ALEXANDRU IOAN CUZA UNIVERSITY OF IAȘI
FACULTY OF COMPUTER SCIENCE

BACHELOR'S THESIS

Proving SAT Solving Algorithms and Data Structures in Dafny

proposed by

Cezar-Constantin ANDRICI

July, 2019

Adviser

Assoc. Prof. Ştefan CIOBÂCĂ, PhD

ALEXANDRU IOAN CUZA UNIVERSITY OF IAȘI
FACULTY OF COMPUTER SCIENCE

Proving SAT Solving Algorithms and Data Structures in Dafny

Cezar-Constantin ANDRICI

July, 2019

Adviser

Assoc. Prof. Ştefan CIOBÂCĂ, PhD

Avizat,
Îndrumător lucrare de licență,
Conf. Dr. CIOBÂCĂ Ştefan.

Data: .... Semnătura: ....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **ANDRICI Cezar-Constantin** domiciliat în **România, jud. Iași, mun. Iași, strada Camille Flammarion, nr. 6**, născut la data de **06 august 1996**, identificat prin CNP **1960806226738**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2018, 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 **Proving SAT Solving Algorithms and Data Structures in Dafny** elaborată sub îndrumarea domnului **Conf. Dr. CIOBÂCĂ Ştefan**, 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 student:
....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Proving SAT Solving Algorithms and Data Structures in Dafny**, 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 **Cezar-Constantin ANDRICI**

Data: .... Semnătura: ....

Contents

1 Introduction

In the last few years, a lot of progress was done in making program verifiers easier to use, starting a big shift towards rewriting classic applications in a programming language designed with verification in mind, permitting the development of provably correct code. At the base of these projects there is a SAT solver, which is usually not automatically verified. With these new tools, it makes sense to employ them and try to use the existing paper proofs to construct a competitive, formally verified implementation of a SAT solving algorithm. There is a lot of interest in this topic because the SAT problem can model many practical applications such as electronic design automation, software and hardware verification, artificial intelligence, and operations research [6].

This thesis presents the results of proving actual implementations of two SAT solving algorithms and their associated data structures in Dafny. The algorithms proved in this thesis are: a basic backtracking search and a version of Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Both are implemented using recursion, but in the first one all the data structures are passed as arguments, while in the second, the data is kept as part of a class, to improve performance. DPLL is at the core of the industry standard Conflict-Drive Clause Learning (CDCL) algorithm, which is not presented in this thesis, but more information about what will mean to prove it by extending this project are in the section 5.1.

We chose Dafny [7] for this project, developed by Microsoft Research, because it seemed to be stable, to have good resources and a lively community, and to be simple and easy to use programming language, which later proved to be a great choice. Compared to other program verifiers, Dafny is straight forward to install, use, to extract code and to compile it. The syntax is very similar to other known modern languages and the annotations are very easy to write being comparable to the hand written ones.

The first ambition was to construct a proved competitive solver for the boolean satisfiability problem but, even if Dafny code can be extracted to C# and compiled, the performance is not comparable to a C version of the same code (more in Section 3.4). Still, making a proved solver in a slower language makes sense because a SAT solver should be sound, and this can not be proven for an unchecked code without some extra work. As a general solution, the certificates of unsatisfiability were introduced as standing as a proof for the instances that are not satisfiable, but these are hard to generate and cost a lot of space. A verified solver does not need to generate this kind of files if its annotations are strong enough. More about this in Section 2.1.

Even if program languages with proof verifiers/assistants became easier to use, there are still a lot of difficulties which have to be overcome to be able to create a big project. Most of them are related to the fact that imperative code that modifies global variables is much harder to prove than a pure function. We discuss more about the problems we have faced in the Section 4.

2 Background

2.1 Boolean satisfiability problem

The Boolean satisfiability problem (SAT) consists in deciding if a propositional logic formula is satisfiable. Usually, this is done by searching for a truth assignment under which the formula would evaluate to true, or by proving that no such assignment exists. Even if new concepts were introduced in the last decades, modern SAT solving algorithms are still based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6], which was described in 1962 [2].

One important property of a SAT solving algorithm is to be sound, which means that every time it returns that a formula is satisfiable or unsatisfiable, the result must be correct. If the formula is satisfiable, the algorithm will also return the truth assignment, which is easy to verify, but if it returns unsatisfiable then you have to trust the implementation. To overcome this trust issue, the certificates of unsatisfiability were introduced, which should be generated by the SAT Solver and contain information about the decision it made in such a way that could be verified. At the SAT Competition 2014, some runs produced proofs of over 100GB and a lot of participants in the following years showed no interest in providing resolution proofs because they said that were to complicated to produce and cost too much storage space[1].

In theory, a proved SAT solving algorithm does not need to generate a certificate for unsatisfiability, because the implementation will be trusted, but in reality, at least in this thesis, the algorithms were proved using an unproved programming language, which uses an unproved SMT solver, running on an unproved operating system. Of course, this situation will be overcome in the future because in the last years a lot of projects started that try to fix exactly these problems.

Most of the SAT solvers, including the algorithms proved in this thesis, require the formula to have a particular structure, mainly to be in conjunctive normal form (CNF). A formula is formed from variables, literals, and the operators AND (denoted by $\wedge$), OR ($\vee$), NOT ($\neg$). A boolean variable is usually represented with x_i and can take the values *true* or *false*. A literal is either a variable in which case it is called a positive literal, or is the negation of a variable, in which case it is called a negative literal and written as $\neg x_i$ (if $x_i = true$, then $\neg x_i$ evaluates to *false*). A clause is a disjunction of multiple literals having a form like $x_1 \vee \neg x_2 \vee x_3$. A formula in CNF is a conjunction of multiple clauses (an example: $(x_1) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$).

A simple way to define such an input is to write a clause on every line, and to represent the literals using positive/negative integers. This way, such an input is very easy to generate, read and parse. There are some competitions that evaluate the correctness and the performance of new approaches using hand made or industrial tests that use this format.

2.2 Similar Projects

There are related projects that try to prove SAT solving algorithms, most of them in Coq and Isabelle/HOL. One of the most recent ones is one initiative called IsaFoL (Isabelle Formalization of Logic), which proposes a formal framework for CDCL, offering a convenient way to prove metatheorems and experiment with variants. As they say, the

[1]SAT Competition 2018, Certified UNSAT, http://sat2018.forsyte.tuwien.ac.at/index.php?cat=certificates

objective is not to eliminate the paper proofs, but to bring rich formal companions to complement them. They developed this project in Isabelle/HOL, which is a functional programming with total correctness guarantees [1].

Our work seems to be the first that tries to prove imperative SAT solving algorithms.

2.3 Dafny

We chose the programming language Dafny, developed by Microsoft Research, because it comes with a program verifier which permits the programmer to annotate the code with propositions which can be verified. Dafny has a simple programming language to code in but, most importantly, the syntax for defining pre- and post-conditions, invariants and proofs of termination is very expressive and easy to use. Even when more complex proofs are needed, they can be created fairly easily by giving hints, using all kind of structures, or using reductio ad absurdum (which I used more than I like to admit). Using Dafny, the problem of writing bug-free code reduces to writing bug-free annotations.

3 Proving SAT Solving Algorithms and Data Structures

We propose the following example to better explain the following algorithms and to analyze the data structures.

Example 1. This formula has 7 variables and 5 clauses. It is easy to see that it is satisfiable for the truth assignment *(true, false, true, true, true, false, true).*

$$\begin{gathered}(x_1 \lor x_2 \lor x_3)\land\\ (\neg x_1 \lor \neg x_2)\land\\ (x_2 \lor \neg x_3)\land\\ (x_2 \lor x_4 \lor x_5)\land\\ (x_5 \lor x_6 \lor x_7)\end{gathered}$$

3.1 Concepts and Notations

These notations are chosen for consistency between the conventional notations and the syntax of Dafny.

Notation 1.

variablesCount	is the number of variables that must be assigned. Natural and bigger than 0;
variable	$0 \le \mathit{variable} < \mathit{variablesCount}$;
literal or ℓ	for a variable, the positive literal is $\mathit{variable} + 1$, and the negative literal is $-\mathit{variable} - 1$. This is because the literals are represented with positive and negative integers in the input;
clause	set of literals ordered ascending;
clauses or formula	set of clauses;
truthAssignment or tau or τ	$\mathit{truthAssignment}[\mathit{variable}]$ could be -1, 0 or 1. -1 means that the variable is not set, 0 means that is set to false, and 1 that is set to true;
$\mathit{tau}[v := \mathit{val}]$	variable v is set in the *truthAssignment* to the value *val* (returns a new assignment);
$\mathit{tau}[\ell := \mathit{val}]$	the corresponding variable for the literal ℓ is set to the necessary value so ℓ to be evaluated to *val* (returns a new assignment);
$\mathit{getLiteralValue}(\tau, \ell)$	The function returns the value of the literal ℓ in the truth assignment τ;
$\lvert x\rvert$	the number of elements of the sequence/array x;
$x.0$	Given a pair (x, y), $(x, y).0 = x$ and $(x, y).1 = y$.

We make use of the following predicates which we defined.

Predicate 1. $isClauseSatisfied(tau, clause)$: A clause is satisfied by a truth assignment *tau* if $\exists \ell \in \mathit{clause}$ s.t. $\mathit{getLiteralValue}(\mathit{tau}, \ell) = 1$.

Predicate 2. $isSatisfied(tau, clauses)$: A set of clauses is satisfied by a truth assignment *tau* if $\forall clause \in clauses, isClauseSatisfied(tau, clause)$.

Predicate 3. $isExtendingTau(tau, tau')$: For two truth assignments tau and tau', we say that tau' extends tau if $\forall i \bullet 0 \leq i < |tau| \wedge tau[i] \neq -1 \Longrightarrow tau[i] = tau'[i]$.

Predicate 4. $isTauComplete(tau)$: A truth assignment tau is complete if $\forall i \bullet 0 \leq i < |tau| \Longrightarrow tau[i] \neq -1$.

Predicate 5. $isSatisfiableExtend(tau, clauses)$: A set of clauses are satisfiable by extending a truth assignment tau if
$\exists tau' \bullet isTauComplete(tau') \wedge isExtendingTau(tau, tau') \wedge isSatisfied(tau', clauses)$.

Predicate 6. $isUnitClause(tau, clause)$: A clause is called a unit clause under a truth assignment tau if $\exists \ell \in clause \bullet getLiteralValue(tau, \ell) = -1 \wedge \forall \ell' \bullet \ell' \in clause \wedge \ell' \neq \ell \Longrightarrow getLiteralValue(tau, \ell') = 0$.

Predicate 7. $isEmptyClause(tau, clause)$: A clause is called empty under a truth assignment tau if $\forall \ell \in clause, getLiteralValue(tau, \ell) = 0$.

Predicate 8. $hasEmptyClause(tau, clauses)$: A set of clauses has an empty clause under a truth assignment tau if $\exists c \in clauses$ s.t. $isEmptyClause(tau, c)$.

3.2 Basic Backtrack Search

The simplest way to solve the SAT problem is using a basic backtracking search. We used this as as a chance to accommodate with the problem, Dafny and the pre- and post-conditions necessary for proving the correctness of the solution returned by the program.

At every step, the algorithm sets the next unset variable and if reaches in an UNSAT state, it backtracks. The implementation passes at every step all the data necessary as parameters.

The correctness of Algorithm 1 is given by the 2 post-conditions. The first one says that if a solution is returned, then the formula evaluates to true, which is easy to prove and easy to write. The second one says that if at the current step, the returned result is an empty sequence, then the formula would not be satisfied by extending the current truth assignment. This is done using the predicate $isFormulaSatisfiable$ which requires that all variables smaller than $step$ to be set and says that the current $truthAssignment$ is satisfiable, if $truthAssignment[step := 0]$ or $truthAssignment[step := 1]$ is satisfiable. So, to say that the current $truthAssignment$ is unsatisfiable, we have to know that both extensions are unsatisfiable. At the branching step, the next instruction checks if the formula under one of the two extensions has an empty clause and if it has, using Lemma 1, it returns that the extension is unsatisfiable.

Lemma 1. *If a formula has an empty clause under a truth assignment tau, then any truth assignment extending tau makes the formula false.*

Proof. We know that there is a clause c which is empty, that meaning all literals from c are evaluated to false under the current tau. We can assume that a complete tau' that extends tau and satisfies the formula exists, but the literals from c will evaluate the same under tau and tau', therefore, tau' violates the clause c, and does not satisfy the formula, resulting a contradiction with the assumption. □

```
predicate isFormulaSatisfiable(
    step: int, truthAssignment: seq<int>, clauses: seq< seq<int> >
)
    requires ∀ step' • 0 ≤ step' < step ⟹ truthAssignment[step'] ≠ -1;
    requires ∀ step' • step ≤ step' < |truthAssignment| ⟹ truthAssignment[step'] = -1;
{
    if step = |truthAssignment| then isSatisfied(truthAssignment, clauses)
    else isFormulaSatisfiable(step+1, truthAssignment[step := 0], clauses) ∨
        isFormulaSatisfiable(step+1, truthAssignment[step := 1], clauses)
}

method solve(
    step: int, truthAssignment: seq<int>, clauses: seq< seq<int> >
) returns (solution: seq<int>)
    requires ∀ step' • 0 ≤ step' < step ⟹ truthAssignment[step'] ≠ -1;
    requires ∀ step' • step ≤ step' < |truthAssignment| ⟹ truthAssignment[step'] = -1;

    ensures |solution| > 0 ⟹ isSatisfied(solution, clauses);
    ensures |solution| = 0 ⟹
        ¬isFormulaSatisfiable(step, truthAssignment, clauses);
{
    if (isSatisfied(truthAssignment, clauses)) {
        return truthAssignment;
    } else if (hasEmptyClauses(truthAssignment, clauses)) {
        return [];
    }
    solution := solve(step+1, truthAssignment[step := 0], clauses);
    if (|solution| = 0) {
        solution := solve(step+1, truthAssignment[step := 1], clauses);
    }
}
```

Algorithm 1: Basic Backtrack Search in Dafny

3.3 Davis-Putnam-Logemann-Loveland Algorithm

The DPLL procedure is more complex than backtracking because it prunes the search space by avoiding falsified clauses. A unit clauses has the property that all literals are evaluated to *false*, except one, which has no value yet. If this literal would be set to *false*, the clause will be violated by the current truth assignment, so we assign the literal to *true*. To do that, unit clauses are searched and set to *true* every time a new literal is set. This process is called unit propagation.

Another improvement to basic backtracking is to select an heuristic for choosing the next unset literal to be assigned. In the previous algorithm, we chose the first unset variable and tried both boolean values, but that is clearly inefficient because other choices might reduce the search space. A simple heuristic that we chose is to select the first unset literal that appears in the first unsatisfied clause. The idea is to try to satisfy all clauses as fast as possible. More complicated, but more efficient heuristics are presented in other papers, but we chose this one because is easier to implement. More about what will mean to change it can be found in Section 5.1.

A simple recursive DPLL procedure can be seen in Algorithm 2. To explain how it works, let us take Example 1, and look at Figure 1. First, the algorithm will chose the literal x_1 and will try to set it to *true*. At the next instruction, it will find that the second clause is unit and will set $\neg x_2$ to *true*, which will make the third clause unit, so $\neg x_3$ will be set to *true*. After the unit propagation, the next clause not satisfied yet is the fourth, and the first unset literal is x_4, which at the branching step it will first be assigned to *true*. Furthermore, only one clause is not satisfied yet, and the next decision will be to choose x_5 and try to set it to true, which will make the formula satisfiable, even if we have not yet chose values for x_6 and x_7.

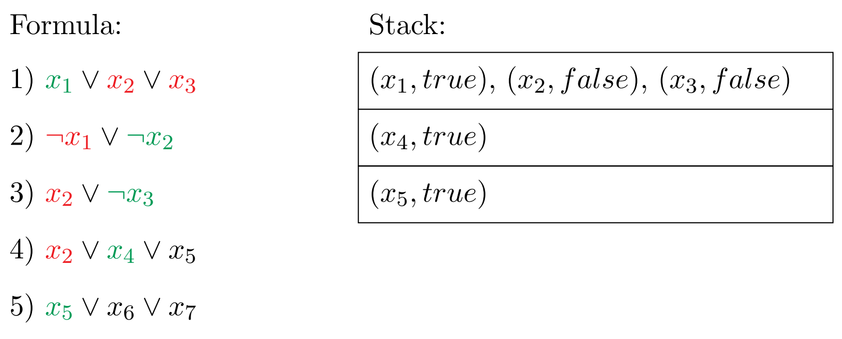
```
Function DPLL-recursive($F$, $\tau$)
  input  : A CNF formula $F$ and an partial assignment $\tau$
  output: UNSAT, or an assignment satisfying $F$
  while $\exists$ unit clause $\in F$ do
    $\ell \leftarrow$ the unset literal from the unit clause
    $\tau \leftarrow \tau[\ell := true]$
  if $F$ contains the empty clause then return $UNSAT$;
  if $F$ has no clauses left then
    Output $\tau$
    return $SAT$
  $\ell \leftarrow$ first unset literal that appears in the first not satisfied clause
  if $DPLL - recursive(F, \tau[\ell := true]) = SAT$ then return $SAT$;
  return $DPLL - recursive(F, \tau[\ell := false])$
```

Algorithm 2: Presented in Satisfiability solvers, 2017 [5], slightly modified in order to match our implementation.

3.3.1 Data Structures and their Invariants

A functional programming approach would have been more easier to prove because of the pure properties of the functions, but would mean more time lost allocating and freeing memory (and maybe the size of the heap would have been a problem too). Therefore, because we wanted a competitive SAT solver, we tried to use object orientated programming just enough to have a fast implementation. For example, the unit propagation is easy to implement using the scope of a pure function, because the memory heap will know what literals were set using this method, but to do that an imperative way, a new data structure was needed and a lot of new predicates, lemmas and variants were necessary to prove that it is correct.

We have 2 classes, *Stack* and *Formula*. For both of them and their data structures we formulated the conditions that hold before and after each step that modifies them.



Formula:
1) $x_1 \lor x_2 \lor x_3$
2) $\neg x_1 \lor \neg x_2$
3) $x_2 \lor \neg x_3$
4) $x_2 \lor x_4 \lor x_5$
5) $x_5 \lor x_6 \lor x_7$
Stack:
$(x_1, true)$, $(x_2, false)$, $(x_3, false)$
$(x_4, true)$
$(x_5, true)$

Figure 1: Stack representation for Example 1.

The Stack contains the trail of assignments made until the current state, divided into layers. A new layer is created at every branch step where a new unset variable v is chosen and set. The first element is $(v, < bool >)$ and the rest of the sequence in the layer contains assignments done after the unit propagation. In this way, every time the algorithm backtracks it knows exactly how many assignments to revert to reach the previous state. An instance of the stack is exemplified in Figure 1 for Example 1, where it can be seen that after the first iteration, the variable chose to be set was x_1, and the variables set by the unit propagation were x_2 and x_3.

Ghost constructs are used only during verification [4]. We use this feature for *contents*, which is a variable that makes easier to implement and prove various conditions about the content of the *stack*. It has exactly the same content as *stack*, but it is stored as a set where structure and order does not matter.

Stack is initialized only once by Formula, which uses it to maintain *truthAssignment* updated.

Data Structure 1: Stack

```
class Stack {
    var size : int;
    var stack : array< seq<(int, bool)> >;
    var variablesCount : int;
    ghost var contents : set<(int, bool)>;
}
```

Invariant 1. *Stack* contains assignments only on the good layers. $\forall i \bullet 0 <= i < size - 1 \implies |stack[i]| > 0 \wedge \forall i \bullet size <= i < |stack| \implies |stack[i]| == 0$.

Invariant 2. A variable appears only once in the stack.

```
    ∀ i,j  •  0 ≤ i < |stack| ∧  0 ≤ j < |stack[i]| ⟹
        (∀ i', j'  •  i < i' < |stack| ∧ 0 ≤ j' < |stack[i']| ⟹
            stack[i][j].0 ≠ stack[i'][j'].0)
        (∀ j'  •  j < j' < |stack[i]| ⟹
            stack[i][j].0 ≠ stack[i][j'].0)
      )
```

Invariant 3. Every assignment which occurs in the stack also occurs in the ghost var *contents*.

```
    (∀ i, j  •  0 ≤ i < |stack| ∧ 0 ≤ j < |stack[i]| ⟹
        stack[i][j] in contents)

    ∧

    (∀ c  •  c in contents ⟹
      ∃ i,j  •  0 ≤ i < stack.Length
           ∧ 0 ≤ j < |stack[i]|
           ∧ stack[i][j] = c)
```

The Formula is a tuple between $(variablesCount, clauses, stack)$. Based on these, we have created 5 more efficient data structures (see Data Structure 2) that contain the same information, which have the following invariants.

Data Structure 2: Formula

```
class Formula {
    var variablesCount : int;
    var clauses : seq< seq<int> >;
    var stack : Stack;

    var truthAssignment : array<int>; // from 0 to variablesCount - 1, values: -1, 0, 1

    var trueLiteralsCount : array<int>; // from 0 to |clauses| - 1
    var falseLiteralsCount : array<int>; // from 0 to |clauses| - 1

    var positiveLiteralsToClauses : array< seq<int> >; // from 0 to variablesCount - 1
    var negativeLiteralsToClauses : array< seq<int> >; // from 0 to variablesCount - 1
}
```

truthAssignment is an array that is indexed from 0 to $variablesCount - 1$, where $truthAssignment[v]$ means that for the current state the variable v has the value: -1 if unset, 0 if false, 1 if true. It is created based on the data structure *stack* and is updated every time *stack* is updated. Looking at Invariant 4, it is easy to see how simple *truthAssignment* is defined by using the ghost variable *stack.contents*. If were to define it based on the layers, a lot more universal quantifiers would have been needed.

Invariant 4. $validTruthAssignment()$

```
|truthAssignment| = variablesCount ∧

(∀ i • 0 ≤ i < |truthAssignment| ⟹ -1 ≤ truthAssignment[i] ≤ 1)

∧

(∀ i • 0 ≤ i < |truthAssignment| ∧ truthAssignment[i] ≠ -1 ⟹
  (i, truthAssignment[i]) in stack.contents)

∧

(∀ i • 0 ≤ i < |truthAssignment| ