

SATISFIABILITATEA FORMULELOR PROPOZIȚIONALE

Anghel Costel Junior, Covercă Elena-Daniela, Coroian Dan-Marius, and
Bogdan Simina

West University of Timisoara

Rezumat Această lucrare analizează implementarea și performanța MiniSat, un solver SAT pentru determinarea satisfiabilității formulelor booleene. Studiul se concentrează pe implementarea algoritmului CDCL, împreună cu structurile de date necesare. Evaluarea a fost realizată pe două familii de benchmark-uri, acestea fiind: tseitin-formulas și equivalence-chain-principle, cu timpi de execuție diferiți, contribuind la înțelegerea unui solver SAT.

Keywords: Satisfiability · Minisat · Benchmarks

1 Introducere

1.1 Context

Problema satisfiabilității (SAT) urmărește să determine dacă variabilele unei formule booleene pot primi valori astfel încât formula să fie adevărată. Această problemă are multiple aplicații practice în verificarea formală și optimizarea sistemelor.

Minisat, care este folosit pentru a rezolva probleme de satisfiabilitate booleană, a câștigat mai multe competiții SAT și este utilizat pe scară largă. Acesta s-a dovedit capabil să gestioneze probleme complexe ce implică foarte multe variabile [5]. Arhitectura sa, dezvoltată de Niklas Eén și Niklas Sörensson, reprezintă un punct de referință pentru implementarea soluțiilor SAT. Algoritmul se bazează pe formula logică în Formă Normal Conjunctivă (CNF), transformată sub format DIMACS, furnizată de către utilizator prin interfața linie de comandă, fie prin intermediul unui fișier, fie introdus direct de la tastatură, așa cum se poate observa și în Figura 1. Sistemul aplică tehnica CDCL (Conflict-Driven Clause Learning) pentru a determina dacă formula primită este satisfiabilă sau nu. Astfel, în funcție de rezultatul obținut, se identifică două cazuri: dacă formula este satisfiabilă, atunci sistemul afișează valoarea SAT și furnizează și un model care satisface formula; dacă formula este nesatisfiabilă, sistemul returnează UNSAT către utilizator.

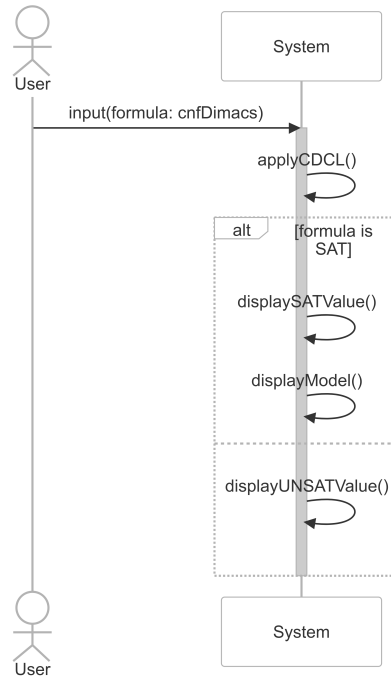


Figura 1. Diagramă de secvențe MiniSat

MiniSat este, în mare parte, ușor de utilizat. Utilizatorul oferă solver-ului un fișier ce specifică clauzele în forma normală conjunctivă (CNF), iar acesta afișează informații despre satisfiabilitate: fie indică faptul că formula este satisfiabilă, împreună cu o atribuire a variabilelor care o satisface, fie indică faptul că este nesatisfiabilă, când formula nu poate fi satisfăcută. De asemenea, solver-ul afișează informații despre procesul intern, cum ar fi aspecte ale căutării (DPLL). MiniSat folosește două tehnici principale: propagarea unităților, care determină rapid consecințele alegerilor făcute, și învățarea din conflicte, care ajută la evitarea repetării aceluiași greșeli [4].

Prin analiza implementării acestuia, putem înțelege mai bine cum funcționează un solver SAT și cum pot fi îmbunătățiți algoritmi existenți. De asemenea, prin documentarea procesului de instalare și evaluarea performanței folosind benchmark-uri vedem cum utilizăm practic acest instrument.

1.2 Scopul și obiectivele lucrării

Din punct de vedere tehnic, vrem să studiem structurile de date și algoritmul CDCL folosit în MiniSat, să identificăm posibile îmbunătățiri ale acestora și să documentăm implementarea acestora. De asemenea, vom analiza codul sursă pentru a înțelege structura internă și a propune îmbunătățiri.

Pentru partea practică, ne propunem să instalăm și să configurăm mediul de lucru, să rulăm o familie de benchmark-uri din competiția SAT 2024 cu un timp limită de 24 de ore și să măsurăm performanța obținută.

1.3 Instalare MiniSat

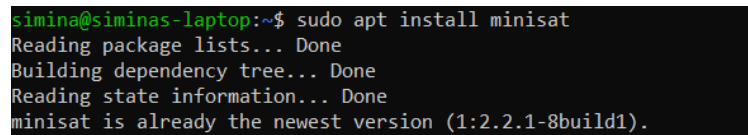
Pentru instalarea MiniSat pe un sistem de operare Windows, am ales să utilizăm Windows Subsystem for Linux. În primul rând, a fost necesară instalarea WSL. Pentru aceasta, am deschis PowerShell cu drepturi de administrator și am executat comanda:

```
1 wsl --install
```

După instalare, sistemul a necesitat o repornire pentru a finaliza configurarea. Acest proces instalează Ubuntu, care este suficient pentru necesitățile noastre.

După configurarea WSL și pornirea terminalului Ubuntu, am actualizat gestiunea de pachete, am instalat Minisat iar la final am verificat instalarea. Am făcut aceasta prin următoarele comenzi:

```
1 sudo apt update
2 sudo apt install minisat
3 minisat
```



```
simina@siminas-laptop:~$ sudo apt install minisat
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
minisat is already the newest version (1:2.2.1-8build1).
```

Figura 2. Instalarea MiniSat

După cum se poate observa în Figură, comenzile au fost executate cu succes, confirmând că MiniSat este instalat și funcțional în sistem.

