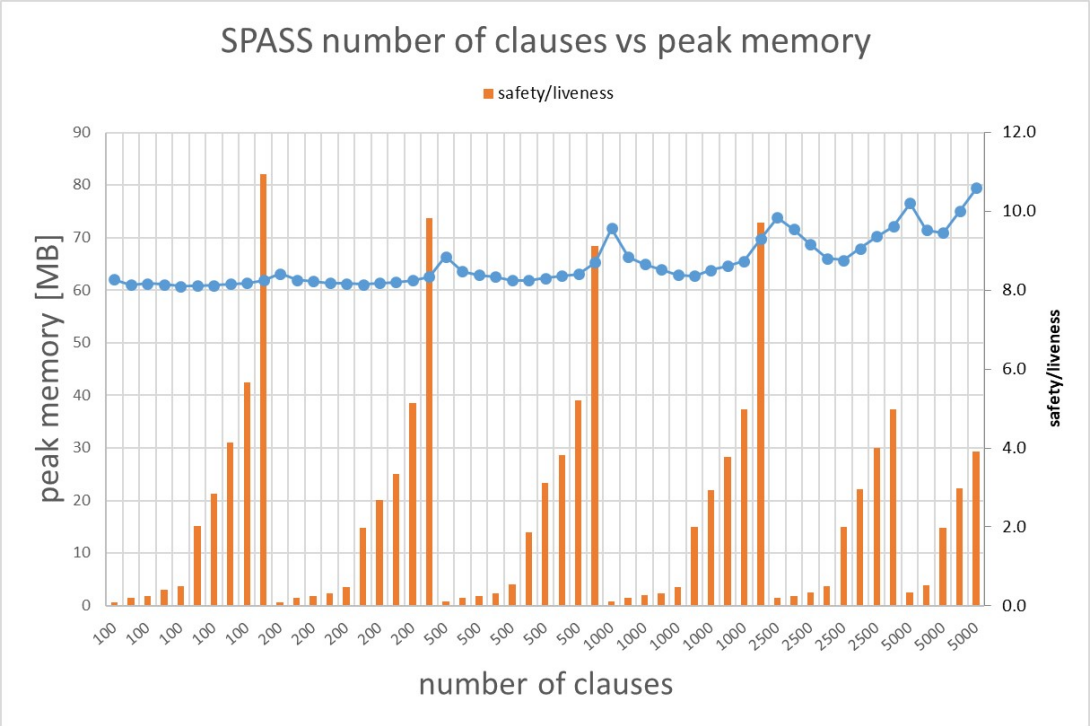
Rysunek 15: Prover9 zestaw 2 liczba klauzul vs czas



Rysunek 16: Prover9 zestaw 2 liczba klauzul vs pamięć



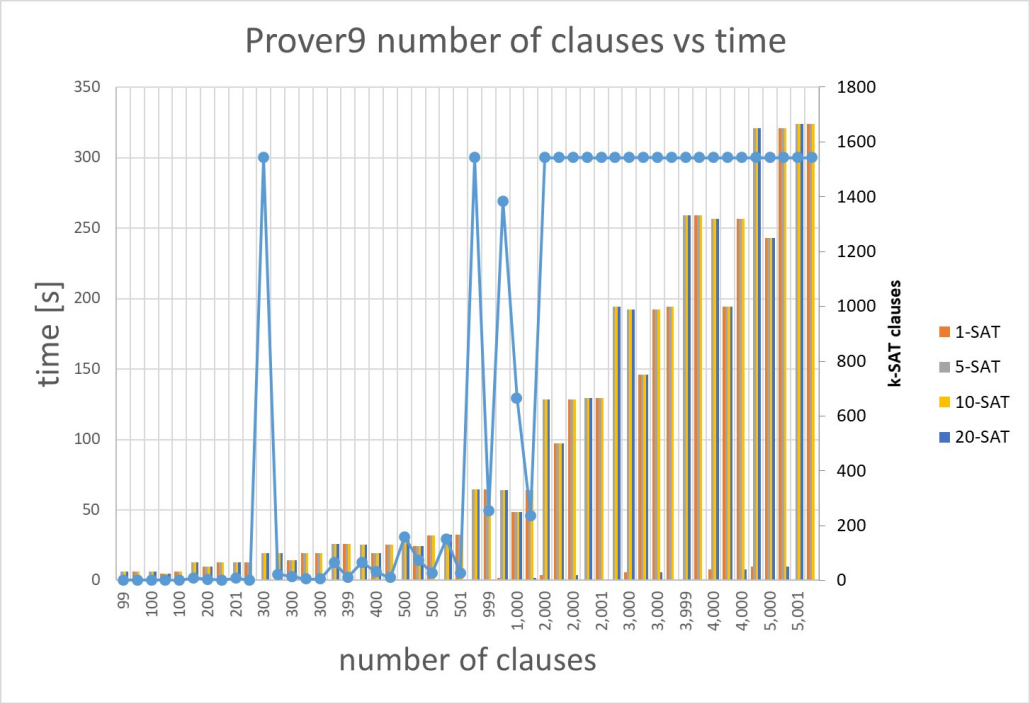
Rysunek 17: SPASS zestaw 2 liczba klauzul vs czas



