AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Bachelor of Science Thesis

System losowego generowania formuł logicznych dla logiki pierwszego rzędu
System of random generation of logical formulas for first order logic

Author: *Mateusz Grzeliński*
Degree programme: *Computer Science*
Supervisor: *Radosław Klimek PhD, DSc*

Kraków, 2020

Contents

1. Introduction

SAT (boolean satisfiability problem) is the problem of determining if there exists an combination of variables (variable can be true or false) that satisfies a given boolean formula. For solving such problem dedicated programs are created, called SAT solvers (or provers).

Recently there has been substantial development in this area. Modern approach to SAT solving will likely use conflict-driven clause learning [1], various heuristics and make use of better and better hardware. This rapid development in theory of solving SAT problems is followed by number of implementations. Picking best (fastest) implementation of SAT solver for given input problem is not trivial as solving algorithms are based on similar algorithm. Moreover SAT problem has NP complexity what encourages optimizing solvers to specific cases - that is if input formula has frequent pattern, it would be beneficial to optimize solver around that pattern.

SAT problem can be represented in many logical systems and can contain various theories. To name a few possibilities, problem can be expressed with first order logic or propositional logic, additionally integer or floating point arithmetic may be used if needed. The use of appropriate theories must be carefully chosen by engineer when encoding a problem into logical formula as it will affect available range of SAT solver. Having that done, engineer must choose best solver for presented problem. This can be done by benchmarking chosen solver against set of formulas. Pre-generated input formulas can be often obtained from projects like TPTP (Thousands of Problems for Theorem Provers) (see section 2.3.3) but sometimes custom set of formulas would be handy (see chapter 3 for more detail on available datasets). Generator has this advantage over pre-generated dataset that it can precisely map encountered real-life problem into measurable, repeatable, custom dataset.

This thesis focuses on creating a tool that will assist generating custom set of formulas. The basic function of this tool is generating random set of formulas which can be used to test performance of provers. The goal is to create extendable, easy to use FOL (First Order Logic) formula generator. First order logic has been chosen deliberately. First order logic can be considered well balanced then it comes to expressiveness, complexity and adaptation among solvers. Less expressive and simpler logic would be propositional logic, whereas more complex

logic is for example higher order logic. FOL formulas will be encoded in TPTP format. It is important for the ease of use to keep inner representation of first order logic the same as mathematical definitions of elements of first order logic. Any arbitrary rule can be injected into generation to fulfill user specific needs. Those rules are meant to represent fragment or reality that engineer wants to represent in FOL. Formulas generated in this way can be used as performance testing tool as randomness of formulas can be finely controlled.

Secondary objective of the thesis is to acquaint reader with the topic of system properties and use of chosen provers: Z3 (section 4.4.2), Prover9 (section 5.4.1) and SPASS (section 5.4.2). Z3 solver was used internally by random formula generator. Procer9 and SPASS were used in chapter 5, which focuses on examining properties of systems in context of performance of provers in automated logical reasoning. Performance (time used to compute SAT and memory used) of Prover9 and SPASS were compared based custom generated dataset. Presented dataset represents mix of system properties encoded in first order logic.

2. Logical systems

Formulas must be presented in some kind of formal system. A formal system consists of a language over some alphabet of symbols together with (axioms and) inference rules that distinguish some of the strings in the language as theorems. These rules used to carry out the inference of theorems from axioms are known as the logical calculus of the formal system. A formal system may represent a well-defined system of abstract thought. A logical system is a formal system together with its semantics. Some recognizable logical formal systems are:

- propositional logic
- first order logic (FOL)
- higher order logic (HOL)
- temporal logic

The majority of formulas are probably going to be represented in propositional calculus because of simplicity of such notation. Propositional logic deals with variables which are connected with logical connectives. Variable can be true or false. Formula in propositional logic is easy to parse but may become quite long when compared to different formal logical systems. For more complex problem it may be easier to encode problem in more expressive system and the next natural step would be first order logic as it is not redefinition but extension of propositional logic.

2.1. First Order Logic

First order logic (also known as predicates logic, first-order predicate calculus) extends propositional logic by adding predicates, functors and quantifiers and non-logical objects. Definitions of FOL elements are contained below.

Variable unlike propositional logic, variables in FOL can stand for a relation (between terms) but which has not been specifically assigned any particular relation. Usually starts with

capital letter. If variables is used in quantifier it is called bound variable, otherwise it is called free variable.

Singleton variable is used only once in clause.

Functor is logical operator, that returns term. Functor has constant arity. Usually noted as $functor\_name/arity$, eg. $f/1$. Name of functor is usually lowercase f, g, h.

Constant functor is functor with arity 0.

Predicates is logical operator which returns true or false. Predicates operates on specific number of terms. This number is constant and called predicate **arity**. Usually noted as $predicate\_name/arity$, for example $p/1$. Name of predicate is usually lowercase p, q, r. Predicate describes relation between objects.

Term is variable, constant or result of functor.

Atom is logical statement, that can not be further divided. Atom consists of predicate, or in case of atom with equality it consists of terms.

Literal is atom or its negation.

Clause is disjunction of literals.

Unit clause is clause with only one literal.

Horn clause is clause, which contains at least one positive literal.

Quantifier specifies the quantity of specimens in the universe that satisfy an open formula (formula with at least one free variable). A formula beginning with a quantifier is called a quantified formula.

Existential quantifier is equivalent to a logical disjunction of propositions, this is, at least one proposition must be true.

Universal quantifier is equivalent to a logical conjunction of propositions, this is, all propositions must be true.

The following FOL example contains 1 existential quantifier, 2 variables W, Z, 1 predicate $p/2$, 2 constant functors $a/0$, $b/0$.

$$\exists_{W,Z} p(W, Z) \vee p(a, b) \tag{2.1}$$

2.2. Normal forms

Representing logic formula in one of normal forms sometimes can enable new or shorter reasoning not available otherwise. One of normal forms mentioned in this thesis is **conjunctive normal form (CNF)**. It is a conjunction of one or more clauses, where a clause is a disjunction of literals. Other example of normal form is DNF (Disjunctive Normal Form) where formula is

disjunction of conjunctions. Skolem normal form is often used in provers as it is a quantifier-free formula in conjunctive normal form.

The focus of this thesis is conjunctive normal form as it is one of the most basic forms that represent logical formulas and any statement in FOL can be converted to CNF. CNF is also a very comfortable tool for expressing system requirements and system properties. Statements connected with "and" statements are natural form of expressing requirements in natural laguage.

2.3. Formats used for representing formulas

There are many formats that allow representing logic, a few of them are described in this section. Many provers implement its own language for representing logic formulas like Prover9 what creates technical problems of comparability between solvers. In this thesis a standard called TPTP will be used to represent first order logic formulas.

To give a context just how complex issue of compatible formats can be, lets examine mentioned provers: Z3, Prover9 and SPASS (see table 2.1). Z3 has the best support for format called SMT-LIB. Tutorials on official Z3 website show examples in this syntax. Z3 also provides built in support for TPTP as input format, but can also display proof in DIMACS format in some situations (even thought DIMACS is used for propositional logic). Prover9 operates only on format provided by library called LADR - it is developed specifically for Prover9. SPASS has native supports for formats TPTP and a custom one, which based on yet another syntax.

Because of this complexity, many translators has been created. TPTP2X utility can convert TPTP syntax to 14 different and library LADR also has some tools dedicated to translation.

2.3.1. DIMACS

DIMACS[1] is also the name of one of popular formats for encoding propositional logic. Although variations of DIMACS can be found, it is mainly used to encode CNF what will be presented next. At the beginning there is always problem line: `p cnf VARIABLES CLAUSES`, where VARIABLES is number of variables in formula and CLAUSES is number of clauses in formula. c means line is a comment. Variables are noted with integers, negation is minus sign. Clause delimiter is `0`.

[1] *https://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf*

Prover / Formula syntax	Z3	Prover9	SPASS (prover)
SMT-LIB	supported as input format	not supported	not supported
TPTP	supported as input format	not supported	supported as input format
DIMACS	proof can be displayed in DIMACS	not supported	not supported
LADR	not supported	supported as input format	not supported
SPASS (syntax)	not supported	not supported	supported as input format

Table 2.1. Correlations between formula syntax and solver. Comparison includes only native support for syntax, without use of any translators.

```
p cnf 3 2
1 -3 0
2 3 -1 0
```

Listing 1. DIMACS formula example, formula consists of 2 clauses, 3 variables (1, 2, 3) and 5 literals)

2.3.2. SMT-LIB

SMT-LIB [2] is an international initiative aimed at facilitating research and development in SMT (Satisfiability Modulo Theory). Its 2 major projects handled by SMT-LIB are defining standard for encoding logical problems and creating benchmarks (more about benchmarks in section 3.1). Mentioned standard covers:

– a language for writing terms and formulas in a version of first order logic - it means that using SMT-LIB any subset of first order logic can be presented, including propositional logic

– a language for specifying background theories and defining a standard vocabulary of symbols for them - it means that theories like integers, floating points, real number etc. can be used. New theories can be easily defined.

– a language for specifying logics - it means that new (different than first order logic) logic systems can be created. For example first logic system can allow quantified formulas, second can be quantifier free.

– a command language for interacting with SMT solvers - it means in there are syntax elements which allow asserting and retracting formulas, querying about their satisfiability or their unsatisfiability proofs, and so on.

Compared to DIMACS, SMT-LIB is much more sophisticated, what can be advantage or disadvantage depending on use case. SMT-LIB standard is much more than simply format for encoding formulas.

```
; Basic Boolean example
(set-option :print-success false)
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; returns 'unsat'
(exit)
```

Listing 2. Example of quantifier free boolean formula with free (uninterpreted) sort and function symbols - QF_UF

2.3.3. Thousands of Problems for Theorem Provers

TPTP [3] - Thousands of Problems for Theorem Provers - is both problem library used for testing ATP (Automated Theorem Proving) systems and standard describing syntax for those tests. TPTP defines syntax for several logic systems: TPI (TPTP Process Instruction), THF (Typed Higher-order Logic), TFF (Typed First-order Logic), FOF (First Order Formula), CNF (Conjunctive Normal Form) but in this thesis only FOF and CNF will be used.

