UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Verificarea algoritmului DPLL in F*

propusă de

Alexandru Donica

Sesiunea: iunie/iulie, 2023

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Verificarea algoritmului DPLL in F*

Alexandru Donica

Sesiunea: iunie/iulie, 2023

Coordonator științific

Conf. Dr. Ștefan Ciobâcă

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Ștefan Ciobâcă.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Donica Alexandru** domiciliat în **România, jud. Iași, mun. Iași, strada Costache Negri, nr. 35, bl. A1, ap. 42**, născut la data de **07 aprilie 2000**, identificat prin CNP **5000407226761**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2022, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Verificarea algoritmului DPLL in F*** elaborată sub îndrumarea domnului **Conf. Dr. Ștefan Ciobâcă**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Verificarea algoritmului DPLL in F***, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea ”Alexandru-Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Alexandru Donica**

Data: ..... Semnătura: .....

Cuprins

Motivație

Rolul unui 'SAT solver' este de a rezolva problema satisfiabilității booleene, făcându-se si in ziua de azi cercetări care au scopul de a îmbunătăți algoritmi existenți. Acest 'SAT solver' găsește o soluție pentru o formula data in cazul in care formula este satisfiabilă, in caz contrar formula este nesatisfiabilă.

Corectitudinea oricărui rezultat al unei formula satisfiabile poate fi verificata folosind algoritmi simpli. Însă un 'SAT solver' complex creat pentru a procesa formule de dimensiuni din ce in ce mai mari sau pentru a avea o viteza de rezolvare a problemei mai rapida decât alți 'SAT solveri', poate conține erori de programare ce ar produce rezultate false, cum ar fi, in urma procesării unei formula nesatisfiabile acesta sa enunțe ca este satisfiabilă, sau invers.

De aceea este importanta crearea unor programe care verifica corectitudinea acestor 'SAT solveri' si am creat un 'SAT solver' verificat formal folosind limbajul de programare F* (FStar).

Introducere

Problema satisfiabilitatii booleene (SAT) se regaseste in multe ramuri ale domeniului informaticii, precum in verificarea de software si hardware, in domeniul criptografiei si securitatii sau optimizarea si analiza altor programe.

'SAT' este considerata prima problema ce a fost demonstrata a fi NP-completa [1], motiv pentru care este importanta dezvoltarea unor programe care sa poata gasi o solutie cat mai rapid, eficient si sigur.

In ziua de astazi exista si inca se dezvolta si optimizeaza 'SAT solveri' eficienti, cum ar fi cei care au participat la concursul "SAT 2022": Mallob, sau Paracooba.

Algoritmul DPLL este una din primele optimizari eficiente folosite in rezolvarea problemei SAT. Este considerat a fi un punct de plecare fundamental pentru dezvoltarea unui 'SAT solver' eficient, stand la baza multor 'SAT solveri' moderni.

Insa chiar si o implementare a unui algoritm demonstrat a fi eficient si complet poate produce erori sau rezultate gresite, iar unul din motivele pentru care apar aceste erori este eroarea umana a programatorului, a carei frecventa poate creste impreuna cu complexitatea algoritmului ce trebuie implementat. Pentru ca este testarea unui 'SAT solver' nu poate garanta absenta erorilor, s-au conceput programe care sa verifice ca implementarea in sine a algoritmului este lipsita de erori si matematic corecta.

Limbajul de programare F* (FStar) pune la dispozitie un astfel de sistem care in momentul compilarii executa o verificare formala a programului in functie de specificatiile introduse de programator. Acest sistem ajuta la depistarea bug-urilor, prevenirea erorilor, si ofera o siguranta mai mare ca programul este corect implementat si nu va prezenta rezultate false sau gresite.

Astfel, am ales sa implementez algoritmul DPLL folosind limbajul de programare FStar, pentru a evidentia cum arata parcursul crearii si verificarii unui 'SAT solver' in acest limbaj si cum se poate extinde din punct de vedere al eficientizarii.

[1] https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem

Aceasta lucrare va prezenta in continuare o implementare a algoritmului DPLL folosind limbajul FStar. Vor fi prezentate detalii despre algoritmul DPLL, despre limbajul de programare folosit, detalii de implementare cu exemple de cod si motivele pentru care 'SAT solver-ul' rezultat este complet, corect si produce mereu un rezultat.

Capitolul 1

Algoritmul DPLL (Davis-Putnam-Logemann-Loveland)

Algoritmul DPLL pentru rezolvarea problemei SAT este considerat o optimizare a metodei de backtracking si sta la baza crearii a multor 'SAT solveri' moderni.

Acest algoritm a fost demonstrat sa aibe proprietatea ca pentru orice formula care se poate scrie in forma normala conjunctiva (CNF), va returna o solutie valida daca aceasta exista, si va specifica mereu ca nu exista daca formula este intr-adevar nesatisfiabila.

Optimizarea cea mai importanta pe care acest algoritm o aduce metodei backtracking este 'propagarea clauzelor unit', unde o 'clauza unit' este o clauza ce nu contine nici un literal cu valoarea *True* si care contine un singur literal neasignat inca.

Astfel, la fiecare pas recursiv, se cauta fiecare clauza 'unit' si se asigneaza o valoare variabilei ramase astfel incat literalul din clauza respectiva sa aibe valoarea True. Motivul pentru care aceasta metoda functioneaza este ca daca s-ar asigna cealalta valoare de adevar unui astfel de literal, valoarea de adevar a clauzei respective ar fi *False*, deci solutia ar fi invalida.

Urmeaza pseudo-codul metodei recursive 'dpll', formulata dupa algoritmul DPLL modificat sa corespunda implementarii acestui proiect proiectului.