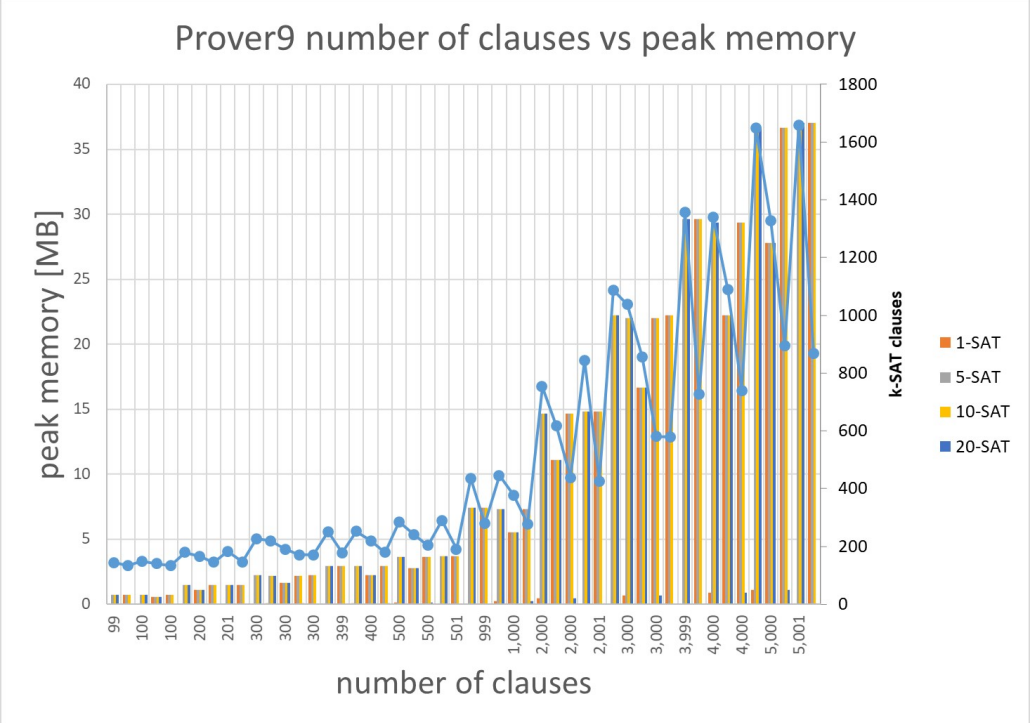
Rysunek 18: SPASS zestaw 2 liczba klauzul vs pamięć