Compared to DIMACS and SMT-LIB, TPTP is middle ground when it comes to complexity. Unlike DIMACS it supports multiple languages, but unlike SMT-LIB, new logics can not be defined and it has no language for interacting with SMT solvers. On the other hand it has set of tools that allow for manipulation of formulas described in section 2.3.3.2.

2.3.3.1. First order logic in Thousands of Problems for Theorem Provers syntax

The following description is a description of FOL syntax elements used in TPTP [2].

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 `fof(name,role,formula,source,[useful_info])` or `cnf(name,role,formula,source,[useful_info])`

- `name` is only for documenting purposes
- `role` gives the user semantics of the formula, in context of SAT problem it will be usually `axiom`, but can also for example `hypothesis` or `definition`. They are accepted, without proof, as a basis for proving conjectures. In CNF problems the axiom-like formulae are accepted as part of the set whose satisfiability has to be established. A problem is solved only when all formulas with role `conjecture` 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.
- `formula` is logical formula
- 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 `useful_info` field of an annotated formula is optional, and if it is not used then the `source` field becomes optional

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 like `$true` and `$false`, `=` and `!=` and arithmetic predicates
- system predicates and functors, whose interpretation is ATP system specific, like `$o` - the Boolean type, `$i` - the type of individuals, `$real` - the type of reals, `$rat` - the type of rational, and `$int` - the type of integers.

The universal quantifier is `!` (example of usage is shown in listing 4), the existential quantifier is `?` (example shown in listing 3), and the lambda binder is `^`. Quantified formulae are written in the form `Quantifier [Variables] : Formula`

[2] Full technical document for TPTP syntax can be found at *http://www.tptp.org/TPTP/TR/TPTPTR.shtml*

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

Listing 3. TPTP FOL formula with existential quantifier

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

Listing 4. TPTP FOL formula with universal quantifier

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

2.3.3.2. Additional tools in TPTP library

TPTP ships with TPTP4X (written in c), TPTP2X (written in prolog) utilities, which are used for reformatting, transforming, and generating TPTP problem files. Example of functionalities:

- converting FOF to CNF (for example with otter [4], bundy [5] algorithm, details in [6]) as presented in 5
- converting TPTP to syntax required by prover9, dimacs, otter, dfg and more
- optimization FOF, CNF with different algorithms
- change order of CNF

```
% for every X, Y operation lesseq is the same as less or equal
fof(this_is_obvious, axiom,
  ! [X,Y] : ( $lesseq(X,Y) <=> ( $less(X,Y) | X = Y ) )
  ).
% equivalent in CNF, 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) )).
```

Listing 5. TPTP FOL formula, translated to CNF

3. Overview of existing benchmark solutions

There are number projects that provide input formulas for solvers, in order to test their performance. The majority of them provide pregereneted files grouped in categories, some of them hosts websites with problem (and optionally solutions) database and very few of them generate input formulas on demand. A selection of different benchmark solutions is described in this chapter.

3.1. SMT-LIB

As described in section 2.3.2, SMT-LIB is both standard for encoding formulas and database of benchmarks. There are 2 main categories of benchmarks: incremental and non-incremental. This enables testing solvers in different scenarios. Next benchmarks are divided into different logics (sublogics) what in the end gives several dozen of different categories. Sublogics are logics that allow only certain theories. For example one sublogic is

– logic with closed linear formulas over linear integer arithmetic (SMT-LIB calls it LIA)

– logic with closed linear formulas in linear real arithmetic (LRA)

– logic with quantifier-free integer arithmetic (QF_NIA) and more

3.2. SATLIB

SATLIB [7] is a collection of benchmark problems, solvers, and tools used for SAT related research. Its collection of pregereneted benchmarks is provided in propositional logic (DIMACS format). Benchmarks are grouped into categories, to name a few:

– uniform random-3-SAT

– random-3-SAT instances and backbone-minimal sub-instances

– random-3-SAT instances with controlled backbone size

– "flat" graph colouring

– "morphed" graph colouring and more

3.3. Tough SAT

Tough SAT [8] is an online platform that can generate boolean CNF formulas that encode predefined problems. Problems are represented in propositional logic (DIMACS format). Although Tough SAT is in fact generator it supports only several predefined problems. Mentioned problems are:

– factoring – generate a CNF formula whose satisfying assignment encodes two non-trivial factors of the product $Factor1 * Factor2$ (if there exist any). $Factor1$ and $Factor2$ is provided by user.

– random subset sum – generate a random subset sum instance (the ground set is of size Set Size and whose elements are random positive integers less than Max Number, and the target number is the sum of a random subset of the ground set), and encode the instance as a boolean CNF formula

– manual subset sum – variation of above

– random – k-SAT generate a random k-SAT boolean formula with the specified number of variables and clauses

– cocktail SAT – mix of all of above

3.4. CNFgen

CNFGen [9] is a python script that produces combinatorial benchmarks in propositional logic (DIMACS format). These benchmarks come mostly from research in proof complexity. CNFGen can generate random formulas but also number of predefined problems:

– pigeonhole principle problem – the formula states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item

– parity principle problem – the formula claims that N elements can be matched in pairs

– k-clique problem – the formula claims that there is no clique of size at least k in the input graph G. and more

3.5. Thousands of Problems for Theorem Provers

As described in section 2.3.3, TPTP is both standard and benchmark database. It contains library of problems encoded in TPTP format. Formulas are classified into different domains (groups): LCL - Logic Calculi, COL - Combinatory Logic and more. These formulas can be queried to filter for example formulas from different domains with same number of atoms.

Next to TPTP library there is TSTP (Thousands of Solutions from Theorem Provers) - library of solutions produced by various solvers based on mentioned problem database. TSTP additionally contains tools for examining and manipulating the solutions.

In the end in TPTP there is also TMTP (Thousands of Models for Theorem Provers) - a library of models of axiomatizations for ATP systems. Together TPTP, TSTP and TMTP create set of tools for benchmarking, validating and comparing solvers.

3.6. The need of new solution

Although a lot of ready to use benchmarks can be found, there are not a lot of generators. Tough SAT and CNFgen are generators that generate mainly predefined problem with given size, but they operate only in propositional logic. SMT-LIB, SATLIB, TPTP are repositories of benchmarks, not generators itself. Out of mentioned solutions only SMT-LIB and TSTP support first order logic. None of mentioned solutions can generate first order logic formulas. Out of mentioned projects, only CNFGen can produce random formulas.

None of mentioned existing solutions are able to produce random first order logic formulas thus a new solution will be presented in following chapter.

4. Architecture of FOL formula generator

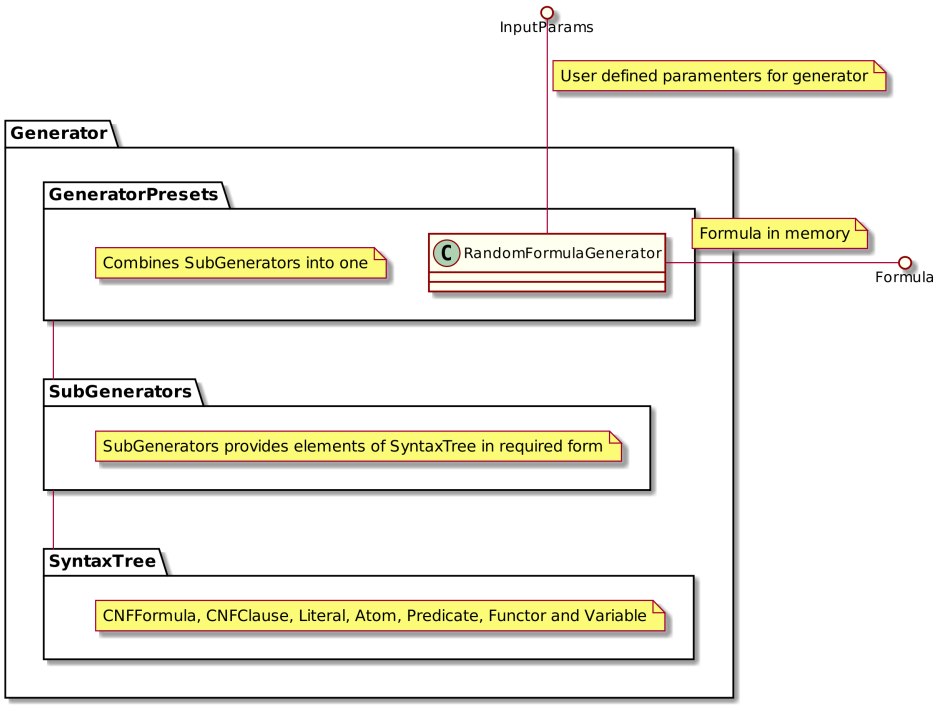
There are 2 parts that together create generator. The main part is $Generator$ (picture 4.1a) which is responsible for creating random formula itself and $ExportUtils$ (picture 4.1b) is complementary package responsible encoding formula to TPTP format and counting statistics.

The base of $Generator$ is $SyntaxTree$ (described in section 4.2) - it defines elements elements of FOL in program. Every element defined in $SyntaxTree$ is has at least one corresponding $SubGenerator$ (see section 4.3), and multiple $SubGenerators$ create $GeneratorPreset$. $GeneratorPreset$ is the element which is going to ensure that generated formula is random within limits. This element is also interface for user (see section 4.4).

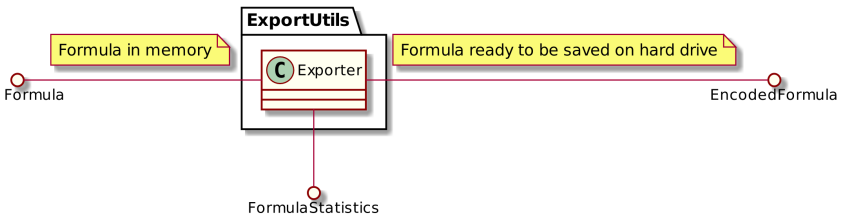
4.1. Generator input parameters

$Generator$ takes as input user provided parameters and outputs random formula. User provided parameters have 2 flavours. The first set of parameters are not processed in any way, the set "background" for further randomizing formula. These include:

- set of variable names $\{'v1', 'v2', \dots\}$
- set of functor names $\{'f1', 'f2', \dots\}$
- set of predicate names $\{'p1', 'p2', \dots\}$
- set of allowed functor arities $a_f = \{0, 1, 2, \dots\}$
- maximum recursion depth n for functors
- set of allowed predicate arities $a_p = \{0, 1, 2, \dots\}$
- set of atom allowed connectives, that is no connective or/and any subset of $AllowedConnectives = \{=, != , \emptyset\}$
- set of allowed clause lengths $ClauseLengths = \{1, 2, \dots\}$



(a) Overview of formula generation process



(b) Overview of formula exporting process

Picture 4.1. Overview of generator architecture

– amount of literals to be negated

Second group of parameters is the main point of interest - these parameters are provided as range of possible values, that user wants to achieve in generated formula:

– formula contains from c_{min} to c_{max} clauses

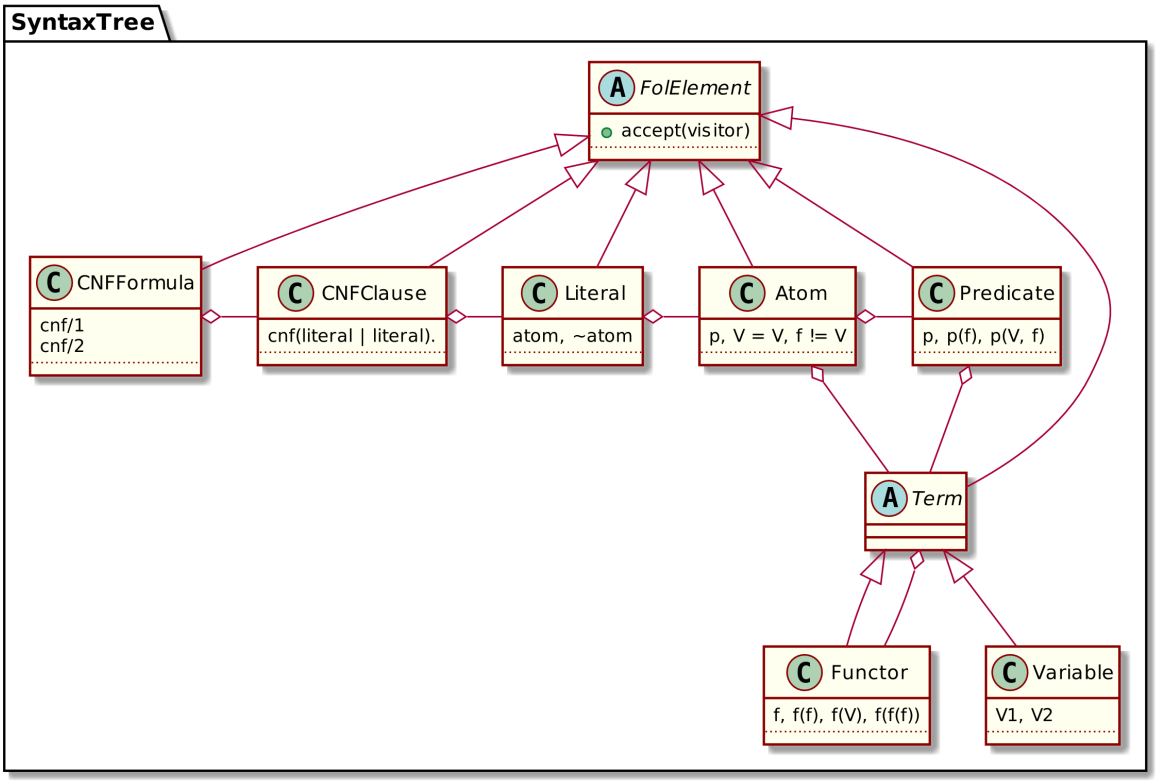
– formula contains from l_{min} to l_{max} literals

4.2. Implementation of syntax tree of first order logic

Abstract syntax tree of FOL is implemented in the same way as mathematical definition, see picture 4.2. All classes have base class $FolElement$ so that they can be easily identified as

element of first order logic and introduce visitor pattern. Visitor pattern is used for exporting formula from intermediate representation to the format of choice (like TPTP) as well as counting statistics about generated formula. See for example listing 8 - it is convention from TPTP to store some statistics about file at the beginning of file as comment.

Variable and functor are terms, that is why the inherit from term class. Functor is recursive structure it can contain variable or another term, that is why it is connected with aggregation with term class. Predicate can contain only terms, atom can contain term or predicate, literal can contain only one atom but adds sign to it, clause is one or more literal, CNF formula is one or more clause.



Picture 4.2. Class diagram for internal representation of first order logic elements

4.3. Implementation of SubGenerators

Generators are meant to be used as cascade (picture 4.3), that is formula generator which provides formulas requires subgenerator which provides clauses, which requires subgenerator which provides literals and so on. This concept ensures that one class has one responsibility.

Note that all parameters from described as "background" are passed to subgenerators. Function $generate()$ returns randomized element of syntax tree.

4.4. Implementation of generator presets

Subgenerators provide very fine control over generated created elements. To automate process of creating set of subgenerators a $GeneratorPreset$ can be defined. One of presets is $RandomCNFFormulaGenerator$ (picture 4.4). This is interface for user to create random CNF formulas, it accepts all parameters defined in section 4.1.

After passing arguments to appropriate subgenerators, there are still 2 left: number of clauses and number of literals (provided as ranges). They require special care as the number of clauses affects the number of literals. To generate random formula within these constrains, number of clauses with appropriate clause length must be computed.

4.4.1. Randomizing formula within limits

In order to compute number of clauses with appropriate clause length, CNF formula must be formalized to create relationship between $NumberOfClauses$ and $NumberOfLiterals$. First, define CNF formula F_{cnf} as set of unordered clauses $c_1, c_2, \ldots, c_n$. Computing number of clauses with this definition is trivial, but computing number of literals is not.

$$F_{cnf} = \{c_1, c_2, \ldots\ c_n\}$$
$$NumberOfClauses(F_{cnf}) = |\{c_1, c_2, \ldots\ c_n\}| = n$$
where n – number of clauses in formula

In next step, we will group clauses by their length. Let $c^m = \{c1, c2, ..\}$ be set of clauses with common length m. Allowed clause lengths $ClauseLengths = \{1, 2, ..\}$ are known in advance as they were passed in input parameters. Now formula becomes sum of set of clauses grouped by length, $NumberOfClauses$ is sum of cardinality of each of clause set and $NumberOfLiterals$ can be computed similarly to $NumberOfClauses$, but number of clauses with defined length must be multiplied by its length m.

$$F_{cnf} = \bigcup_{m \in ClauseLengths} c^m$$
$$NumberOfClauses(F_{cnf}) = \sum_{m \in ClauseLengths} |c^m|$$
$$NumberOfLiterals(F_{cnf}) = \sum_{m \in ClauseLengths} m|c^m|$$

The unknown variables are $|c^m|$ - the number of clauses with length m. After adding constrains defined by user in input parameters:

$$NumberOfClauses(F_{cnf}) = \sum_{m \in ClauseLengths} |c^m| \tag{4.1}$$
$$NumberOfLiterals(F_{cnf}) = \sum_{m \in ClauseLengths} m|c^m| \tag{4.2}$$
$$l_{min} < NumberOfLiterals(F_{cnf}) < l_{max} \tag{4.3}$$
$$c_{min} < NumberOfClauses(F_{cnf}) < c_{max} \tag{4.4}$$

4.4.2. Using Z3 to resolve user constraints

Z3 [10] is a SMT prover from Microsoft Research. It uses SMT-LIB as input format, but also has bindings to multiple languages. It can be used to solve user constraints presented in 4.4.1. Z3 supports integer arithmetic what is essential for solving user constraints.

Solving equation is implemented as class $CNFConstraintSolver$ (picture 4.4). Parameter $ClauseLengths$ corresponds to equation 4.2, parameter $NumberOfLiterals$ corresponds to range of allowed values of literals - tuple of minimal and maximum range - (equation 4.3), parameter $NumberOfClauses$ corresponds number of allowed values of clause - tuple of minimal and maximum range - (equation 4.4). The set of variables x_i will be returned from method $solve()$ as dictionary, where key is m (clause length) and value is how many clauses are needed.

Function `_solve()` in listing 6 solves user constraints using Z3 solver bindings in Python. Line `X = [z3.Int() ...]` defines Z3 variables, which can contain only integer values. This variable is used to represent x_i from 4.1. Further down the line `s = z3.Solver()` creates model, to which constraints will be added. Constraints are defined by functions `z3.Sum`, `z3.And`, and can be added to model via `s.add`. Variables must follow 2 rules: they must be non negative and be in range defined by 4.4 and 4.3. Solution can be calculated from model with 2 operations: first check if model is satisfayable, next evalueate (get value) of variables in model. If another

solution is needed, it can be done by adding constraints, that forbids concurrencyent solution and recalculating model.

```
    def _solve(self) -> Iterable[Dict[int, int]]:
        """Solve in deterministic order"""
        A = self.literal_coefficients
        n = len(A)
        X = [z3.Int() for i in range(n)]
        s = z3.Solver()

        # solutions must be positive
        s.add(z3.And([X[i] >= 0 for i in range(n)]))

        # clauses must be in range
        s.add(z3.Sum(X) <= self.number_of_clauses[1])
        s.add(z3.Sum(X) >= self.number_of_clauses[0])

        # literals must be in range
        s.add(z3.Sum([A[j] * X[j] for j in range(n)]) <= self.number_of_literals[1])
        s.add(z3.Sum([A[j] * X[j] for j in range(n)]) >= self.number_of_literals[0])

        while s.check() == z3.sat:
            solution = [s.model().evaluate(X[i]) for i in range(n)]
            yield {clause_len: s.as_long() for clause_len, s in zip(A, solution)}
            forbid = z3.Or([X[i] != solution[i] for i in range(n)])
            s.add(forbid)
```

Listing 6. Lazy, deterministic function for solving user constraints

Fundamental problem of using SAT (or SMT) solver in this scenario when any, random solution is needed, is fact that solver will always produce deterministic result for the same input data. There are 2 solutions for this problem. First one, presented here, is using a wrapper that will temporarily skip or discard part of solutions, to randomize deterministic output. In this cave, non public function $\_solve$ in listing 6 is a function that actually yields solutions and $solve$ in listing 7 is a randomizing wrapper. Randomization occurs at the cost of time and memory. Second approach assumes that solver supports soft clause[1]. Using soft clauses a random starting point can be suggested to a solver. For example line added to listing 6 `s.add(z3.Sum([A[j] * X[j] for j in range(n)]) == random_number))` would hint Z3 solver to strat in `random_number` if Z3 supported this feature in Python bindings.