Input: formula : f { length f ≥ 1 } ;

truth_assignment : tau{ length tau $\leq$ length f $\wedge$ is_partial_solution f tau }

Output: result : r { (r is (Sat tau) $\implies$ *calculate_formula_value* f tau = true)

$\wedge$ (r is Unsat $\implies$ $\forall$ truth_assignment : t .

calculate_formula_value f ta = false) }

if *length tau = length (**variables_of_f** f)* **then**
  return *Sat tau*
end
else
  new_var $\leftarrow$ ***get_unassigned_variable*** *f tau*
  tau_2 $\leftarrow$ *tau[new_var : False]*
  tau_2 , new_literals $\leftarrow$ ***unit_clause_propagation*** *f tau_2*
  clauses_with_new_literals $\leftarrow$ *get_clauses_that_contains_min_1_literal f new_literals*
  if $\exists$ *c : clause c* $\in$ *clauses_with_new_literals* $\wedge$ *is_clause_false c tau_2* **then**
    tau_2 $\leftarrow$ *tau[new_var : True]*
    ...aceeiasi pasi ca pana acum, dar cu noul *tau_2* ,si returneaza *NotSat* in block-ul *if-then* corespondent...
  end
  else
    res $\leftarrow$ *dpll_recursiv f tau_2*
    if *res is NotSat* **then**
      tau_2 $\leftarrow$ *tau[new_var : True]*
      ...aceeiasi pasi ca pana acum, dar cu noul *tau_2* ,si returneaza *NotSat* in block-ul *if-then* corespondent...
    end
    else
      return *res*
    end
  end
end

Algorithm 1: dpll-recursiv

Exemplu: formula ce contine 5 clauze, numerotate de la c_1 la c_5 si 6 variabile numerotate de la x_1 la x_6

$$(\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_4 \vee \neg x_6)$$

Implementarea acestui proiect ar incepe cautarea unei solutii asignand unei variabile noi o valoare oarecare, mai exact, variabilei x_1 valoarea *False*. Urmeaza etapa de unit propagation, in care se gaseste clauza c_3 a fi clauza unitara, si se asigneaza variabilei x_2 valoarea *False*. Apoi, recursiv, se ia logica de la capat cu datele noi obtinute. Se va gasi primul literal neasignat x_3 si i se va acorda valoarea *False*. In etapa de propagare a clauzelor unitare se gaseste prima clauza ce contine 2 literali cu valoare falsa si variabila x_4, care va primi valoarea *True*. Tot in etapa de propagare a clauzelor unitare, se gaseste acum clauza c_5, unde singurul literal ramas este $\neg x_6$, si deci variabila x_6 va primi valoarea *False*. Astfel toate clauzele au valoarea true, iar variabila x_5 poate avea orice valoare de adevar. Deci s-a gasit o solutie valida si formula este satisfiabila.

$$x_1 = False;\ x_2 = False;\ x_3 = False;\ x_4 = True;\ x_5 = orice;\ x_6 = False$$

Capitolul 2

Detalii de implementare

2.1 Despre FStar

Limbajul F* este un limbaj de programare orientat pe demonstratii ce poate fi folosit pentru uz general. Printre aspectele acestui limbaj se numara si faptul ca prezinta suport pentru programare functionala. De asemenea, este proiectat sa faca verificare formala a codului, astfel orice programator poate sa adauge functiilor specificatii in forma de pre-conditii, post-conditii, invarianti, pentru a putea asigura completitudinea codului.

2.2 Completitudinea, corectitudinea si terminarea programului

Cele mai importante proprietati pe care le respecta aceast 'SAT solver' sunt completitudinea algoritmului, corectitudinea implementarii si siguranta ca programul se va termina si va produce un rezultat.

2.2.1 Completitudinea programului

Este dovedita completitudinea algoritmului DPLL pentru formula in forma normala conjunctiva (CNF), asa cum este el descris in Capitolul 1.

Singura optimizare a algoritmului DPLL care a fost implementata se leaga de verificarea existentei unei clauze false la fiecare pas recursiv dupa etapa de *'unit_propagation'*.

Mai exact, la fiecare pas exista o lista de literali de lungime cel putin 1 care au fost adaugati la solutie, dar acesti literali noi nu schimba valoarea clauzelor care nu contin nici unul din literalii din lista.

Astfel aceasta optimizare se foloseste de o matrice de aparitie pentru fiecare literal din formula cate o lista cu toate clauzele in care apare respectivul literal.
Se intersecteaza clauzele in care apar toti acesti literali asignati la pasul curent, iar cautarea unei clauze false se face doar peste aceasta multime, in loc de intreaga formula.

2.2.2 Corectitudinea programului

Pentru verificarea specificatiilor introduse de programator, se foloseste un SMT solver numit Z3 [1]. Cu toate acestea, unele conditii pot fi prea dificile de demonstrat intr-un punct anume al programului, unul din motivele pentru care limbajul F* ofera la dispozitia programatorilor instructiuni ce simplifica misiunea de a demonstra si verifica proprietati si conditii scrise in program. Recurente in cadrul acestui proiect sunt structurile 'Lemma' si instructiunile 'assert' care ajuta la demonstrarea unor proprietati intre diferite tipuri de date ce respecta anumite conditii, si ajuta 'SMT solver-ul' spunandu-i sa demonstreze intai proprietati intermediare.