7.3 Zestaw 3

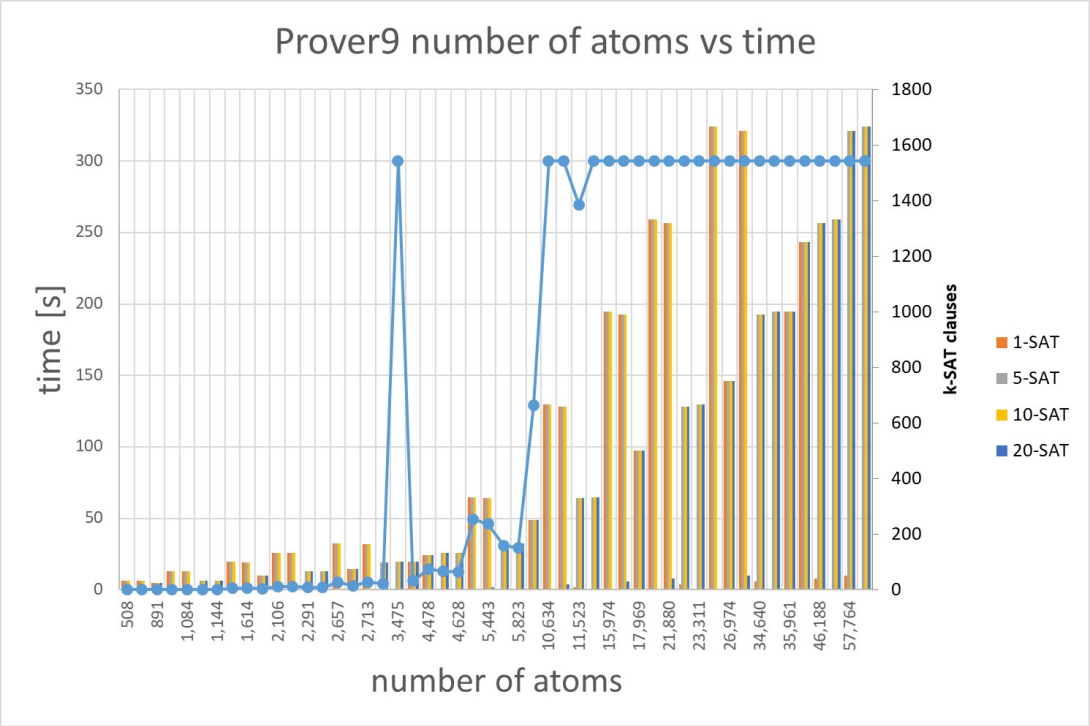
W zestawie 3 badano wpływ udziału klauzul k-SAT na czas wykonywania/pamięć w zależności od k. Zadano klauzule o różnym stopniu rozłożenia spośród wartości k ϵ {1,5,10,20}. Jak można było oczekiwać, test case'y o tej samej sumarycznej liczbie klauzul a o najmniejszym udziale klauzul z wysokim k miały najszybszy czas wykonywania oraz zużywały najmniej pamięci. Warto zwrócić uwagę na stopień tej zależności czyli zdecydowanę przewagę szybkości rozwiązań test case'ów o największym procentowym udziale klauzul 1-SAT, wraz ze wzrostem k czas rośnie lecz przyrost ten maleje co sugeruje zależność logarytmiczną czasu wykonania od k. Dla Provera9 najlepiej wpływ k widać na wykresie zależności liczby klauzul od pamięci (Rys 21.), charakterystyczne są grupy trzech test case'ów o tych samych liczbach klauzul a innym rozłożeniu k. Dla SPASS wyniki dają podobne wnioski, zauważalne jest kolejny raz wysokie zużycie pamięci i niewiele mniejsza liczba timeout'ów w porównaniu do Provera9.



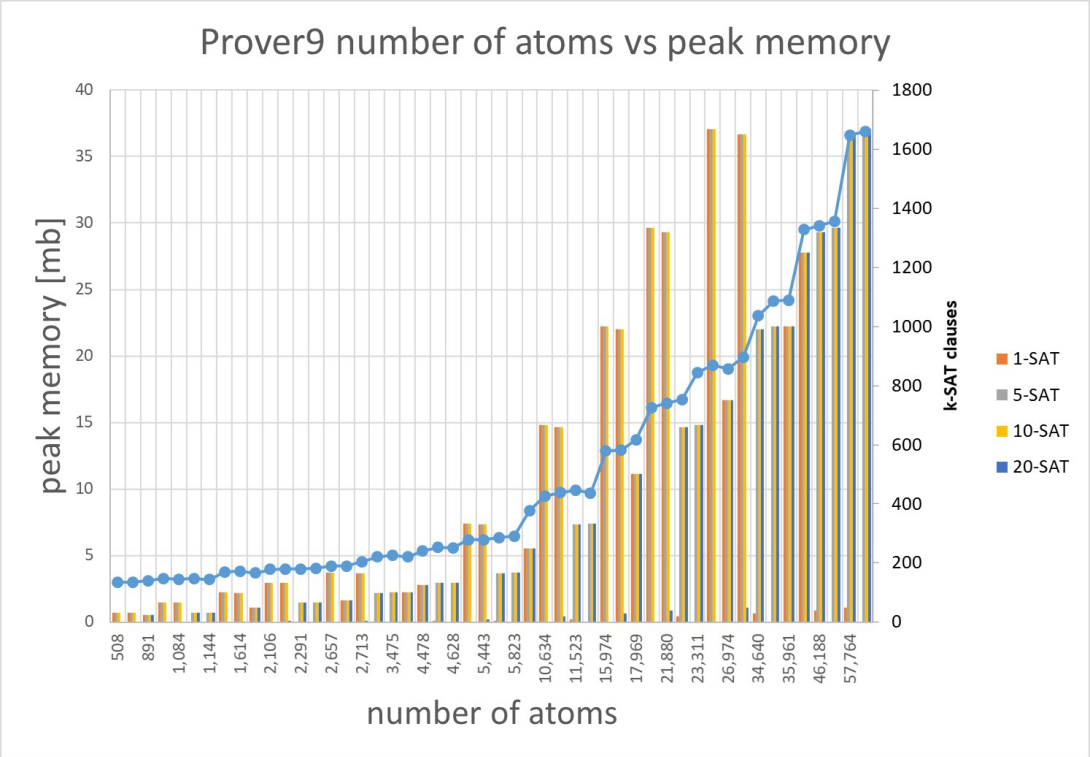
Rysunek 19: Prover9 zestaw 3 liczba klauzul vs czas