[1] contrary to hard clauses, soft clauses does not have to be satisfied

```
    def solve(self, skip_chance: float = None):
        """Solve in random order"""
        skip_chance = random.random() if skip_chance is None else skip_chance
        cache = []
        for solution in self.solve():
            if random.random() < skip_chance:
                yield solution
            else:
                cache.append(solution)

        random.shuffle(cache)
        for cached_solution in cache:
            yield cached_solution
```

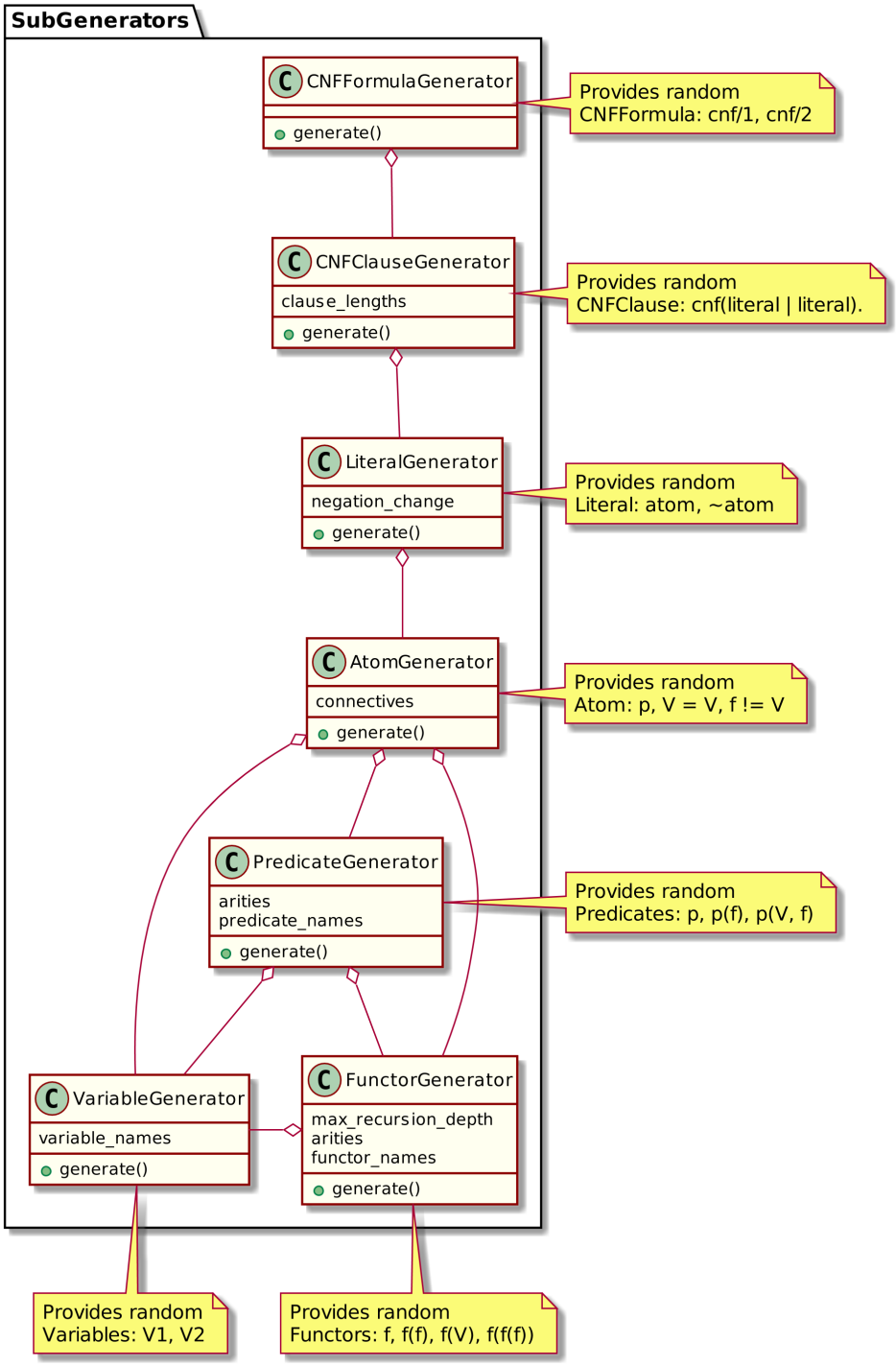
Listing 7. Lazy, randomizing wrapper around deterministic solver 6

4.5. Export and generate statistics about formula

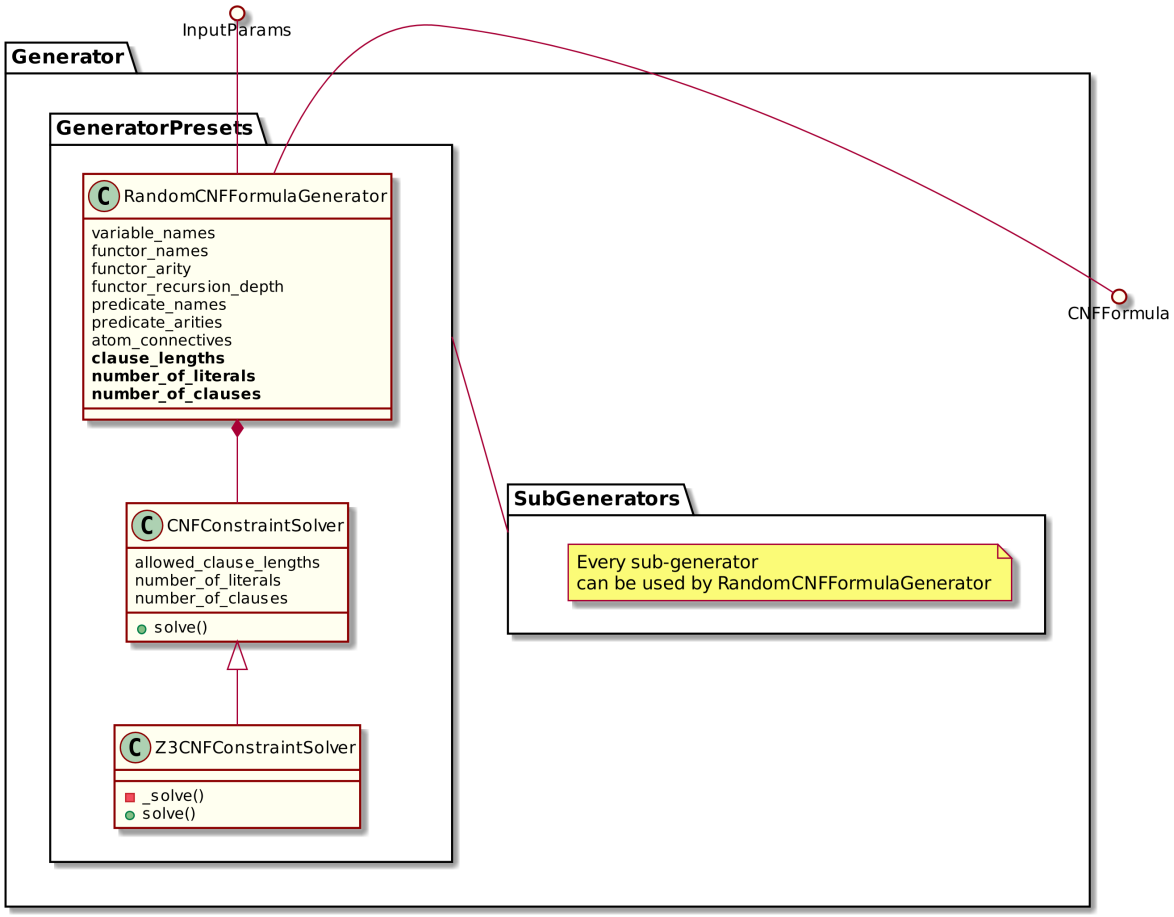
After generating formula it needs to be encoded and optionally statistics can be generate. Those two things are done with visitor pattern. As input $Exporter$ receives only $CNFFormula$, as shown in picture 4.5. First statistics are generated - $CNFFormula$ is passed to $CNFFormulaVisitor$ the result is provided as $CNFFormulaInfo$. Then $CNFFormula$ and $CNFFormulaInfo$ is handled by $TPTPExporter$ where again it is used with visitor pattern to encode formula in TPTP format. Additionally $CNFFormulaInfo$ is used to create unique to TPTP header (see for example listing 8).

```
% ------------------------------------------------------------------------------
% File     : 0.p
% Syntax   : Number of clauses     :   95 ( 95 non-Horn;   0 unit;   - RR)
%            Number of atoms       :  950 (  0 equality)
%            Maximal clause size   :   10 ( 10 average)
%            Number of predicates  :   20 (206 propositional; 0-4 arity)
%            Number of functors    :   20 (940 constant;   0 arity)
%            Number of variables   :  943 (327 singleton)
%            Maximal term depth    :    0 (  - average)
%
% ------------------------------------------------------------------------------
cnf(name,axiom,p4(V6)|p10(V1, V4, V9)|~p7|p17|p1(V4, V3, V9)|p7|p17|p1(f11, f11,
↪  f5)|p7|~p0(f2)).
cnf(name,axiom,p2(V7, V9)|p4(f19)|p2(f8, f5)|p10(f7, f8, f8)|p12|p6(V2, V6)|p14(f8,
↪  f16, f9, f16)|p3(f14, f3, f14, f18)|p11(V3, V9)|p12).
...
```