Exemplu 'lemma':

```
let rec lemma_values_from_clause_same_for_all_supra_set_of_t
(t : truth_assignment)
(c : clause)
: Lemma
(requires are_clause_vars_in_assignment t c)
(ensures
        (forall (other_t : truth_assignment
                {forall (v : variable_info{L.mem v t}).
                        (L.mem v other_t)}).
                (L.mem true (get_values_from_clause t c)
                = L.mem true (get_values_from_clause other_t c)))
)
=
if length c = 1
```

[1] https://en.wikipedia.org/wiki/Z3_Theorem_Prover

```
then ()
else lemma_values_from_clause_same_for_all_supra_set_of_t t (L.tl c)
```

Aceasta lemma ajuta la demonstrarea faptului ca pentru orice clauza *c*, ai carei literali au o valoare asignata in multimea *t*, atunci valoarea de adevar a clauzei *c* in corespondenta cu multimea *t* va fi egala cu valoarea de adevar a lui *c* in corespondenta cu orice multime care este supra-set pentru *t*.

Exemplu instructiune 'assert':

```
assert(L.noRepeats ress);
```

Aceasta simpla instructiune ajuta a arata faptul ca la momentul verificarii acestei instructiuni, lista 'ress' nu contine elemente duplicate.

Exemple asemanatoare se vor gasi si in sectiunile ce urmeaza.

Dupa verificarea unui fisier '.fst' se va extrage, in fisiere tipice limbajului Ocaml, codul ce este folosit la formarea rezultatului si este eliminat codul ce ajuta la specificatii si demonstratiile unor proprietati. Cu cat validarile devin mai complexe, cu atat se poate observa diferenta mare de linii de cod dintre fisierele '.fst' si cele '.ml'. De exemplu, modulul 'DpllPropagation' contine multe instructiuni ce ajuta la verificare, in total ajungand la ≈600 de linii fisierul, insa corespondendul modul Ocaml extras din cel FStar, are ≈100 linii, insemnand ca s-au eliminat cele ≈500 linii care contribuiau la verificarea modulului.

2.2.3 Terminarea programului

Terminarea programului si producerea unui rezultat trebuie sa fie o certitudine, indiferent de valoarea rezultatului. Limbajul FStar poate verifica, daca este specificat acest lucru, daca o functie/un program se va termina si va produce un rezultat indiferent de parametrii primiti. De asemenea poate verifica daca programul/functia ar produce acelasi rezultat daca ar fi apelata de mai multe ori cu aceeiasi parametrii.

Programatorului i se ofera posibilitatea de a adnota functii folosind cuvantul "Tot" pentru a asigura ca la momentul compilarii se va incerca si demonstrarea faptului ca fiecare functie, mai ales cele recursive, se termina, produc un rezultat si mereu acelasi rezultat pentru aceleasi argumente.

Toate metodele din modulele ce ajuta la formarea rezultatului algoritmului DPLL sunt 'functii totale', fie folosind adnotarea 'Tot' in signatura functiei, fie prin absenta oricarei adnotari, efectul 'Tot' fiind cel implicit. Succesul compilarii programului inseamna ca acesta se termina si fiecare input are asociat un singur rezultat. In cazul in care o formula are mai multe solutii, modul in care a fost implementat algoritmul va decide care din acele solutii se va returna.

Acesta este un exemplu de declaratie a unei functii ce contine cuvantul cheie "Tot":

```
let rec are_variables_in_truth_assignment'
(vars: list nat_non_zero {
        L.noRepeats vars
        /\ length vars > 0 }
)
(t: truth_assignment)
: Tot
(res : bool {
        ((all_variables_are_in_truth_assignment' vars t ) <==> res) }
)
(decreases length vars)
```

Se observa faptul ca metoda este recursiva si ca este nevoie de a specifica modul in care dimensiunea parametrilor scade astfel incat la un moment dat va ajunge intr-un punct minim in care functia nu se va mai apela pe sine, mai exact prin ultima linie prefixata de cuvantul cheie *decreases*.

2.3 Structurile de date folosite

S-a ales proiectarea unor structuri de date si colectii simple, pentru care au fost create functii ce implementeaza operatii generice.

Conform definitiei, o formula este o multime de clauze, astfel implementarea are forma unei liste inlantuite de clauze.

```
type formula = f : list clause
```

O clauza este o multime de literali, de aceea structura ei apare ca o lista inlantuita de literali.

```
type clause = c : list literal { length  c > 0}
```

Un literal reprezinta o variabila a carui valoare de adevar poate fi negata, motiv pentru care tipul de data 'literal' este creat astfel, folosind 2 constructori 'Var' pentru a evidentia ca nu se neaga valoarea de adevar a variabilei, si 'NotVar' pentru a arata negatia variabilei.

```
type literal =
        | Var : a: nat_non_zero -> literal
        | NotVar : a: nat_non_zero -> literal
```

Doi literali, 'Var x' si 'NotVar x' sunt considerati a nu fi egali, insa au variabilele egale.

Variabilele sunt reprezentate folosind numere naturale nenule, deoarece un format des intalnit al datelor de intrare pentru aceasta problema si care s-a considerat la conceperea proiectului reprezentau literalii folosind numere intregi nenule. Am ales literalii sa fie reprezentati prin 2 constructori alaturi de numere intregi strict pozitive pentru a evita verificari repetate in cod cu privire la semnul unei variabile oarecare si pentru a se putea face usor distinctia intre tipurile de literali.

```
let nat_non_zero = x:int{x > 0}
```

Pentru matricea de aparitie a clauzelor si literalilor, s-a folosit o structura de date intermediara care reprezinta o linie a matricei si este formata dintr-o pereche de 2 elemente, un literal si o lista inlantuita de clauze.

```
type aux_occurence_vector =
        {lit : literal ; clauses : list clause }


type occurence_vector = oc_v : aux_occurence_vector
                {length oc_v.clauses > 0
                /\ (forall (c : clause{L.mem c oc_v.clauses}).
                        (L.mem oc_v.lit c))}
```

Conditia acestui tip de date este ca lista 'clauses' trebuie sa contina toate clauzele in care apare literalul 'lit' din formula primita la inceperea programului.

Matricea in sine este o lista inlantuite de 'occurence-vector', si trebuie mereu sa respecte conditia ca nu exista 2 elemente in lista a caror membrul 'lit' sa fie egal ca si valoare.

```
let rec occurence_matrix_condition
        (oc_matrix : list occurence_vector) =
        (   length oc_matrix > 0
                ==>
        (forall (ocv : occurence_vector{L.mem ocv (L.tl oc_matrix)}).
                (ocv.lit <> (L.hd oc_matrix).lit))
        )
        /\ (length oc_matrix > 0 ==>
                occurence_matrix_condition (List.Tot.tl oc_matrix))

type occurrence_matrix = oc_matrix : list occurence_vector{
        occurence_matrix_condition oc_matrix
}
```

Urmeaza un fragment in care se afiseaza conditiile care trebuie sa fie respectate intre formula 'f' si matricea de aparitie a formulei notata in aceasta functie cu 'res'.

```
        let get_occurence_matrix
        (f : formula {length f > 0})
        : (res : occurrence_matrix{
        length res = length (get_lits_in_formula f)
        /\ (forall (l : literal{L.mem l (get_lits_in_formula f)}).
                (is_lit_in_occurence_matrix res l))
        /\ (forall (oc_v : occurence_vector{L.mem oc_v res}).
        /\ (forall (oc_v : occurence_vector {L.mem oc_v res}).
                (forall (c : clause {L.mem c oc_v.clauses}).
                        (L.mem c f)))
        /\ (forall (oc_v : occurence_vector {L.mem oc_v res}).
                (forall (c : clause {
                        L.mem c f /\ (L.mem c oc_v.clauses = false)}).
                        (L.mem oc_v.lit c = false)))})
```

Rezultatul functiei ce verifica satisfiabilitatea unei formule este in sine un o structura de date simpla creata pentru aceasta problema, reprezentata prin doi constructori, unul pentru situatia in care formula e nesatisfiabila si unul care contine o posibila combinatie de variabile pozitive si valori booleene ce reprezinta o solutie valida pentru formula.

```
type result =
        | NotSat
        | Sat : a: truth_assignment{ length a > 0} -> result
```

In aceasta lucrare, se va referi la o combinatie de variabile pozitive si valori booleene prin termenul 'truth-assignment' prescurtat 't' sau 'tau'.

Tipul de data 'truth_assignment' reprezinta o lista inlantuita de elemente, structuri formate din doi parametri, unul 'value' ce reprezinta valoarea unei variabile si 'var' ce reprezinta in sine variabila.

```
type variable_info = {value: bool; variable: nat_non_zero}


type truth_assignment = ta : list variable_info { ta_contition ta }
```

Acest 'truth_assignment' trebuie sa respecte conditia ca nu exista 2 elemente in lista care sa aibe aceeasi valoare a membrului 'var'. Invariantul a fost scris astfel simplificand conditia, si anume ca membrul 'var' al elementului din capul listei trebuie sa nu apara in oricare alt element din lista, iar daca submultimea 'tail_t' formata din toate elementele lui 't' cu exceptia capului listei trebuie sa respecte aceeasi proprietate.

```
let rec ta_contition (ta : list variable_info) =
(forall (var : variable_info {List.Tot.mem var ta}).
        (count_variables_occurrence ta var.variable = 1 ))
        /\ (length ta > 0 ==> ta_contition (List.Tot.tl ta))
```

Structurile de date au fost alese a fi simple in detrimentul eficientei vitezei de executie a programului, pentru a putea analiza dificultatea si complexitatea mentinerii corectitudinii in implementarrea unui algoritm complet in limbajul F*.

2.4 Modulele ce alcatuiesc programul

Separarea programului pe mai multe fisiere s-a realizat pentru a usura dezvoltarea si modificarea codului in etapa implementarii. De asemenea, orice schimbare intr-un fisier rezulta in re-verificarea acestuia la momentul compilarii, proces care devine cu atat mai indelungat cu cat fisierul prezinta mai multe functii si demonstratii complexe.

Astfel, modulele acestui program sunt:

• DataTypes - fisier unde s-au definit toate structurile de date explicate anterior, invariantul pentru cele ce aveau nevoie de unul, si anumite functii ajutatoare pentru a manipula tipurile de date tip colectii demonstrand in acelasi timp corectitudinea operatiilor efectuate pe colectiile respective.

Un exemplu de astfel de functie ar fi metoda *'add_var_to_truth_assignment'*, al carei simplu scop este de a adauga un nou element in colectia *'truth_assignment'*, insa care trebuie sa poata respecta urmatoarea post_conditie.

```
let add_var_to_truth (t : truth_assignment) (var : variable_info {count
: Tot
(res : truth_assignment  {
is_variable_in_assignment res var.variable
/\ List.Tot.mem var res
/\ count_variables_occurrence res var.variable = 1
/\ (length res = length t + 1)
/\ (forall (v : variable_info {
        List.Tot.mem v res /\ v <> var}). (List.Tot.mem v t))
/\ (forall (v : variable_info {
        (List.Tot.mem v t)}). ((List.Tot.mem v res)))
/\ (forall (v: variable_info {
        List.Tot.mem v res = false}).
        ((List.Tot.mem v t = false)))
/\ (forall (l : literal{
        is_variable_in_assignment t (get_literal_variable l) }).
        (is_variable_in_assignment res (get_literal_variable l)
/\ get_literal_value t l = get_literal_value res l))
```

Pe scurt, sustine ca noua lista trebuie sa contina toate elementele din multimea veche, sa aibe lungimea cu 1 mai mare doar decat lista initiala, iar toate variabilele prezenta in lista initiala au aceeasi valoare in ambele multimi.

Modulul 'DataTypeUtils' contine un numar considerabil de metode ajutatoare, folosite in mai multe module ale proiectului si care trebuie deci sa fie disponibile intr-un singur loc. Printre aceste metode apar:

- functii de procesesare a parametrilor primiti;

  ex: *get_clause_value* - returneaza valoarea de adevar a unei clauze considerand un *'truth_assignment'* primit ca parametru;

- *lemme* care ajuta la asigurarea si demonstrarea corectitudinii programului;

  ex: *lemma_no_vars_in_t_outside_f_length_compare* folosita pentru a arata ca daca un 'truth_assignment' nu contine nici o variabila care nu este prezenta in formula 'f', atunci sigur lungimea variabilei 't' este mai mica sau egala cu numarul variabilelor distincte ce apar in formula 'f';

```
let rec lemma_no_vars_in_t_outside_f_length_compare
(vars: list nat_non_zero)
(t : truth_assignment)
: Lemma
(requires
        L.noRepeats vars
        /\ (no_variables_outside_f_are_in_t' vars t)
)
(ensures length t <= length vars )
(decreases (length t)) =
if length t = 0
then let l = length vars in ()
else let new_t =
        (remove_variable_from_assignment t (L.hd t).variable) in
let new_vars =
        (remove_var_from_list vars (L.hd t).variable) in
lemma_no_vars_in_t_outside_f_length_compare new_vars new_t
```

- pre/post-conditii importante si relevante demonstrarii corectitudinii salvate in variabile globale, pentru a reduce cantitatea de cod repetat si pentru a putea generaliza functiile la nivel inalt, avand posibilitatea in viitor de a optimiza aspecte ale programului precum structurile de date, pentru care ar trebui modificate doar implementarea metodelor ce proceseaza aceste structuri;

  ex: *t_cant_be_solution_for_f*, folosit ca post-conditie care enunta ca orice 'truth_assignment' ce contine toate variabilele formulei 'f' si pentru care parametrul 't' este o sub-multime a sa, nu este o solutie valida pentru 'f';

  t_cant_be_solution_for_f:

```
(forall (whole_t: truth_assignment{
        t1_is_sublist_of_t2 t whole_t
        /\ are_variables_in_truth_assignment f whole_t
        /\ length whole_t
                = length ( get_vars_in_formula f)}).
        (is_solution f whole_t = false))
```

- predicate, functii care evalueaza daca anumiti parametrii respecta o anumita proprietate

  ex: *is_solution*, desi simplu, este unul din cei mai importanti predicati ai programului; acesta sustine ca daca un *'tau'* contine toate variabilele din formula 'f', atunci este solutie doar daca este solutie partiala, adica daca nu exista clauze false.

```
let is_solution
(f: formula  { length f > 0 })
(t: truth_assignment {
        ((length (get_vars_in_formula f)) = length t)
        /\ all_variables_are_in_truth_assignment f t
})
= is_partial_solution f t
```

Modulul DpllPropagation conține puține metode însă importante pentru demonstrate si dificil la rândul lor de specificat si verificat. Aceste funcții se ocupă cu pasul de propagare a clauzelor 'unit' din algoritmul DPLL.

O mica parte însa cea mai importanta din specificarea metodei principale este:

```
        t1_is_sublist_of_t2 t (fst res)
        /\ ((t_cant_be_solution_for_f f t) <==> (
                t_cant_be_solution_for_f f (fst res)) )
        /\ length res._1 >= length t
```

Post-condițiile enunță următoarele:

- faptul ca tot ce e inclus in 'tau' primit ca parametru trebuie sa fie existent si in 'tau' trimis ca rezultat;
- lungimea 'tau'-ului rezultat trebuie sa fie macar egala cu lungimea 'tau'-ului primit, lucru care ajuta la asigurarea terminarii programului;
- 'tau'-ul primit ca parametru nu poate fi solutie pentru formula 'f' daca si numai daca 'tau'-ul rezultat nu poate fi solutie

Modulul 'OccurenceMatrix' contine functii necesare crearii, procesarii si accesarii matricei de aparitie a literalilor in clauzele formulei si de asemenea implementarea optimizata pentru metoda ce verifica daca un 'tau' oarecare este solutie partiala pentru formula data.

Modulul 'Dpll' contine functia principala ce primeste o formula si ofera un rezultat, unde se specifica si verifica legatura intre valoarea rezultatului si formula.

```
: Tot (res: result {
(Sat? res = true ==>
        ((length (get_truth_from_result res) =
                length ( get_vars_in_formula f))
        /\ (all_variables_are_in_truth_assignment f (get_truth_from_res
/\ ( is_solution f (get_truth_from_result res) = true)))
/\ ((NotSat? res = true) ==>
        ( forall (whole_t: truth_assignment{
        (forall (v : variable_info {(List.Tot.mem v pre_dpll_t)}). ((Li
        /\ are_variables_in_truth_assignment f whole_t
        /\ length whole_t =  length ( get_vars_in_formula f)}).
        is_solution f whole_t = false))
})
```

Modulul 'InputFileParser' a fost conceput pentru a putea folosi formule diverse de diferite dimensiuni prin parsarea unor fisiere de intrare ce respecta un anumit format. [2]

Modulul 'ConvertorToString' este folosit pentru a converti orice obiect creat de program intr-un sir de caractere si mai ales pentru a se afisa rezultatul pe ecran sub o forma usoara de citit si inteles.