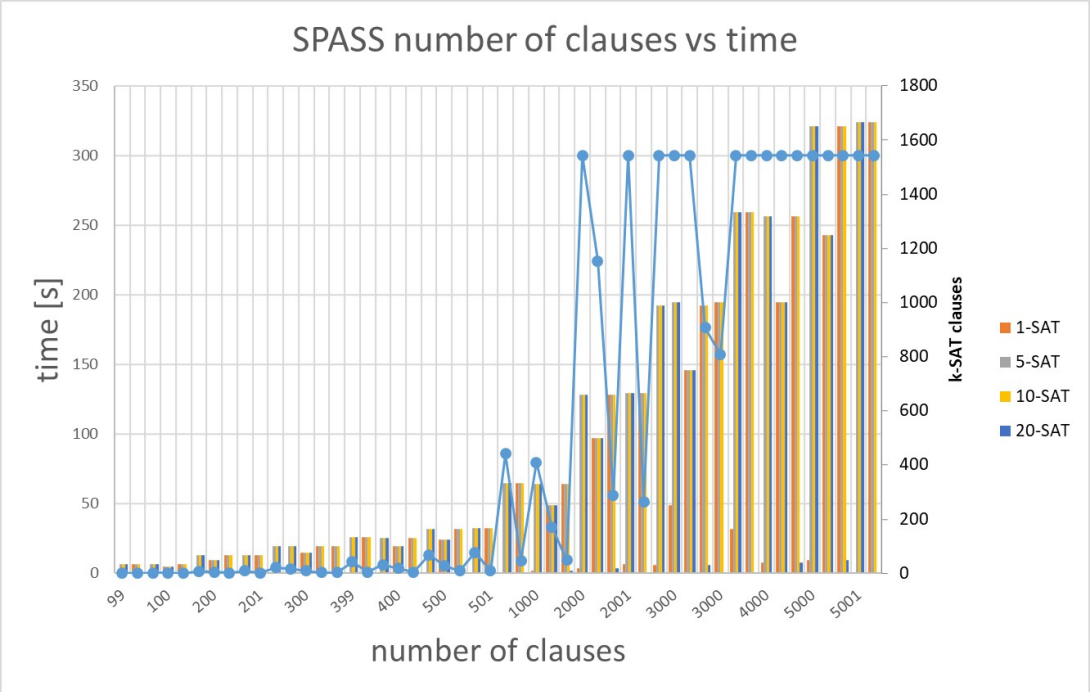
Rysunek 20: Prover9 zestaw 3 liczba klauzul vs pamięć