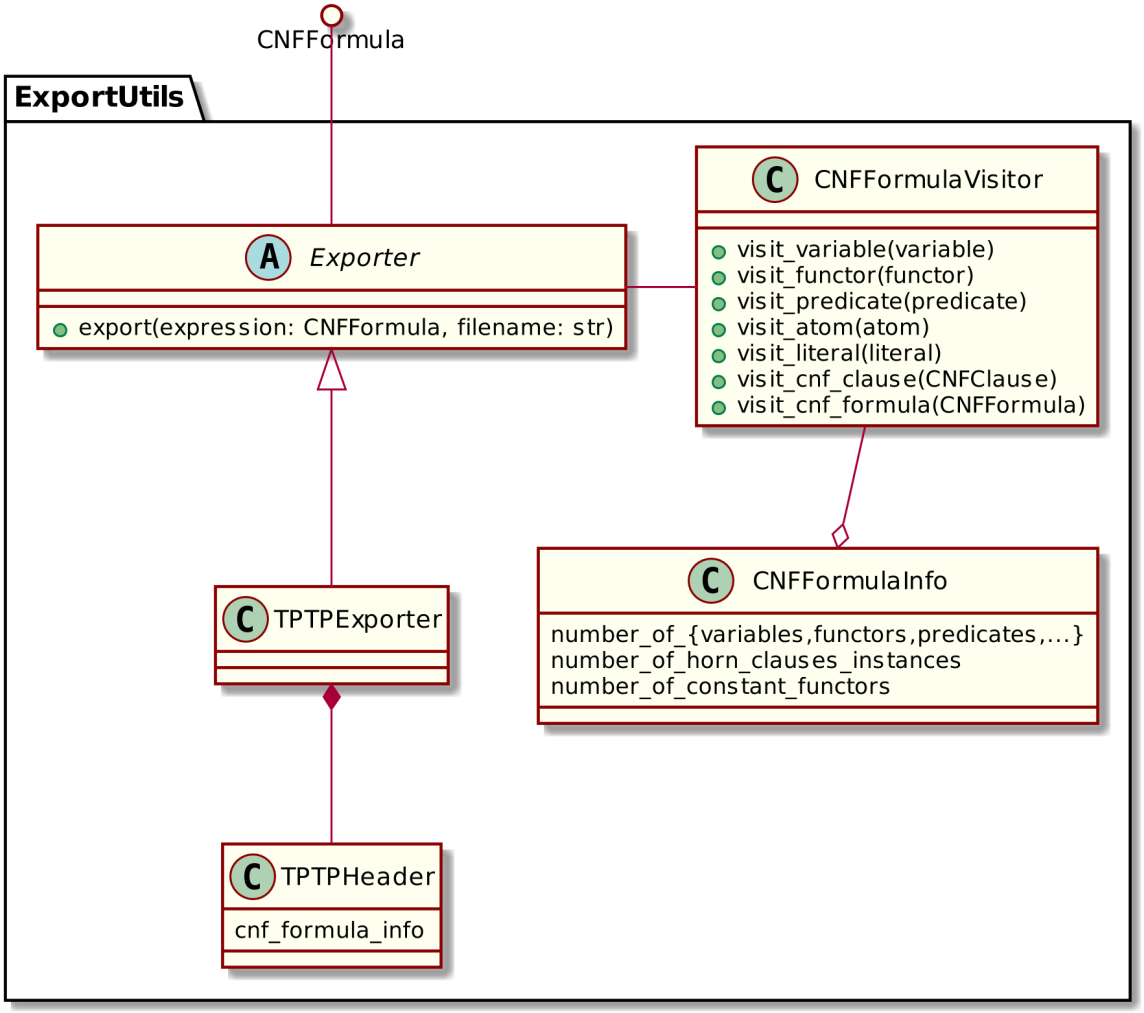
Listing 8. Example of generated formula (limited)



Picture 4.3. Class diagram of SubGenerators in CNF formula generator



Picture 4.4. Class diagram of CNF formula generator



Picture 4.5. Classes for which take part in generating statistics and exporting FOL CNF formula

5. Generating system properties

One of the uses of random FOL can be to generate random formula that represent system properties. That formula can be used as input for benchmark.

5.1. Properties of computer systems

System properties were first discussed in context of concurrency [11] as a tool for formal verification multiprocess programs. One of first proposed properties were safety and liveness. These properties can apply to computer systems in general and be expressed in different formal systems.

Liveness [12] is system property, that states, that something good will eventually happen. Liveness formula guarantees that there is at least one case, where formula evaluates to true.

Safety [12] is system property, that states, that something bad will never happens. Safety formula must always be satisfied.

An informal example of system with liveness and safety properties could be an elevator. Statement: "Elevator will eventually stop" is liveness property of elevator as may be running now but eventually will hit the ceiling or floor. Statement: "Elevator must never run when the door is open" is safety property - the statement is always true.

5.2. Properties of computer systems in first order logic

In FOL liveness and safety can be expressed as quantifiers, safety as universal quantifier and liveness as existential quantifier. If we were to use previously mentioned elevator example, the statements can be written as follows:

– "Elevator will eventually stop" can be converted to logic formula: $\exists_t e(t)$ where predicate e means "elevator is still", assuming $t < \infty$. The formula reads: there exists moment t that the elevator is not moving

– "Elevator must never run when the door is open" can be converted to logic formula: $\forall_t \neg e(t) \wedge d(t)$ where predicate e means "elevator is still" and d means "doors are open". The formula reads: for every moment t elevator is running and the doors are shut (at the same time).

5.2.1. Properties of computer systems in first order logic in quantifier free

Every FOL can be converted to CNF, so system properties can be also represented in CNF but quantifiers must be also converted to CNF. The process of removing all the existential quantifiers from a formula is known as skolemization. The result is a formula in skolem normal form that is equivalent in computational complexity to the original. Skolemization follows several rules:

– Variables bound by existential quantifiers which are not inside the scope of universal quantifiers can simply be replaced by constants - $\exists_t e(t)$ can be replaced by $e(f)$ where f is new constant (functor)

– When the existential quantifier is inside a universal quantifier, the bound variable must be replaced by a Skolem function of the variables bound by universal quantifiers - $\forall_x e(x) \wedge \exists_y e(t)$ can be replaced as $\forall_x e(x) \wedge e(f(x))$

Elevator example converted to quantifier free form looks as follows[1]::

– "Elevator will eventually stop" $\exists_t e(t)$ in quantifier free form: $e(f)$ - new constant functor f replaced variable t

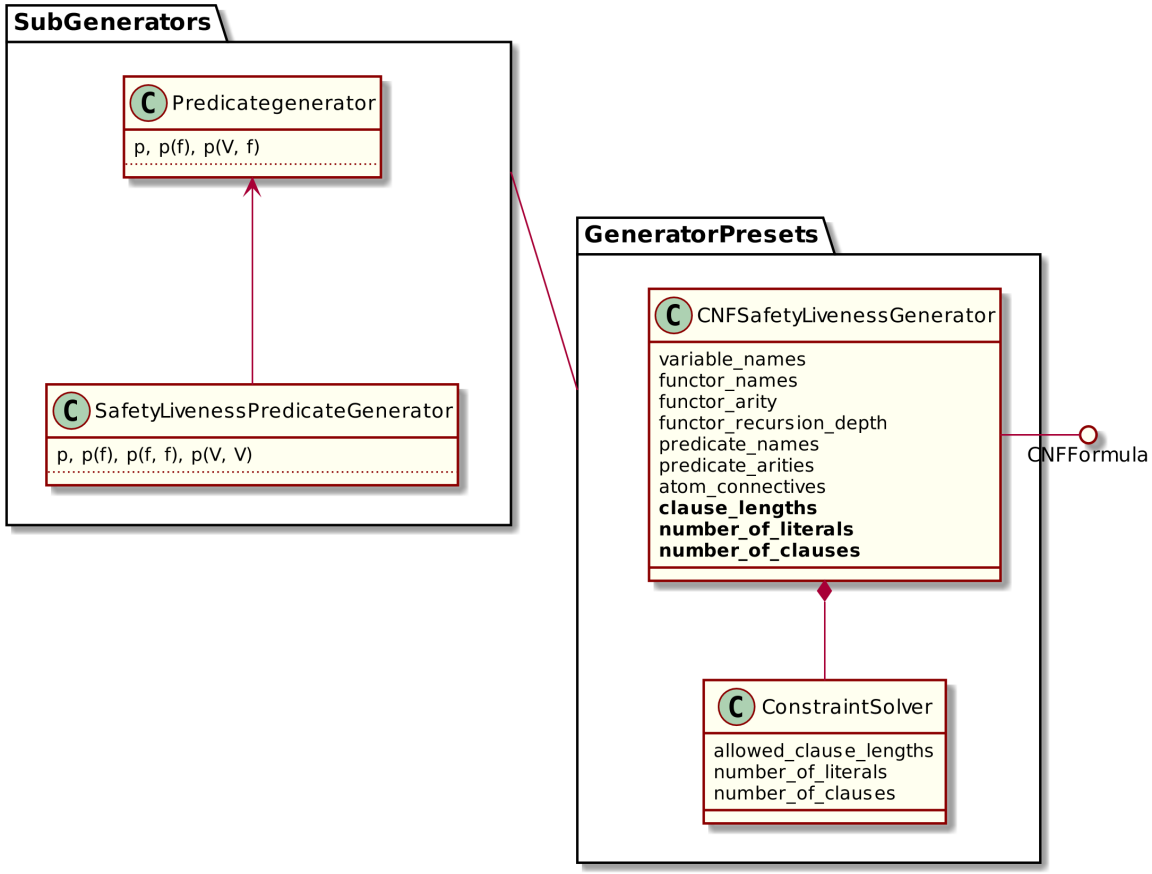
– "Elevator must never run when the door is open" $\forall_t \neg e(t) \wedge d(t)$ in quantifier free form: $\neg e(X) \wedge d(Y)$ - 2 new variables X and Y replaced variable t bounded to universal quantifier

In conclusion safety and liveness can be represented in CNF as 2 form of predicates: liveness as predicate, where arguments are only constant functors, safety as predicate where arguments are variables.

[1] removing quantifiers from first order logic can be automated with TPTP utility using TPTP2X utility, option `-t clausify` 2.3.3.2

5.3. Dataset with system properties

The goal is to generate liveness and safety clauses, but do not mix them. In order to achieve that, class $PredicateGenerator$ needs to be modified to yield either formulas with all variables (safety) or formulas with 0-arity functor (liveness). To achieve that $PredicateGenerator$ was subclassed (see picture 5.1) and used to create alternative version of $CNFFormulaGenerator$. To control functors parameter $functor_arity$ needs to be set to allow only 0-arity functors, $FunctorGenerator$ subgenerator doe not need to be modified. Using newly defined $SafetyLivenessPredicateGenerator$ new preset is defined: $CNFSafetyLivenessGenerator$.