2 Rezultate experimentale

2.1 Context

Pentru derularea testelor s-au folosit două familii de benchmark-uri preluate din Global Benchmark Database [3].

În primul rând, a fost rulată întreaga familie de benchmark-uri **tseitin-formulas**, fiecare test din aceasta având un timp de execuție limitat la 3 ore. Familia de benchmark-uri tseitin-formulas este derivată din transformarea Tseitin, aceasta convertind grafuri în formule propoziționale [2], fiind adesea nesatisfiabile, dar oferind benchmark-uri controlate și structurate, acestea utilizându-se adesea în competiții. Formulele Tseitin sunt dificile pentru minisat și pentru

solverele bazate pe CDCL datorită numărului mare de variabile și clauze, complexitatea problemei crescând exponențial. De asemenea acestea generează un număr mare de conflicte, solverele bazate pe CDCL întâmpinând dificultăți la navigarea prin spațiul soluțiilor. Transformarea Tseitin este folosită în conversii de formule logice în CNF (formă normală conjunctivă) prin introducerea de variabile auxiliare. Aceasta produce o formulă cu aceeași satisfiabilitate ca formula inițială, dar care nu este logic echivalentă. Pașii aferenți transformării sunt următorii:

- Crearea arborelui de parsare: formulele sunt reprezentate sub forma unui arbore, porțiilor logice asociindu-li-se o variabilă auxiliară.
- Adăugarea variabilelor auxiliare: pentru fiecare subformulă este introdusă o variabilă auxiliară iar rădăcina arborelui este declarată întotdeauna ca adevărată.
- Adăugarea constrângerilor variabilelor: se adaugă constrângeri care definesc relațiile între variabilele auxiliare și subformulele pe care le reprezintă.

În al doilea rând, a fost folosită familia **equivalence-chain-principle**, pentru fiecare dintre teste având timpul de execuție limitat la 6 ore. Ne sunt descrise două principii fundamentale, *FmlaEquivChain* și *FmlaImplyChain*, care generează tautologii bazate pe echivalențe și implicații iterabile între formule booleene și le transformă în sisteme de CNF-uri nesatisfiabile [2]. Aceste principii sunt folosite pentru a studia complexitatea demonstrațiilor Frege de adâncime fixă. Un sistem Frege este un sistem formal de demonstrație utilizat în logica matematică pentru a valida formule logice.

Un lanț de echivalență presupune că dacă formula T_i este adevărată, atunci T_{i+1} este echivalentă și adevărată (menținând valorile variabilelor la schimbări între nivele). Un lanț de implicații presupune că dacă T_i este adevărată, T_{i+1} este implicată logic. Complexitatea acestor lanțuri variază în funcție de:

- Parametrii formulelor:
 - **d**: Adâncimea formulelor T_i , reprezentând numărul de nivele de porți logice (\wedge și \vee) de la rădăcină până la frunzele arborelui logic.
 - **f**: Fanin-ul uniform al porților logice, adică numărul de intrări (variabile) pe care le poate primi fiecare poartă (\wedge sau \vee).
 - **n**: Numărul total de formule T_1, T_2, \dots, T_n din lanț, fiecare obținută prin aplicarea unor schimbări succesive asupra formulei precedente.
- Numărul de schimburi între noduri în formule consecutive ($T_i \rightarrow T_{i+1}$).

Așadar, complexitatea benchmark-urilor ce urmează a fi rulate poate varia în funcție de orice item enumerat anterior.

2.2 Interpretarea rezultatelor

Familia Tseitin-formulas Pentru a derula testele asupra întregii familii de benchmark-uri **Tseitin-formulas** a fost alocat un interval de timp de 24 de ore. Familia având 8 benchmark-uri, s-a acordat o perioadă de rulare de 3 ore

fiecărui benchmark. Un fapt demn de menționat este acela că fiecare benchmark din această familie are ca rezultat UNSAT. În urma rulării acestor teste s-a putut observa că în cadrul majorității, acestea atingeau rapid un progres cu o valoare sub 0.01% iar după aceea nu mai avansau, numărul de conflicte crescând vertiginos, excepție a făcut un singur test care a evoluat până la un progres de 14% după aceea rata de creștere scăzând drastic.

Tabela 1. Rezultate rulare benchmark-uri din familia tseitin-formulas.

ID Benchmark	Procentul Progresului	Număr de Conflicte	Rezultat Așteptat	Rezultat Obținut
0fa86232a40b5	14.618%	17,044,326	UNSAT	N/A
07b47f6c38150	0.009%	436,832,664	UNSAT	N/A
24dbf465b4b367	0.002%	291,221,704	UNSAT	N/A
36db206c7d947	0.001%	291,221,704	UNSAT	N/A
556cf803fbd507	0.001%	291,221,704	UNSAT	N/A
35789168b67a	0.000%	194,147,731	UNSAT	N/A
aa396dd8852d4	0.019%	291,221,704	UNSAT	N/A
c62915a6a9c088	0.002%	291,221,704	UNSAT	N/A

Așa cum se poate observa din tabelul de mai sus nici unul dintre benchmark-uri nu a reușit să ruleze până la final în timpul limită de 3 ore, iar un alt aspect notabil este acela că 5 din cele 8 teste au ajuns la același număr de conflicte înainte de finalizarea rulării, testul care a progresat cel mai mult având de asemenea și un număr semnificativ mai mic de conflicte comparativ cu restul.

Familia equivalence-chain-principle Am folosit un interval de timp de 24 de ore pentru a rula această familie, compusă din 4 benchmark-uri, fiecărui benchmark atribuindu-se câte 6 ore. Se aștepta valoarea UNSAT pentru toate cele 4 seturi de date. Două dintre ele au avut un progres foarte rapid, 10%, respectiv 15%, cel din urmă atingând și o finalitate. Pentru celelalte 3 benchmark-uri nu s-a obținut un rezultat final.

Tabela 2. Rezultate rulare benchmark-uri din familia equivalence-chain-principle.

ID Benchmark	Procentul Progresului	Număr de Conflicte	Rezultat Așteptat	Rezultat Obținut
8d2dfc50c175	0.561%	129,431,749	UNSAT	N/A
9cd3acdb6765	15.267%	3,134,121	UNSAT	UNSAT
63732c330adb	10.034%	17,044,326	UNSAT	N/A
f18c7c7b8666	0.652%	129,431,749	UNSAT	N/A

De menționat este că rezultatul celui de-al doilea set de date a fost obținut într-un timp de 671.383 secunde. S-a observat o dispunere mai uniformă a datelor față de celelalte benchmark-uri, clauzele fiind mai scurte, nedepășindu-se un maxim de 41 Literals/Clause la momentul progresului cu procentul 10%. În toate celelalte benchmark-uri, analiza a scalat foarte rapid din punctul de vedere al literalilor per clause, îngreunând astfel procesul de reducere al clauzelor și de

continuare al analizei în timpul acordat. Putem astfel concluziona că în cadrul acestui set de date, valorile parametrilor menționați la [Parametrii formulelor](#) au avut o valoare mică, ajungând la un rezultat final într-un timp eficient.

2.3 Analiza pas cu pas a unui exemplu dat

În formula pe care o vedem în Figura 2.3, se urmărește să vedem dacă aceasta este satisfiabilă sau nu. Pentru ca o formulă să fie satisfiabilă, toate clauzele din formă conjunctivă (CNF) trebuie să fie satisfăcute. Facem acest lucru atribuind valori variabilelor (x_1, x_2, \dots) astfel încât întreaga formulă să fie adevărată.

Formula este convertită în format DIMACS, deoarece acest format este utilizat de MiniSat pentru procesare. În format DIMACS, fiecare variabilă este reprezentată de un număr întreg pozitiv, iar negarea sa este reprezentată de acel număr precedat de semnul „-”. Fiecare clauză este scrisă pe o linie, iar finalul clauzei este indicat prin „0”.

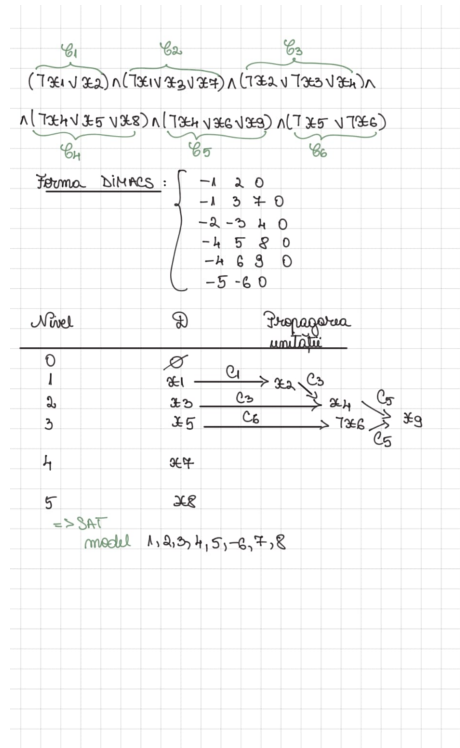


Figura 3. Aplicare în scris a algoritmului CDCL

Algoritmul CDCL gestionează procesul de satisfiabilitate a formulei trecând prin nivele de decizie și folosind propagarea unităților. **Propagarea unităților** se referă la procesul prin care, după atribuirea unei valori pentru o variabilă, determinăm automat valorile celorlalte variabile astfel încât clauzele să rămână adevărate. **Nivelele de decizie** indică numărul de decizii făcute până în acel moment.

La nivelul 0, nu există decizii sau propagări, deoarece nu avem nicio clauză unitară care să forțeze atribuirea unei valori adevărate. Algoritmul începe la nivelul 1, unde se face prima decizie: se atribuie o valoare variabilei x_1 . Aceasta declanșează propagarea lui x_2 . La nivelul 2, o altă decizie este luată pentru x_3 , ceea ce declanșează propagarea lui x_4 .

Acest proces continuă până la nivelul 5, unde ultima variabilă, x_8 , este atribuită. După această decizie toate clauzele sunt satisfăcute. Rezultatul este că formula este satisfiabilă (**SAT**), iar atribuirea variabilelor reprezintă modelul care face întreaga formulă adevărată.

2.4 Descrierea îmbunătățirii codului printr-un exemplu UNSAT

Pentru a facilita înțelegerea modului în care MiniSat abordează determinarea satisfiabilității formulelor propoziționale, dar și pentru monitorizarea dependenței dintre performanța acestuia și dificultatea formulei propoziționale, în funcție de numărul de clauze conflictuale și literali, s-a implementat în cod o metodă de extragere a informațiilor despre *UIP* (Unique Implication Point) și a clauzelor învățate în urma determinării grafului de implicație în procesul de analiză a conflictelor. Astfel, s-a putut observa cum MiniSat selectează și reține *First Unique Implication Point*, dar și felul în care clauzele învățate sunt gestionate la un interval de timp succesiv pentru a nu suprasolicita memoria sistemului. Aceste informații au fost observate prin rularea unui benchmark, ce avea ca date de intrare 54411 de literali și 419808 de clauze. În urma rulării, după o durată de aproximativ 30 de minute, rezultatul acestuia a fost UNSAT, așa cum se poate observa și în fișierul *benchmark_out.txt* din folderul *BenchmarksForTesting*. Clauzele conflictuale au fost de diferite dimensiuni, variind de la memorarea a 3 literali, până la un maxim de aproximativ 290, așa cum se poate observa și în fișierul *learnt_clauses.txt* din interiorul aceluiași director.

3 Structura MiniSat

Pentru a determina satisfiabilitatea formulelor propoziționale, din punct de vedere structural, proiectul MiniSat implementează o multitudine de structuri și algoritmi, mențiți să ruleze împreună în mediul proiectului.

3.1 LBool

Clasa LBool este definită în fișierul SolverTypes.h și are un singur atribut de tipul `uint8_t`, denumit *value*. Obiectele de tip Lbool sunt folosite pentru reprezentarea a trei stări:

- *l_True*: reprezintă starea *Adevărat* și are valoarea 0
- *l_False*: reprezintă starea *Fals* și are valoarea 1
- *l_Undef*: reprezintă starea *Nedefinit* și are valoarea 2

Clasa implementează 3 constructori, doi expliciti și unul default, utilizați în diferite cazuri de creare a obiectelor, după cum urmează:

- Un constructor explicit ce primește drept parametru o variabilă de tip `uint8_t`, unde atributul de tip `value` al obiectului `LBool` este inițializat corespunzător cu valoarea pasată ca parametru
- Un constructor implicit, default, fără parametri
- Un constructor explicit ce primește drept parametru o variabilă de tip `bool`, unde atributul de tip `value` al obiectului `LBool` este inițializat corespunzător în funcție de valoarea logică, 0 pentru *Adevărat* și 1 pentru *Fals*, conform mențiunilor anterioare.

Supraîncărcarea operatorilor Clasa `Lbool` supraîncarcă operatorii `==`, `!=`, `&&`, `||` și `^`

1. Supraîncărcarea operatorului `==`.
Metoda primește drept parametru un obiect de tip `Lbool` și returnează adevărat sau fals, în urma evaluării următoarei expresii:
 $((b.value \& 2) \& (value \& 2)) | (! (b.value \& 2) \& (value == b.value))$
Expresia verifică dacă obiectul curent și cel primit ca parametru sunt ambele nedefinite sau dacă obiectul primit ca parametru este definit și valoarea acestuia este egală cu valoarea obiectului curent. Astfel, în cazul în care cel puțin una dintre aceste condiții este îndeplinită, atunci metoda va returna `true`, sau `false` în caz contrar.
2. Supraîncărcarea operatorului `!=`.
Metoda primește drept parametru un obiect de tip `Lbool` și returnează adevărat sau fals, în urma evaluării expresiei: `!(*this == b)`. Astfel, metoda va returna `false`, dacă obiectul primit ca parametru este egal cu obiectul curent și `true` în caz contrar.
3. Supraîncărcarea operatorului `&&`.
Metoda primește drept parametru un obiect de tip `Lbool` și returnează rezultatul operației logice AND pentru obiecte de tip `Lbool`, utilizând operații pe biți.
4. Supraîncărcarea operatorului `||`.
Metoda primește drept parametru un obiect de tip `Lbool` și returnează rezultatul operației logice OR pentru obiecte de tip `Lbool`, utilizând operații pe biți.
5. Supraîncărcarea operatorului `^`.
Metoda primește drept parametru un obiect de tip `bool` și returnează rezultatul operației logice XOR, astfel: dacă valoarea primită ca parametru este `false`, valoarea operației XOR rămâne neschimbată, iar dacă valoarea primită ca parametru este `true`, atunci se schimbă valoarea de adevăr utilizând operații pe biți.

Funcții prietene Clasa `lbool` implementează două funcții prietene, `toInt`, respectiv `toLbool`, cu scopul de a converti în mod corespunzător un obiect de tip `lbool` într-un `int`, prin returnarea valorii, respectiv un obiect de tip `int` într-un obiect de tip `lbool`, utilizând constructorul aferent.

3.2 Lit

Structura de date `Lit` reprezintă abstractizarea literalilor din contextul satisfiabilității formulelor propoziționale. Aceasta este implementată în fișierul `SolverTypes.h` și are un singur atribut de tip `int`, denumită x , utilizată pentru stocarea internă a valorii literalului.

Din punct de vedere al constructorului, structura utilizează metoda `mkLit`, ce primește drept parametrii un obiect de tip `Var`, definit de un `int`, și un obiect de tip `bool`, care marchează prezența semnului și returnează un obiect `Lit` corespunzător. Se instanțiază un nou obiect `p`, de tip `Lit`, iar atributul său x este calculat după următoarea formulă: $p.x = \text{var} + \text{var} + (\text{int})\text{sign}$. Astfel, dacă semnul este false, adică literalul este pozitiv, valoarea acestuia va fi $p.x = 2 * \text{val}$, iar în cazul în care este negativ, acesta va fi $p.x = 2 * \text{val} + 1$.

Supraîncărcarea operatorilor Structura `Lit` supraîncarcă operatorii `==`, `!=` și `<`, `~` și `^` după cum urmează:

1. Supraîncărcarea operatorului `==`.
Metoda primește drept parametru un obiect de tip `Lit` și verifică dacă atributul său x este egal cu atributul x al obiectului curent, returnând `true` în caz afirmativ și `false` în caz contrar.
2. Supraîncărcarea operatorului `!=`.
Metoda primește drept parametru un obiect de tip `Lit` și verifică dacă atributul său x este diferit de atributul x al obiectului curent, returnând `true` în caz afirmativ sau `false` în caz contrar.
3. Supraîncărcarea operatorului `<`.
Metoda primește drept parametru un obiect de tip `Lit` și verifică dacă atributul x al obiectului curent este mai mic decât atributul x al obiectului primit ca parametru, returnând `true` în caz afirmativ, respectiv `false` în caz contrar.
4. Supraîncărcarea operatorului `~`.
Metoda primește ca parametru un obiect de tip `Lit` și returnează rezultatul negației pe biți al acestuia prin schimbarea semnului acestuia.
5. Supraîncărcarea operatorului `^`.
Metoda primește ca parametrii un obiect de tip `Lit` și un obiect de tip `bool` și returnează rezultatul operației logice XOR, convertind variabila `bool` la tipul *unsigned int* și utilizând operații pe biți.

Funcții ajutoare Structura de tip `Lit` implementează și diferite funcții ajutoare, precum funcții de tip `getter` pentru semnul și valoarea unui literal, cât și funcții de conversie explicită a datelor, precum conversia unui obiect de tip `Var`

în `int(toInt)`, conversia unui obiect de tip `Lit` în `int`, prin returnarea atributului `x` a parametrului pasat, cât și conversia în sens invers, a unui obiect de tip `int` într-un obiect de tip `Lit` (`toLit`).

Constante Se definesc de asemenea două obiecte de tip `Lit` constante, unul ce reprezintă literalul nedefinit și are atributul $x = -2$ și unul pentru cazurile de eroare, ce are atributul $x = -1$.

3.3 Clause

Clasa `Clause` din fișierul `SolverTypes.h` reprezintă o abstractizare a clauzelor din cadrul formulelor propoziționale în formă normal conjunctivă, fiind reprezentată de o disjuncție de literali. Clasa dispune de doi constructori expliți, parametrizați, dintre care unul de copiere, dar care nu pot fi utilizați în mod direct la crearea clauzelor, datorită faptului că nu alocă destulă memorie. Astfel, crearea clauzelor se realizează prin intermediul unei alte clase, mai exact *ClauseAllocator*, care este clasă prietenă a clasei `Clause`.

Funcții membre

- `calcAbstraction()` : metoda calculează o abstractizare aferentă fiecărei clauze la nivel de biți, utilizând literalii care alcătuiesc clauza
- `size()`: metoda returnează numărul de literali din alcătuirea clauzei
- `shrink(int i)`: metoda primește drept parametru un număr întreg `i` și reduce dimensiunea clauzei cu `i` literali
- `pop()`: metoda se folosește de metoda `shrink` pentru a elimina un literal din clauză
- `learn()`: metoda verifică dacă clauza este învățată
- `has_extra()`: metoda verifică dacă clauza are informații extra
- `const Lit& last()`: metoda returnează ultimul literal din alcătuirea clauzei
- `abstraction()`: returnează abstractizarea clauzei, utilizată în operațiile de subsumes.
- `subsumes(const Clause)`: verifică dacă clauza curentă satisface logic o altă clauză, sau dacă cealaltă clauză poate fi simplificată cu ajutorul primei clauze prin rezoluție. Aceasta returnează:
 - `lit_Error`: Clauza nu se poate simplifica sau satisface cu ajutorul clauzei curente
 - `lit_Undef`: Clauza este tautologie și devine redundantă în determinarea satisfiabilității
 - `Lit p`: Literalul ce se poate elimina din cea de a doua clauză.

Supraîncărcarea operatorilor

- `operator[]`: asigură accesul prin index la literalii care alcătuiesc clauza
- `strengthen(Lit)`: Elimină literalul primit ca parametru din clauza curentă și recalculază abstractizarea acesteia.

3.4 Solver

Clasa Solver din interiorul fișierului Solver.cc reprezintă fundamentul întregului proiect, atributele sale fiind responsabile de tehnicile și algoritmi specifici problemei satisfiabilității, precum propagarea și operațiile pe clauze, utilizând variabile și funcții specializate.

Atributele membre clasei

- var_decay: contor care prioritizează alegerea unei variabile într-un arbore de decizie, bazat pe conflictele în care este implicată.
- clause_decay: contor care prioritizează alegerea unei clauze, bazat pe conflictele în care este implicată.
- ccmn_mode: specifică gradul de simplificare a conflictelor utilizat de către solver
- garbage_frac: contorizează gradul de ocupare a memoriei pe baza căruia se activează garbage collector-ul.
- variabile care eficientizează procesul de căutare: random_var_freq, phase_saving, luby_restart, restart_first, restart_inc.

Conflict-Driven Clause Learning (CDCL) Algoritmul CDCL este un algoritm bazat pe DPLL, ce eficientizează determinarea satisfiabilității formulelor propoziționale în formă normal conjunctivă, utilizând backtracking non-cronologic(backjumping) și tehnici specifice de analizare a conflictelor, reprezentând mecanismul fundamental în MiniSat, fluxul său de execuție fiind ilustrat în Figura 4.

Așa cum se poate observa în diagrama de flux, algoritmul începe cu inițializarea solverului și extragerea variabilelor și a clauzelor aferente furnizate prin intermediul datelor de intrare. După acest pas, se verifică dacă există conflicte pe nivelul 0, iar dacă cel puțin un conflict este detectat, algoritmul se oprește și returnează UNSAT. Dacă nu există conflicte inițiale, algoritmul continuă printr-o propagare a unităților succesivă, cu ajutorul unei structuri repetitive de tip loop(for), verificând la fiecare pas dacă toate clauzele sunt satisfăcute. Dacă toate clauzele sunt satisfăcute și toate variabilele sunt atribuite, algoritmul returnează SAT și un model ce reprezintă soluția identificată pentru satisfacerea formulei propoziționale primite spre rezolvare. În cazul în care nu sunt satisfăcute toate clauzele la un anumit pas, se analizează conflictele întâmpinate și sistemul învață o nouă clauză, cu scopul de a evita repetarea conflictului la un pas ulterior, urmând de un backjumping, pentru a reveni pe nivelul care a inițiat conflictul, procedeele de propagare succesivă și analizare repetându-se fie până la determinarea modelului, fie până la epuizarea propagărilor și, prin urmare, detectarea unui conflict pe nivelul 0 de propagare.

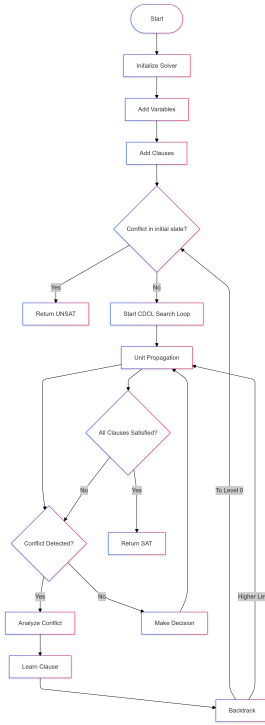


Figura 4. Diagrama de flux a algoritmului CDCL

Fluxul de execuție în MiniSat Fluxul de execuție în MiniSat este bazat pe CDCL și este reprezentat și în cadrul diagramei de secvențe la nivel de sistem ce se poate observa și în Figura 5, în cadrul căreia sunt detaliate interacțiunile dintre clasele principale ale sistemului și funcțiile care le utilizează. Se identifică astfel 3 mari categorii de componente ale sistemului: componente de tip boundary, componente de tip controller și componente de tip entity. Obiectul de tip boundary este componenta *User/Caller* ce reprezintă interfața în linie de comandă dintre utilizator și sistemul MiniSat, fiind separat de detaliile de implementare și procesare interne, oferind sistemului modularitate și abstractizare. Această componentă boundary interacționează cu funcțiile inițiale de creare și atribuire corespunzătoare structurilor necesare, precum *newVar()* și *addClause()*, ale căror apeluri sunt preluate de clasa controller, a cărei responsabilitate este gestionarea logicii interne. Controller-ul reprezintă obiectivul central din logica MiniSat, identificându-se prin funcțiile implementate de clasa *Solver*, care gestionează întregul proces de determinare a satisfiabilității formulei propoziționale pasată de către obiectul boundary, inclusiv inițializarea structurilor și variabilelor, dar, în special, a rulării algoritmului CDCL prin funcția *solve()*. Controller-ul comunică în mod direct cu obiectele de tip entity ale sistemului, precum *ClauseAllocator*, *Clause* și *Watchers*, intermediind accesul între utilizator și datele de implemen-

tare, asigurând rigurozitatea și robustețea fluxului de execuție. Clasa de tip entity, *ClauseAllocator*, se află în strânsă legătură cu clasa Solver, ea având rolul de a gestiona în mod eficient memoria aferentă unei clauze, reprezentată la nivel de sistem de clasa entity *Clause*, asociate formulei propoziționale, facilitând astfel accesul la acestea pe parcursul propagării sau a analizelor de conflicte de care este responsabil controller-ul, dar și adăugându-se corespunzător în liste tip *Watchers*. Clasa entity *Watchers* definește un mecanism de bază a solverului, prin intermediul căruia se urmărește în mod eficient starea clauzelor și a literalilor propagați. Structura sa internă este optimizată pentru a verifica dacă o anumită clauză poate sau nu să fie satisfăcută pe parcursul propagărilor succesive, actualizând datele pe măsură ce clauzele suferă modificări.

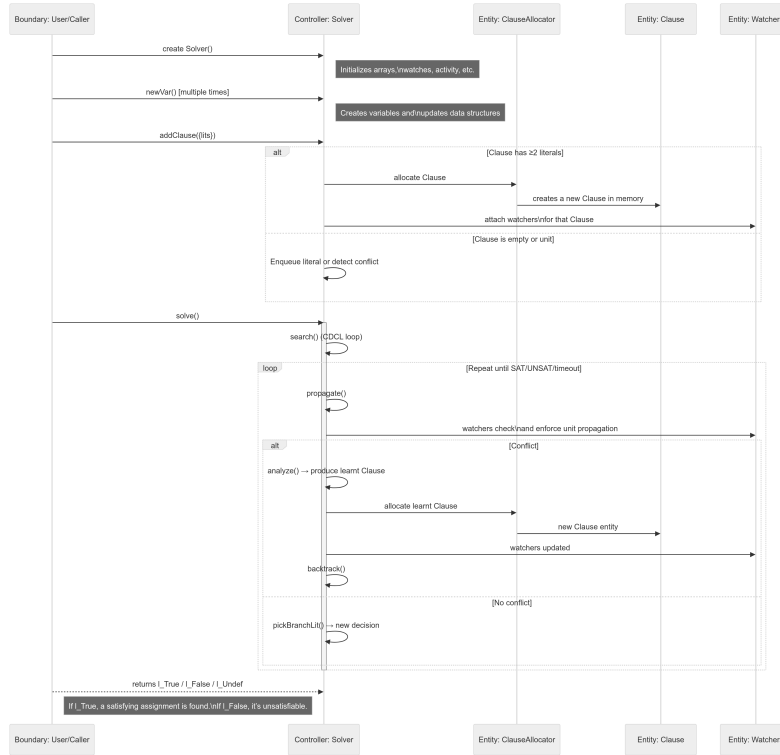


Figura 5. Diagrama de secvențe la nivelul sistemului a MiniSat

1. **Inițializarea datelor:** Funcția *solve()* a clasei Solver este responsabilă de inițializarea datelor și pornește algoritmul de atribuire a variabilelor în vederea identificării unui posibil model pentru formula propozițională în CNF (formă normal conjunctivă), sau a conflictului care determină nesatisfiabilitatea acesteia.

2. **Atribuirea recursivă:** Funcția *search()* a clasei Solver implementează parcurgerea iterativă prin care variabilele sunt atribuite valorile de adevăr variabilelor. Constrângerile sunt pe rând propagate, utilizând metoda *propagate()* până la identificarea unui conflict, moment în care acesta este pasat spre funcția *analyze()* pentru a identifica sursa acestuia, dacă nu ne aflăm pe primul nivel de decizie. În caz contrar, căutarea se oprește și formula este nesatisfiabilă.
3. **Propagarea:** Metoda *propagate()* este utilizată pentru a propaga constrângerile pentru clauzele aferente. Această iterează prin lista de literali și pentru fiecare dintre aceștia, iterează prin lista *watches* (clauze în care literalul apare negat) și se verifică dacă există un literal în aceste clauze, care este adevărat și ar putea reduce clauza la o tautologie. Dacă un literal din această listă *watches* devine fals, se încearcă înlocuirea sa cu un alt literal neatribuit din clauză. Dacă nu se poate înlocui, clauza rămâne unitară, iar literalul ori este adăugat în coada literalilor pentru propagare, ori se generează un conflict, așa cum este explicat și în [1].
4. **Analiza conflictelor:** Conflictele întâmpinate pe parcursul propagării sunt pasate către funcția *analyze()*, unde se identifică clauza ce trebuie reținută pentru a evita reproducerea conflictului în următoarele propagări, cât și nivelul pe care trebuie să se reîntoarcă utilizând tehnica de backjumping. Se parcurge graful de implicații care au cauzat conflictul, fiecare literal nou apărut contorizându-se, precum și nivelul de decizie pe care se află. La final, se formează o clauză care se memorează cu ajutorul literalilor și se determină cel mai mic nivel de pe care se poate relua propagarea.
5. **Backjumping:** Tehnica backtracking non-cronologică, cunoscută și ca backjumping, eficientizează căutarea și este implementată în MiniSat prin funcția *cancelUntil()* care resetează nivelul de decizie la nivelul de la care a pornit conflictul, pasat ca parametru.
6. **Euristici de decizie:** Odată cu trecerea la un alt nivel de decizie, funcția *pickBranchLit()* decide ce literal trebuie inițializat în continuare pe baza unei euristici caracteristici. În primul rând se încearcă alegerea unui literal aleatoriu, în funcție de polaritatea definită de către utilizator. Dacă acest lucru nu este însă posibil, se creează o structură de tip coadă cu literalii, prioritizați pe baza unui scor de activitate, care este determinat în mod direct de implicarea acestora în conflicte. Astfel, se alege literalul cu cel mai mare scor de prioritate și i se alege polaritatea corespunzător (adevărat sau fals).
7. **Gestionarea clauzelor învățate:** Pentru a eficientiza alocarea memoriei, cât și complexitatea algoritmului din punct de vedere al timpului, clauzele învățate sunt analizate, pentru a putea șterge clauzele care nu sunt sugestive algoritmului, cu ajutorul funcției *reduceDB()*, unde clauzele sunt sortate în funcție de conflictele în care sunt incluse și ținând cont de valoarea atributului *clause_decay*, clauzele neutilizate fiind eliminate.
8. **Simplificarea:** Funcția *simplify()* este responsabilă de simplificarea clauzelor, prin reducerea tautologiilor și a literalilor care nu influențează valoarea de adevăr a unei clauze prin propagarea constrângerilor aferente, actualiza-

rea listei de literali ce trebuie atribuiți, dar și a clauzelor învățate care sunt deja satisfăcute.

9. **Restart:** Pentru a evita blocarea algoritmului sau vizitarea unor ramuri de decizie prea costisitoare, algoritmul are implementat un mecanism de restart periodic, bazat pe atributele specifice menționate anterior. acesta apelându-se în funcție de numărul de conflicte întâmpinat pe parcursul propagărilor succesive.

3.5 Deosebiri între CDCL din MiniSat și CDCL aprofundat la curs

Deși MiniSat este un SAT solver ce se fundamentează pe algoritmul CDCL pentru determinarea satisfiabilității, acesta vine cu îmbunătățiri aduse tehnicilor consacrate, cu scopul de a eficientiza procesul de rezolvare.

O primă diferență notabilă față de tehnica CDCL clasică o reprezintă însuși procesul de propagare a literalilor, unde funcția implementată în MiniSat se bazează pe atribuirea și accesarea facilă a literalilor cu ajutorul unei liste de *watches*, care determină ordinea atribuire a acestora în funcție de clauzele pe care le compun. Spre deosebire de acesta, tehnica CDCL utilizează, în majoritatea cazurilor, ordinea arbitrară de atribuire a literalilor, lucru care poate determina fluctuații considerabile atât din punct de vedere al complexității timpului, cât și a memoriei programului.

O altă deosebire între cele două tehnici de propagare se poate observa și în euristicile de decizie utilizate, MiniSat implementând o euristică de decizie specifică, menită să eficientizeze procesul de căutare prin prioritizarea literalilor ce au apărut cel mai recent în clauze conflictuale, fiind considerate critice pentru atribuire. Acest lucru este posibil datorită structurii interne, fiecare literal având un atribut ce definește un scor de activitate, care se actualizează adecvat la fiecare implicare a acestuia în conflicte. Pentru a menține un domeniu de valori stabil și posibilitatea de schimbare a ordinii de atribuire în funcțiile de conflictele noi întâmpinate, scorul de activitate este decrementat periodic prin înmulțire. Deciziile sunt apoi efectuate alegând variabila neatribuită cu cea mai mare prioritate, caracterizând această euristică prin dinamicitate. Spre deosebire de Minisat, o euristică de decizie consacrată în procesul de verificare a satisfiabilității o reprezintă **Dynamic Largest Individual Sum**(DLIS) ce evaluează literalii în funcție de numărul de clauze nerezolvate pe care le poate satisface în funcție de valoarea de propagare a acestuia, alegându-se valoarea optimă identificată. Această metodă, deși de asemenea dinamică, poate deveni ineficientă pentru formule complexe din punct de vedere structural, întrucât este necesară repetarea întregului proces și recalcularea acestuia înainte de fiecare pas nou de propagare. Pe de altă parte, metoda **Jeroslow-Wang**, consacrată pentru tehnicile CDCL și DPLL, reprezintă o euristică statică, ce atribuie fiecărui literal o valoare calculată, bazată pe lungimea clauzelor pe care le alcătuiește, dată de următoarea formulă:

$$J(l) = \sum_{c \in \phi, l \in c} 2^{-|c|}$$

, unde:

- $|c|$ =numărul de literali dintr-o clauză
- ϕ =formula evaluată

Aceasta oferă o prioritate ridicată clauzelor mai scurte, calculul inițial reducând din capacitatea de a se adapta la informațiile aprofundate pe parcursul analizei conflictelor. Spre deosebire de Jeroslow-Wang, implementarea din MiniSat nu se limitează la valorile statice, ajustând prioritățile în funcție de natura dinamică a soluției, într-un mod mai eficient decât DLIS, așa cum se poate observa și în tabelul comparativ 3.

Tabela 3. Comparație între MiniSat, DLIS și Jeroslow-Wang.

Euristică	Tip	Complexitate Computațională	Adaptabilitate	Performanță în CDCL
MiniSat	Dinamică	Scăzută	Mare, bazată pe conflicte recente	Foarte bună
DLIS	Dinamică	Ridică	Limitată, recalculări complete	Ineficientă pentru formule mari
Jeroslow-Wang	Statică	Scăzută	Inexistentă, bazată pe priorități fixe	Suboptimală

4 Provocări întâmpinate

- Instalarea MiniSat a fost unul dintre primele obstacole pe care le-am întâmpinat. Deoarece folosim Windows, acest proces a necesitat instalarea subsistemului Windows pentru Linux (WSL), ceea ce a ridicat dificultăți legate de integrarea dintre Windows și Linux. Odată instalat WSL, a fost necesar să configurăm Ubuntu, un pas care s-a dovedit a fi dificil pentru cei dintre noi care erau mai puțin obișnuiți să lucreze în acest mediu.
- Înțelegerea codului MiniSat a fost un alt obstacol. Utilizarea structurilor de date precum LBool, Lit și Clause necesită cunoștințe de programare orientată pe obiect și tehnici complexe de manipulare a memoriei. Supraîncărcarea operatorilor și utilizarea funcțiilor prietene, implicate în propagarea literalilor și gestionarea conflictelor au necesitat un studiu aprofundat pentru a înțelege pe deplin contribuția lor la procedura algoritmică.
- Sincronizarea echipei a fost o altă provocare cu care ne-am confruntat. Deși fiecărui membru al echipei i s-a atribuit un rol specific, unele sarcini s-au suprapus, ceea ce a necesitat un efort suplimentar pentru a asigura eficiența. Am colaborat pentru a testa familii diferite de benchmark-uri, iar observațiile și concluziile obținute au trebuit să le reunim într-un singur raport.
- O altă provocare a fost afișarea clauzelor învățate atunci când benchmark-ul returna SAT. Algoritmul CDCL din MiniSat este optimizat pentru scenarii UNSAT, iar în cazul soluțiilor satisfăcătoare (SAT), clauzele învățate, deși sunt generate intern, nu sunt afișate în exterior utilizatorului.

5 Concluzii

În concluzie, în cadrul acestui proiect am reușit să realizăm un studiu amănunțit asupra funcționalităților MiniSat. Astfel, am început prin rularea de teste, utilizând benchmark-urile selectate de noi și prin analiza minuțioasă a codului sursă.

În urma acestor activități am reușit să obținem un nivel de înțelegere avansat asupra modului de funcționare și a limitărilor pe care acest proiect îl are iar unul dintre elementele pe care le-am considerat utile dar inexistente în versiunea de bază a proiectului Minisat a fost nivelul scăzut de informații pe care acesta ni-l furnizează în timpul rulării. În consecință, am ales să implementăm o metodă care extrage informații despre UIP(unique implication points) și despre clauzele învățate în urma determinării grafului de implicație, acestea fiind redirecționate către un fișier extern unde pot fi vizualizate.

6 Organigrama echipei

Anghel Costel Junior: Testarea familiei Equivalence-Chain-Principle, testare cazuri particulare de benchmarks, identificarea șabloanelor în rezultatele obținute, descriere rezultate, realizare bibliografie.

Coroian Dan Marius: Testarea familiei Tseitin-Formulas, identificare potențiale îmbunătățiri ale codului, descriere rezultate, dezvoltarea scriptului utilizat pentru rularea benchmark-urilor.

Covercă Elena Daniela: Analiza și identificarea claselor principale și a structurilor de date utilizate, analiza și identificarea algoritmilor specifici de determinare a satisfiabilității, documentarea informațiilor identificate.

Bogdan Simina Elisa: Realizarea rezumatului și a părții introductive, a procesului de instalare a MiniSat, realizarea diagramei UML pe baza codului sursă.

Bibliografie

1. Eén, N., Sörensson, N.: An extensible sat-solver [extended version 1.2]. SAT2003 (2003)
2. Heule, M.J., Iser, M., Jarvisalo, M., Suda, M.: Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions. N/A (2024)
3. Iser, M., Jabs, C.: Global Benchmark Database. In: Chakraborty, S., Jiang, J.H.R. (eds.) 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 305, pp. 18:1–18:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). <https://doi.org/10.4230/LIPIcs.SAT.2024.18>, <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2024.18>
4. din Illinois la Urbana-Champaign, U.: Vlsi cad: Logic to layout (2013), curs online disponibil la https://archive.org/details/academictorrents_ec1c86afefda42f4b36c34ae7b235ef0bfd6b9d3
5. Wheeler, D.: Minisat user guide: how to use the minisat sat solver. Wonderings of satgeek,[Online]. Available: <http://www.dwheeler.com/essays/minisat-user-guide.html> (2013)