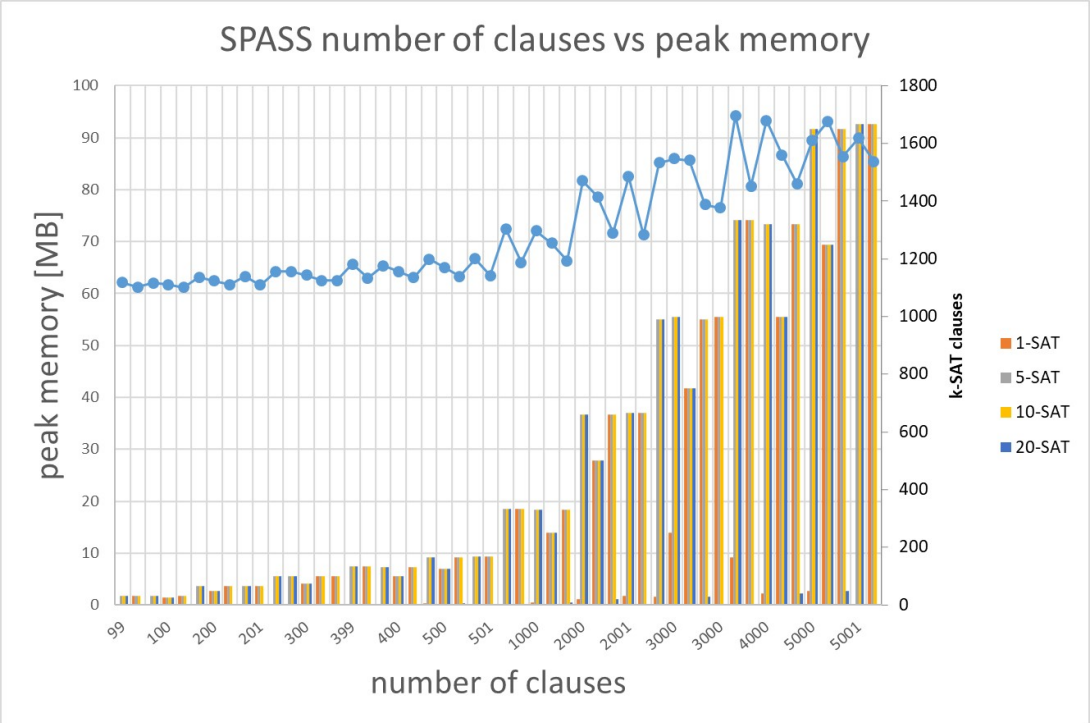
Rysunek 21: Prover9 zestaw 3 liczba atomów vs czas



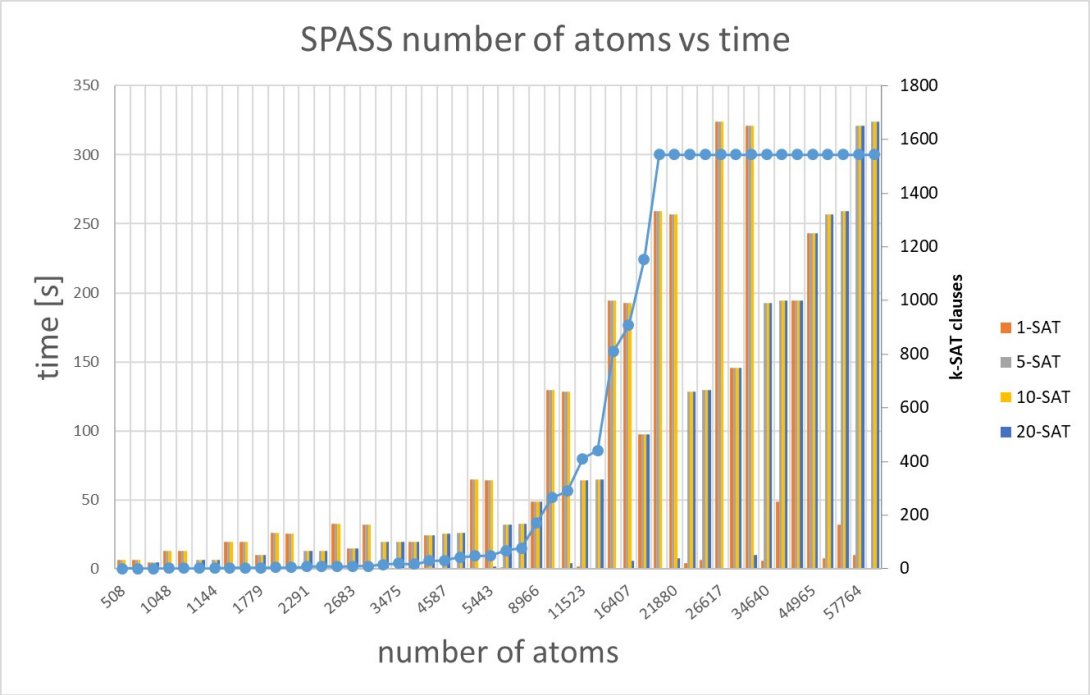
Rysunek 22: Prover9 zestaw 3 liczba atomów vs pamięć