Picture 5.1. Subclassed PredicateGenerator mimics safety and liveness formulas

In this study the impact of ratio of number of atoms to number of clauses will be presented. Formulas with 1000 atoms and 100, 200, 300, 400, 500 clauses were generated, 50 for each combination, 250 in total. 50 formulas is rather small sample to reason about, but it was chosen because of time and hardware limitations. Number of atoms and number of clauses can vary

within 5%, although solver tends to yield small numbers first in general. The rest of parameters for formulas is shown in listing 9.

```
gen = CNFSafetyLivenessGenerator(
    variable_names={f'V{i}' for i in range(10)},
    functor_names={f'f{i}' for i in range(20)}, functor_arity={0},
    functor_recursion_depth=0,
    predicate_names={f'p{i}' for i in range(20)}, predicate_arities={i for i in
    ↪  range(5)},
    atom_connectives={''},
    clause_lengths={i for i in range(2, 11)},
    number_of_clauses=IntegerRange.from_relative(number_of_clauses, threshold),
    number_of_literals=IntegerRange.from_relative(number_of_literals, threshold),
    literal_negation_chance=0.1,
)
```

Listing 9. Snippet for generating dataset of safety and liveness formulas

5.4. Used solvers and results

Formulas were generated and were benchmarked against solvers Prover9 and SPASS.

5.4.1. Prover9

Prover9 [13] is an automated theorem prover for first-order and equational logic. Prover9 ships with library called LADR which also defines its formula input syntax. Prover9 has a fully automatic mode in which the user simply gives it formulas representing the problem, but it also has command line options to modify reasoning process. Along Prover9 there is program called Mace4 which looks for finite models and counterexamples. Finding counterexample helps avoid wasting time searching for a proof.

5.4.2. SPASS

SPASS is an automated theorem prover for full sorted first order logic with equality. SPASS accepts formulas in TPTP format or a custom one. SPASS has a number of command line options which affect reasoning process. There are also number project based on SPASS: TSPASS – SPASS prover which operates in temporal logic, SPASS-SATT – solver for ground linear arithmetic, SPASS-IQ – linear arithmetic solver.

5.4.3. Results

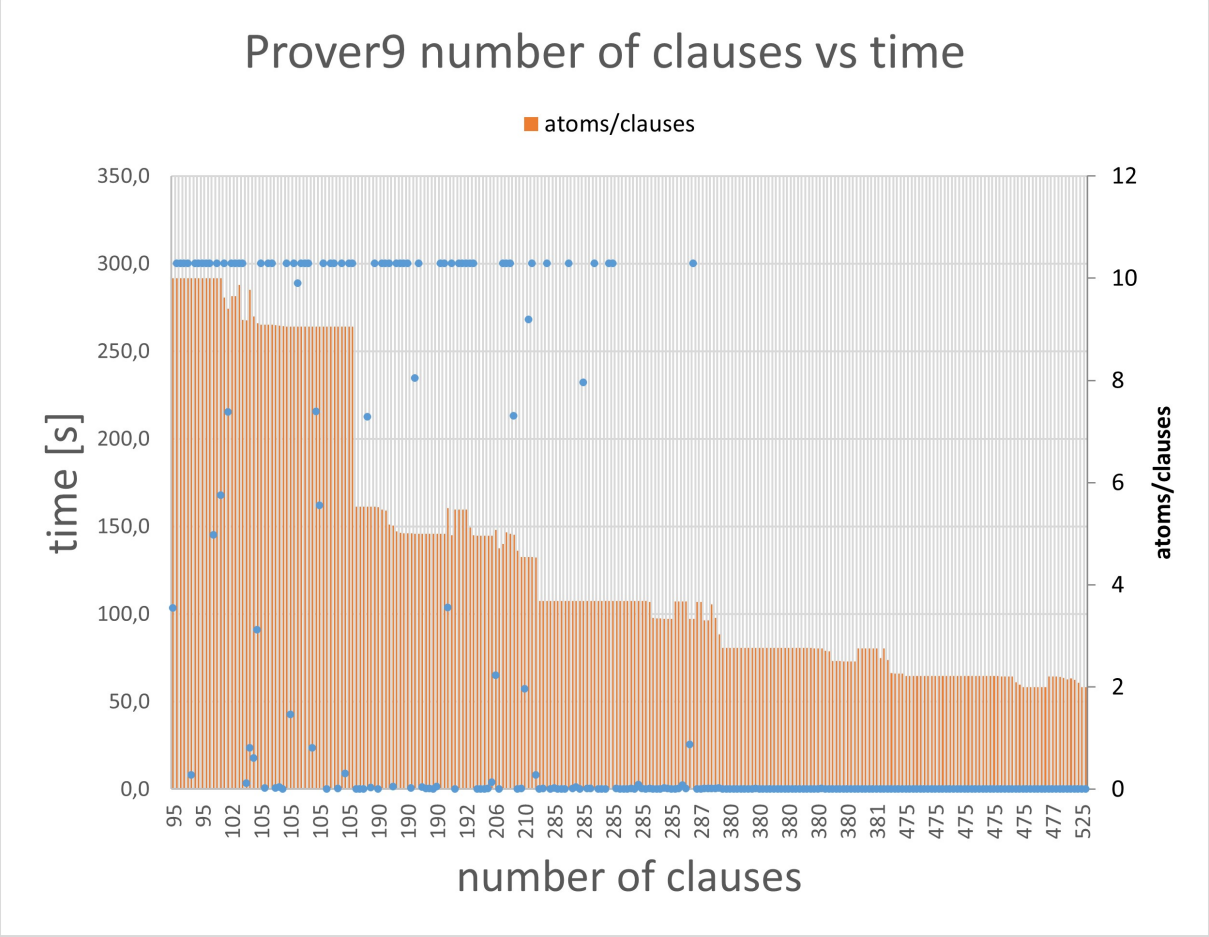
Following charts present results in similar manner: on each chart there are 2 graphs: orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents measured value: either time of execution (in seconds) or peak memory usage during execution. Maximum execution time of single formula was trimmed at 300 seconds (it remains unknown if formula is satisfiable or unsatisfiable). If execution time of formula is less than 300 seconds, it means it that solver concluded if formula is satisfiable or unsatisfiable, it is not relevant which one.

First thing worth noticing based on picture 5.2 and 5.3 is SPASS solver is capable of solving much more formulas than Prover9. SPASS returned only 5 timeouts which compared to Prover9 56 timeouts out of 250 test cases works to the advantage of SPASS over Prover9 in tested area. This difference is beyond measurement error even though generated formulas are random.

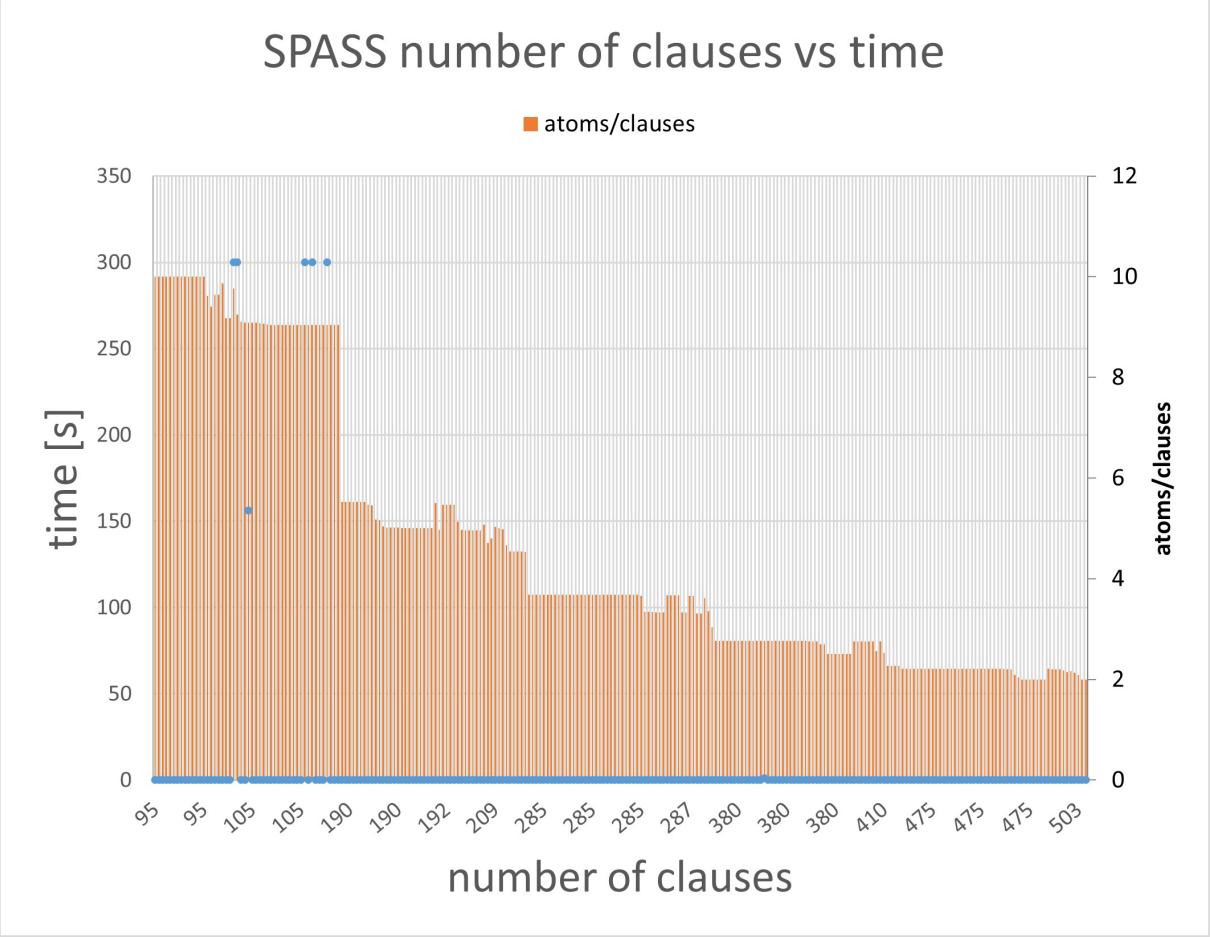
SPASS to a certain extent uses constant memory (picture 5.5) whereas Prover9 (picture 5.4) requires notably more memory the longer it runs. Memory range used by SPASS varies between 56 Mb and 83 Mb, whereas Prover9 uses much wider range of memory - between 2 Mb up to 555 Mb.