2.5 Metrici orientative

Aceasta sectiune contine metrici aproximative sub forma unor timpi ce reprezinta durata de compilare/verificare a fisierelor sau timpul necesar obtinerii odata a rezultatului 'Satisfiabil' si odata a cazului in care formula nu este satisfiabila.

Algoritmul fiind unul simplu in comparatie cu optimizari moderne, timpul de demonstrare ca o formula este nesatisfiabila ajunge sa fie considerabil de mare chiar si pentru date de intrare cu dimensiuni relativ mici, motiv pentru care sunt necesare imbunatatiri ale algoritmului pentru a ajunge la o eficienta comparabila.

Compilatorul a avut nevoie de aproximativ 500 de secunde pentru verificarea fisierelor '.fst' si extragerea fisierelor '.ml' specifice limbajului Ocaml.

Nume fisier	Durata verificare/compilare
DataTypes	8 secunde
DataTypeUtils	33 secunde
DpllPropagation	285 secunde
OccurenceMatrixUtils	91 secunde
Dpll	71 secunde
ConvertorToString	3 secunde
InputFileParser	4 secunde
Main	3 secunde

Analizand codul sursa, se observa ca fisierele ce contin mai multe linii de cod pentru procesarea datelor si mai putine ce necesita demonstrarea unor proprietati simple, au nevoie de mai putin timp pentru a se verifica si complila. In schimb, fisierele

[2]https://people.sc.fsu.edu/ jburkardt/data/cnf/cnf.html

care au foarte putine linii de cod ce contribuie la formarea rezultatului si ajung sa fie extrase in fisierele '.ml', dar multe proprietati mai complexe a caror demonstratie trebuie verificata, necesita mult mai mult timp de compilare.

Dispozitivul pe care s-a facut aceasta contorizare avea urmatoarele specificatii:

- cpu - intel-core i5-1035G1
- memorie ram - 16gb
- tip hard-disk stocare - ssd

2.6 Posibile optimizari

Multi 'SAT solveri' moderi implementeaza algoritmul CDCL (Conflict-Driven-Clause-Learning) [3] care, pe langa pasii din algoritmul DPLL, adauga un pas nou de invatare si eficientizeaza saltul inapoi in momentul in care se gaseste o solutie falsa.

Un alt rafinament pe langa imbunatatirea algoritmului implementat poate fi eficientizarea structurilor de date folosite, pentru a reduce complexitatea timp a operatiilor ce se repeta cel mai des.

O problema ce poate aparea si in implementarea algoritmul DPLL dar si cea a algoritmului mai complex CDCL, este o metoda/euristica ineficienta de a alege o noua variabila pentru a i se asigna o valoare, si insasi valoarea care sa i se asocieze. Un exemplu ar fi metoda VSIDS [4], care se bazeaza pe asocierea unui scor pentru fiecare variabila, scor care se schimba daca variabila apare intr-un conflict.

O alta optimizare se poate implementa prin paralelizarea calculelor si impartirea ramurilor/solutiilor ce trebuie explorate.

[3] https://www.cs.princeton.edu/ zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf
[4] https://mk.cs.msu.ru/images/1/1f/SAT_SMT_Vijay_Ganesh_HVC2015.pdf

O mica imbunatatire din perspectiva limbajului de programare de aceasta data poate fi detectarea instructiunilor ce au rolul de a ajuta demonstrarea unor proprietati insa ajung pe parcursul dezvoltarii programului sa devina redundante, iar eliminarea lor ar putea reduce putin timpii de validare/compilare.

De asemenea, mai exista varianta de a simplifica problema pe care algoritmul trebuie sa o rezolve, iar analizand fiecare clauza in parte poate duce la identificarea unor clauze si literali redundanti, care pot fi eliminati din formula primita fara a schimba completitudinea si corectitudinea rezultatului final.

Capitolul 3

Pasi necesari pentru a reproduce

Informatii despre replicarea conditiilor necesare executiei programului scris folosind F* se pot gasi pe pagina de github a limbajului F*. [1]

In sectiunea urmatoare se prezinta pasii care au fost luati pentru crearea mediului in care s-a dezvoltat 'SAT solver-ul'. Instructiunile urmatoare sunt compatibile cu sistemul de operare Windows, verificat cu versiunile 10/11.

3.1 Programe si resurse necesare

Urmatoarele aplicatii/programe/resurse trebuie descarcate de la locatia specificata fiecareia in fisierul 'INSTALL.md' gasit pe pagina de github a limbajului FStar.

- OCaml - necesar compilarii si executarii fisierelor OCaml (.ml), care rezulta in urma compilarii fisierelor FStar (.fst) (versiune folosita - 4.14.0)
- opam - folosit pentru a instala pachetele necesare compilarii fisierelor specifice limbajului de programare OCaml

  (versiunea folosita pentru lucrare - 2.0.10)
- cygwin - ofera posibilitatea compilarii si executarii a programelor tipice sistemelor de operare Unix si Linux, ceea ce include suport pentru fisiere 'Makefile' (versiunea folosita pentru lucrare - 3.4.6)
- Z3 - folosit pentru a valida fisierele ce contin programe scrise folosind F*

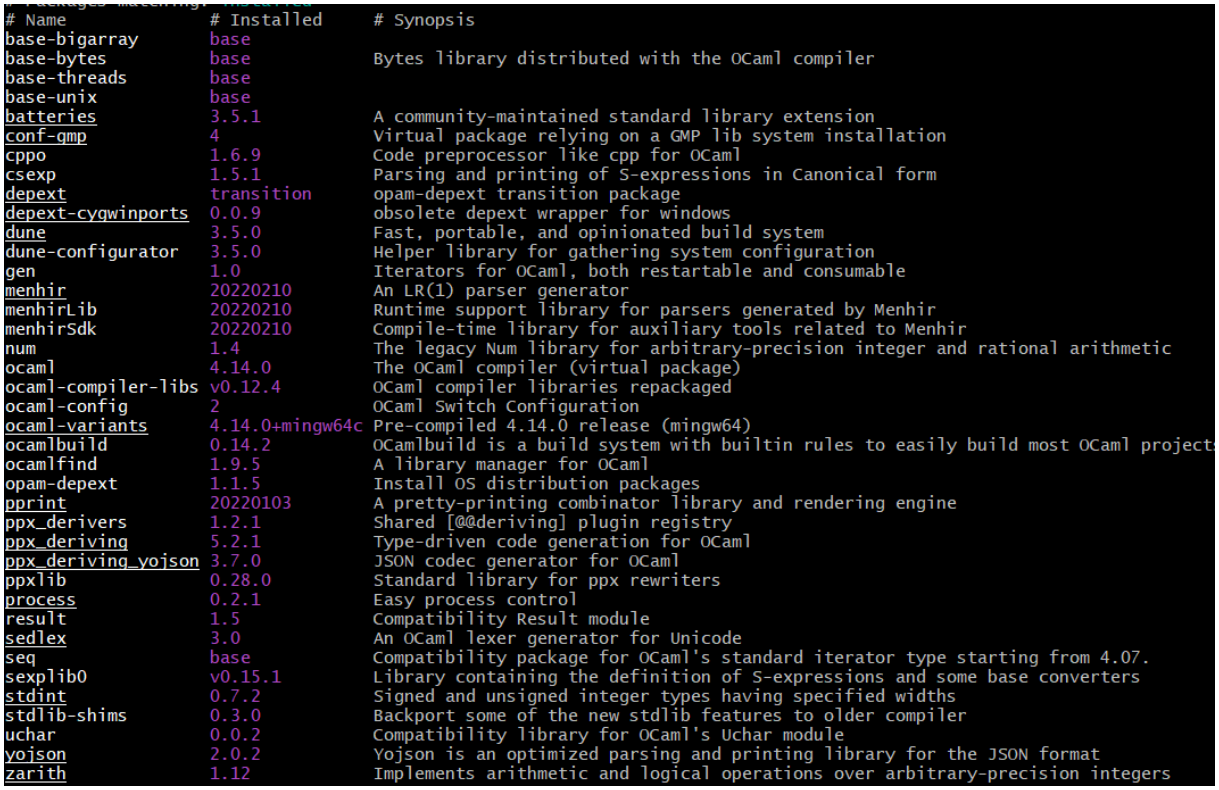
  (arhiva folosita pentru Windows - z3-4.8.5-x64-win.zip)

[1] https://github.com/FStarLang/FStar/blob/master/INSTALL.md

Dupa descarcarea/instalarea acestor resurse, trebuie clonata ramura 'master' a proiectului FStar pe dispozitivul local. (locatia clonei pe dispozitivul folosit pentru acest proiect: "D:/fstar", versiunea - F* 2023.04.26 dev)

Trebuie adaugate path-urile absolute catre ".../fstar/bin" si ".../z3-win/bin" in variabila 'Path' a sistemului.

Dupa instalarea programului 'opam', trebuie instalate mai anumite pachete de date. Minimul necesar de pachete se poate gasi si pe instructiunile de instalare gasite la link-ul de mai sus, insa pentru a avea la dispozitie toate resursele din proiectul FStar descarcat fara erori, sunt necesare urmatoarele pachete:



# Name	# Installed	# Synopsis
base-bigarray	base	
base-bytes	base	Bytes library distributed with the OCaml compiler
base-threads	base	
base-unix	base	
batteries	3.5.1	A community-maintained standard library extension
conf-gmp	4	Virtual package relying on a GMP lib system installation
cppo	1.6.9	Code preprocessor like cpp for OCaml
csexp	1.5.1	Parsing and printing of S-expressions in Canonical form
depext	transition	opam-depext transition package
depext-cygwinports	0.0.9	obsolete depext wrapper for windows
dune	3.5.0	Fast, portable, and opinionated build system
dune-configurator	3.5.0	Helper library for gathering system configuration
gen	1.0	Iterators for OCaml, both restartable and consumable
menhir	20220210	An LR(1) parser generator
menhirLib	20220210	Runtime support library for parsers generated by Menhir
menhirSdk	20220210	Compile-time library for auxiliary tools related to Menhir
num	1.4	The legacy Num library for arbitrary-precision integer and rational arithmetic
ocaml	4.14.0	The OCaml compiler (virtual package)
ocaml-compiler-libs	v0.12.4	OCaml compiler libraries repackaged
ocaml-config	2	OCaml Switch Configuration
ocaml-variants	4.14.0+mingw64c	Pre-compiled 4.14.0 release (mingw64)
ocamlbuild	0.14.2	OCamlbuild is a build system with builtin rules to easily build most OCaml project
ocamlfind	1.9.5	A library manager for OCaml
opam-depext	1.1.5	Install OS distribution packages
pprint	20220103	A pretty-printing combinator library and rendering engine
ppx_derivers	1.2.1	Shared [@@deriving] plugin registry
ppx_deriving	5.2.1	Type-driven code generation for OCaml
ppx_deriving_yojson	3.7.0	JSON codec generator for OCaml
ppxlib	0.28.0	Standard library for ppx rewriters
process	0.2.1	Easy process control
result	1.5	Compatibility Result module
sedlex	3.0	An OCaml lexer generator for Unicode
seq	base	Compatibility package for OCaml's standard iterator type starting from 4.07.
sexplib0	v0.15.1	Library containing the definition of S-expressions and some base converters
stdint	0.7.2	Signed and unsigned integer types having specified widths
stdlib-shims	0.3.0	Backport some of the new stdlib features to older compiler
uchar	0.0.2	Compatibility library for OCaml's Uchar module
yojson	2.0.2	Yojson is an optimized parsing and printing library for the JSON format
zarith	1.12	Implements arithmetic and logical operations over arbitrary-precision integers

La finalul acestor pasi, folosind terminalul Cygwin si instructiunile de tipul 'make' in folder-ul 'fstar', ar trebui sa functioneze verificarea si executarea oricaror fisiere surse scrise in F*, fisiere proprii sau exemple ce faceau deja parte din proiect.

3.2 Executarea solver-ului SAT

Sursele corespunzatoare proiectului prezentat se gasesc la: FStar-DPLL github.

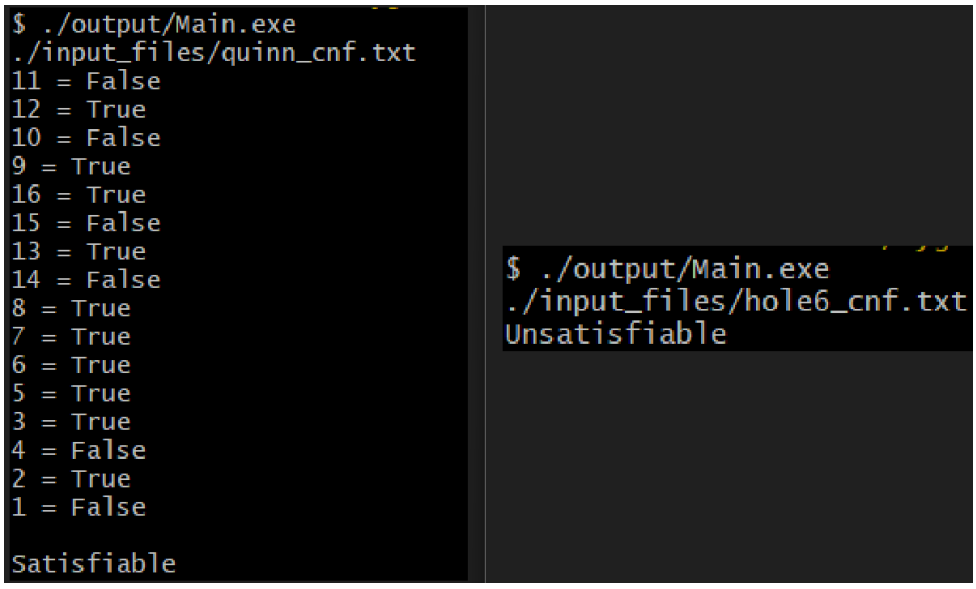
Aceste surse trebuie descarcate, salvate intr-un folder in proiectul 'fstar'. Fisierul 'Makefile' trebuie modificat, astfel incat variabila 'FSTAR-HOME' sa faca referire folder-ul 'fstar'. Acelasi pas trebuie facut pentru fisieru 'Makefile' din folder-ul 'output'.

Apoi, in terminalul cygwin deschis in folder-ul proiectului DPLL-FStar, trebuie executata comanda 'make', la finalul careia in folder-ul 'output' vor aparea pentru fiecare fisier sursa '.fst' cate un fisier '.ml' care contin codul Ocaml extras din sursele FStar. De asemenea in folder-ul 'output' se va afla executabilul "Main.exe".

Pentru a recompila si regenera fisierul "Main.exe", trebuie sters cel anterior creat, daca a fost creat.

Imediat dupa pornirea programului "Main.exe", trebuie introdus de la tastatura calea relativa catre un fisier de input. Cateva fisiere de input exista in folder-ul "input-files" si orice alt fisier de intrare trebuie sa respecte acea structura pentru ca parsarea datelor implementata sa functioneze.

La finalul unei astfel de executii, va aparea mesaj la consola cu rezultatul obtinut, fie ca formula data este nesatisfiabila, fie ca e satisfiabila si alaturi o varianta de raspuns ce contine variabilele formulei si valorile lor booleene astfel incat fiecare clauza a formulei sa aibe valoarea de adevar true.



Concluzii

Prin urmare, desi acest 'solver' nu aplica multe tehnici de optimizare sau cele mai eficiente structuri de date, prezinta cum se poate folosi sistemul de verificare formala si demonstrare a limbajului FStar pentru algoritmul DPLL. Astfel toate specificatiile ce corespund verificarii algoritmului DPLL pot functiona ca o baza pentru specificatiile oricarei extensii al metodei DPLL, precum CDCL sau alte variante optimizate.

Solver-ul este dovedit a fi complet, corect, garantat ca produce un rezultat, iar generalizarea modulelor si interactiunii dintre ele poate insemna ca este un bun punct de plecare pentru crearea unui solver mai eficient folosind limbajul FStar, aplicand imbunatatiri precum cele prezentate pe scurt anterior in aceasta lucrare.

Bibliografie

- despre problema SAT:
  https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem
- tutorial fstar:
  https://www.fstar-lang.org/tutorial/
- github fstar:
  https://github.com/FStarLang/FStar
- despre 'sat solveri':
  https://www.borealisai.com/research-blogs/tutorial-9-sat-solvers-i-introduction-and-applications/
- istoric al problemei SAT, descrierea problemei, algoritmului DPLL, si unor extensii ale sale :
  Book: Handbook of Satisfiability Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
  http://gauss.ececs.uc.edu/Courses/c626/notes/history.pdf
- fisiere de intrare: https://people.sc.fsu.edu/ jburkardt/data/cnf/cnf.html
- sursele proiectului:
  https://github.com/alex4482/FStar-DPLL-licenta/tree/main/dpll_optimized