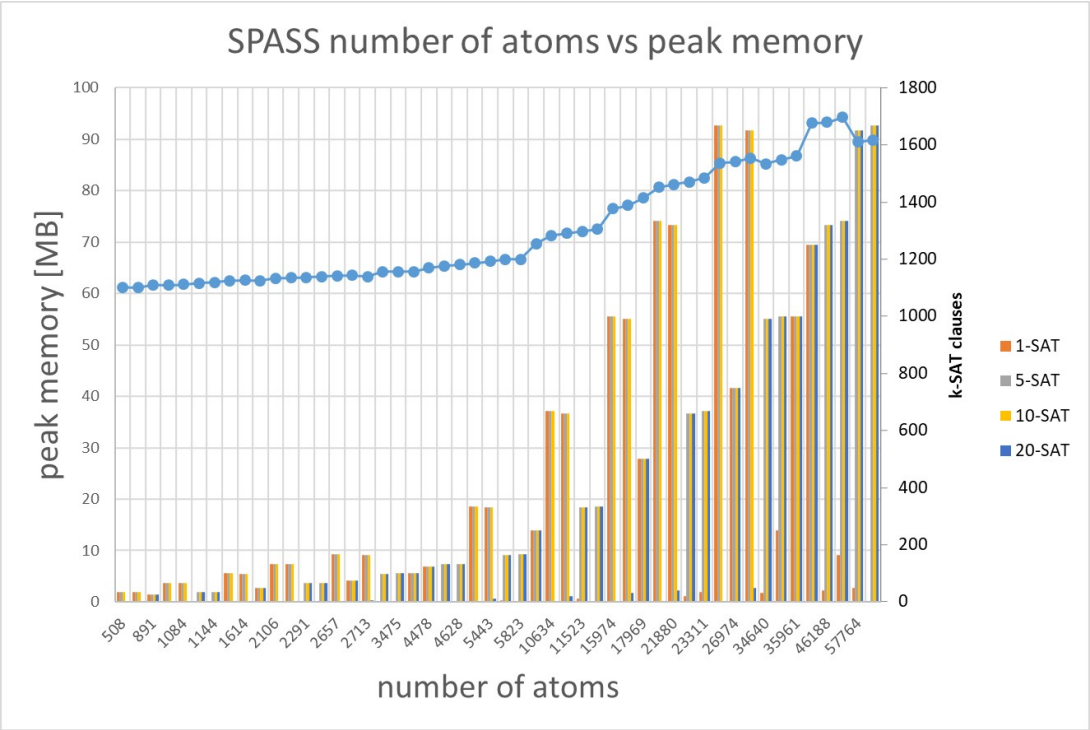
Rysunek 23: SPASS zestaw 3 liczba klauzul vs czas



Rysunek 24: SPASS zestaw 3 liczba klauzul vs pamięć



Rysunek 25: SPASS zestaw 3 liczba atomów vs czas



Rysunek 26: SPASS zestaw 3 liczba atomów vs czas

Skrót

ATP Automated Theorem Proving. ATP systems are enormously powerful computer programs, capable of solving immensely difficult problems, http://www.tptp.org/OverviewOfATP.html

BNF Backus normal form

CNF conjunctive normal form

DNF disjunctive normal form

FOF First Order Formula

FOL First Order Logic

ILP Integer Linear Programming

IP Integer Programming

LADR library written in c, shipped with Prover9. It also defines input syntax required by Prover9

LTL jedna z logit temporalnych

prover SAT solver

PRP Propositional Logic

SAT boolean satisfiability problem

SMT Satisfiability Modulo Theory

SMTLIB SMT-LIB is an international initiative aimed at facilitating research and development in Satisfiability Modulo Theories (SMT). Homepage http://smtlib.cs.uiowa.edu/index.shtml

Temporal Logic Temporal Logic

TFF Typed First-order Logic

THF Typed Higher-order Logic

TMTP Thousands of Models for Theorem Provers

TPI TPTP Process Instruction - source http://www.tptp.org/Seminars/TPI/Abstract.html

TPTP Thousands of Problems for Theorem Provers, official wesite http://www.tptp.org

TPTP2X The TPTP2X utility is a multi-functional utility for reformatting, transforming, and generating TPTP problem files.

TPTP4X The TPTP4X utility is a multi-functional utility for reformatting, transforming, and generating TPTP problem files. It is the successor to the TPTP2X utility, and offers some of the same functionality, and some extra functionality. TPTP4X is written in C, and is thus faster than TPTP2X.

TSTP Thousands of Solutions from Theorem Provers