The most important observation based on picture 5.2 and 5.3 is the higher the atoms/clauses ratio the longer the time of execution is thus timeouts are more frequent. The turning point of this observation seems to occur around atoms/clauses ratio equal to 5, at least for Prover9, as at this point more and more timeouts happen. This observation for SPASS might not so viable as this there are only 5 timeouts and the rest of formulas were computed in similar time (243 formulas in less than 1 second) therefore result can be considered a coincidence as dataset is random. We can get more hints of the same conclusion for SPASS as we have for Prover9 from memory charts in picture 5.5 where visibly for higher atoms/clauses ratio more than average of 60 Mb of memory is used, and for the lower ratio memory usage drops slightly below the average. In other words it can be said, that for the constant number of atoms, formulas with more clauses are solved quicker.

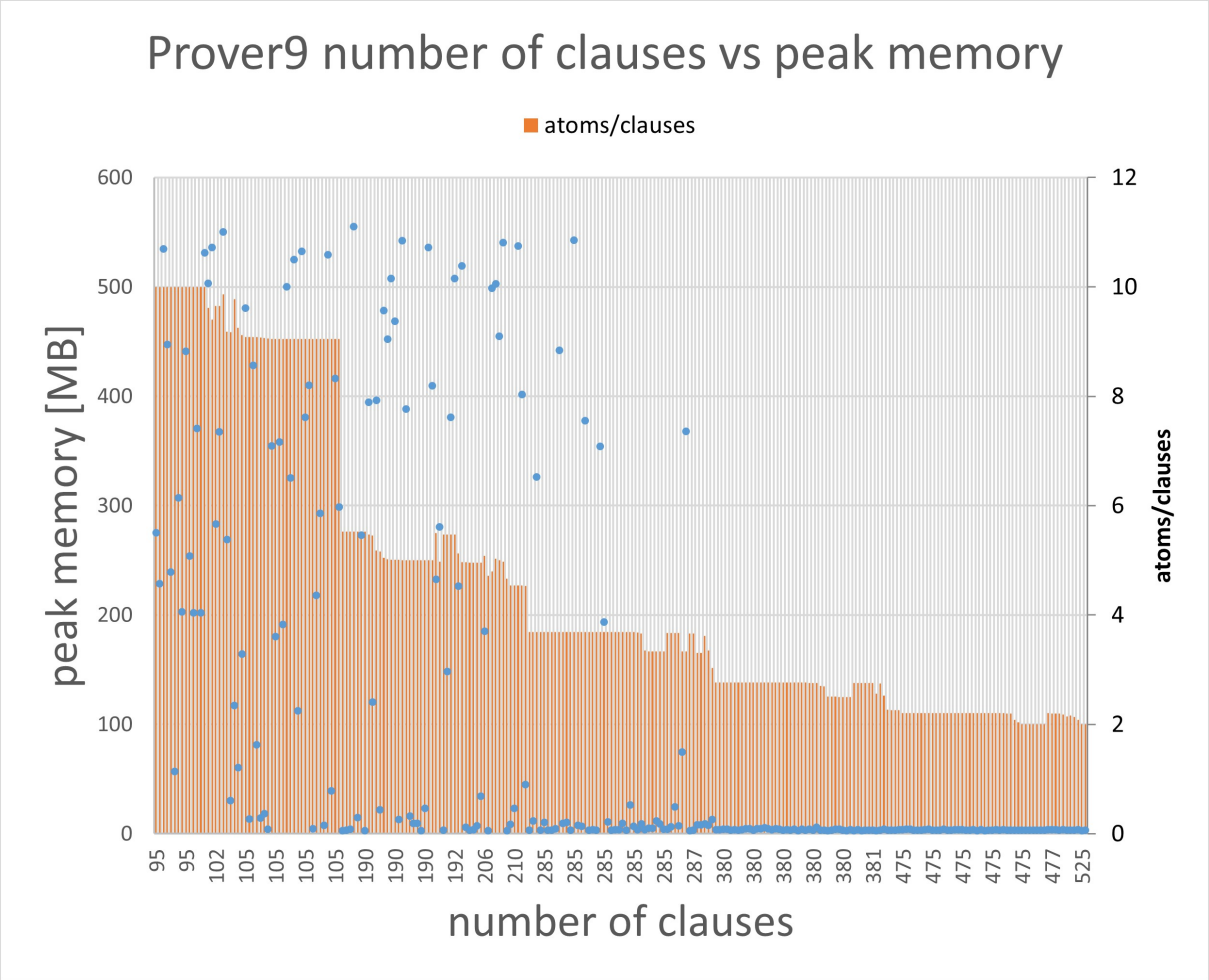
The range of number of atoms was specified in input parameters for generator and is presented in picture 5.6 and 5.7. The influence of number of atoms threshold of 5% is insignificant to the influence of atoms/clauses ratio.



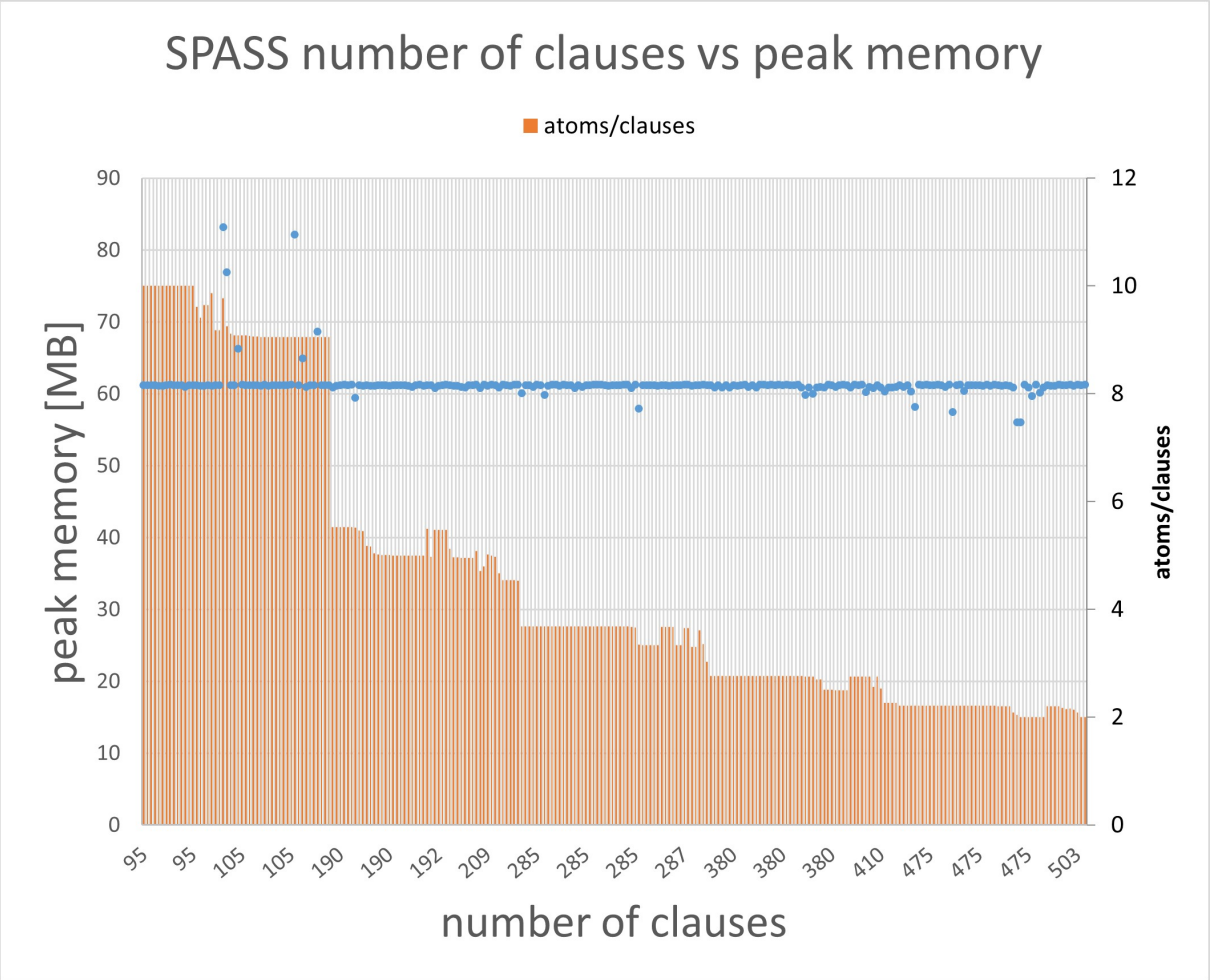
Picture 5.2. Prover9 number of clauses vs time, formulas are sorted by number of clauses. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents time of execution. After 300 seconds computations were terminated, without result.



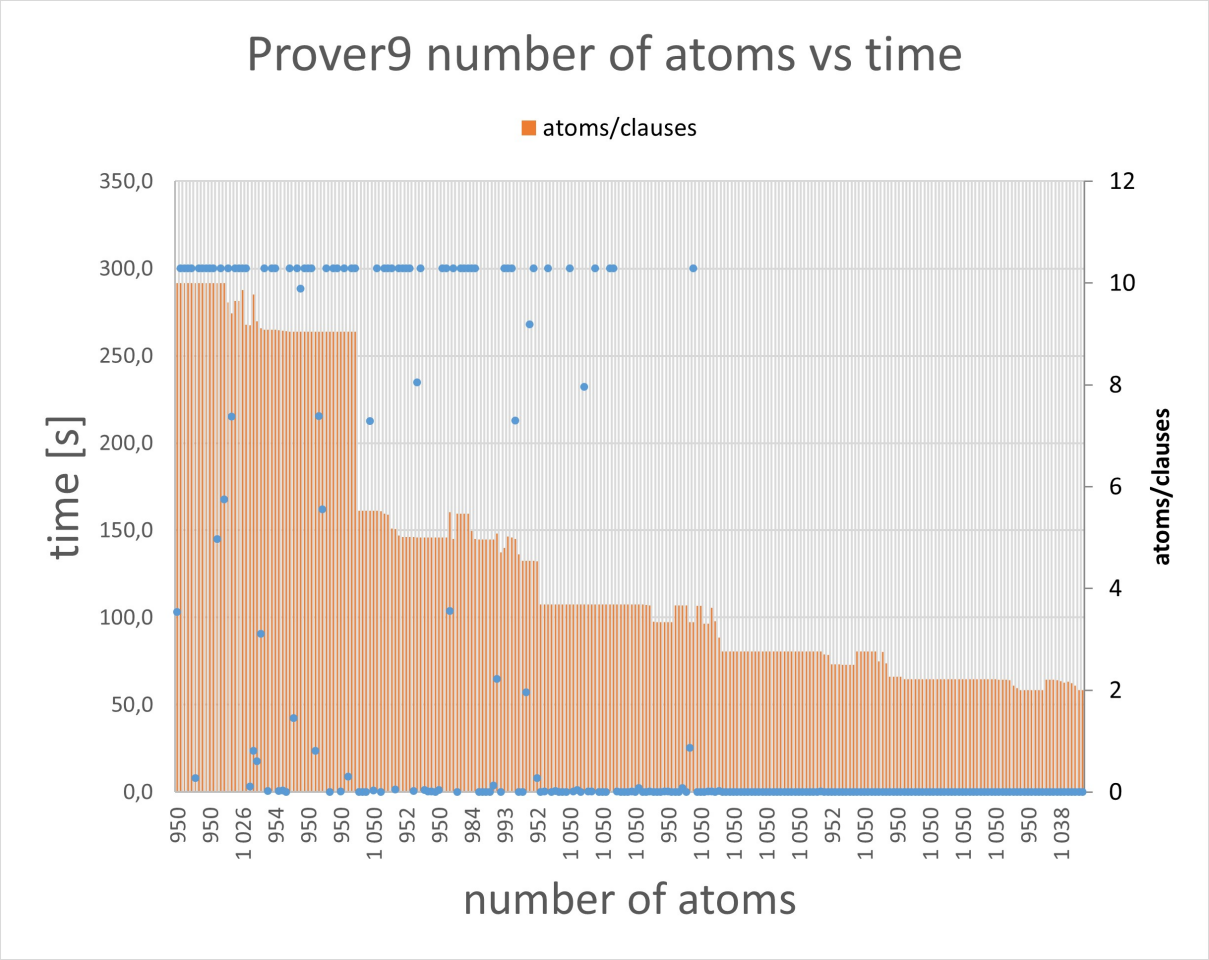
Picture 5.3. SPASS number of clauses vs time, formulas are sorted by number of clauses. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents time of execution. After 300 seconds computations were terminated, without result.



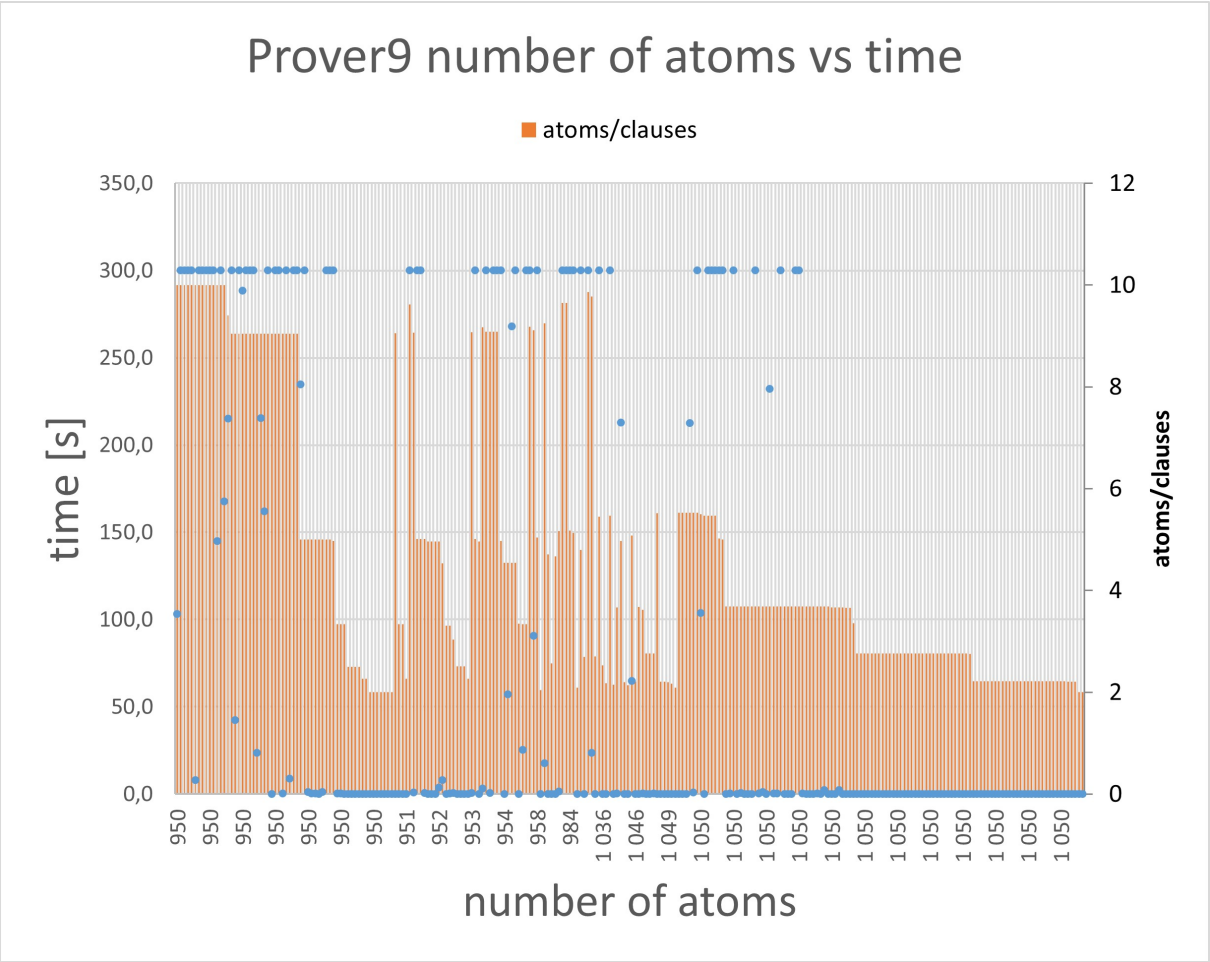
Picture 5.4. Prover9 number of clauses vs peak memory, formulas are sorted by number of clauses. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents peak mmeory usage.



Picture 5.5. SPASS number of clauses vs peak memory, formulas are sorted by number of clauses. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents peak mmeory usage.



Picture 5.6. Prover9 number of atoms vs time of execution, formulas are sorted by number of clauses. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents time of execution. After 300 seconds computations were terminated, without result.



Picture 5.7. Prover9 number of atoms vs time of execution, formulas are sorted by number of atoms. Orange bar graph represents atom to clause ratio (2 means there are 2 atom for each clause) and blue scatter chart represents time of execution. After 300 seconds computations were terminated, without result.

6. Conclusion

In this thesis a base for random first order logic formula generator has been presented. The generator can be easily extended for user specific needs. For example user can add arbitrary rule (by inheriting one of subgenerators) that elements must follow during generation. This enables generating problems on the fly with any complexity. Presented solution can be extended to follow constraints related to any other FOL element like limited number of variables, functors and so on.

The biggest challenge in described solution is randomizing formulas within given range. Solving constrains given by input parameters (as formalized in section 4.4.1 is problem of integer programming (NP problem) that is why enumerating them all is not an option. Instead random number of solutions is skipped to preserve at least pseudo-randomness.

Alternative approach to using SMT solver would be to take locally optimal/random decisions and introduce some randomness when taking those decisions (hit-or-miss approach). Approximate algorithm in this approach would be:

1. start with empty formula
2. add clause with random number of literals
3. if input parameters are met - stop
4. if input parameters are not met - go to point 2

This algorithm has a number of edge cases which would require restarts but what is more important this approach was not used as it is hard to expand. In future development of this generator more parameter would be profided as ranges of possible value, for example user could request formula with 50 to 100 clauses, 400 to 500 literals, 70 to 80 predicates and 50 to 60 functors. A nice ability would be to suggest better parameters when user provided values are not possible.

Acronyms

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

CNF – conjunctive normal form

DNF – disjunctive normal form

FOF – First Order Formula

FOL – First Order Logic

SAT – boolean satisfiability problem

SMT – Satisfiability Modulo Theory

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

Bibliography

[1] João P. Marques Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers." In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Sept. 16, 2009, pp. 131–153. ISBN: 978-1-58603-929-5.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

[3] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0". In: *Journal of Automated Reasoning* 59.4 (2017). TPTP is available online at http://www.tptp.org, pp. 483–502.

[4] W.W. McCune. "Otter: An Automated Deduction System". http://www.cs.unm.edu/ mccune/otter/.

[5] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[6] G. Sutcliffe and S. Melville. "The Practice of Clausification in Automatic Theorem Proving". In: *South African Computer Journal* 18 (1996), pp. 57–68.

[7] Thomas Stützle Holger H. Hoos. "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0". In: *I.P.Gent, H.v.Maaren, T.Walsh* (2000). SATLIB is available online at www.satlib.org, pp. 283–292.

[8] Henry Yuen and Joseph Bebel. "Tough SAT". |https://toughsat.appspot.com/|.

[9] Massimo Lauria. "CNM Gen". |https://massimolauria.net/cnfgen/|.

[10] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary: Springer-Verlag, 2008, 337–340. ISBN: 3540787992.

[11] L. Lampert. “Proving the correctness of multiprocess programs”. In: *IEEE Transactions on Software Engineering* SE-3.2 (1977), pp. 125–143.

[12] Radosław Klimek. *Wprowadzenie do logiki temporalnej*. Kraków : AGH. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, 1999, pp. 104–142.

[13] W. McCune. “Prover9 and Mace4”. |http://www.cs.unm.edu/ mccune/prover9/|. 2005–2010.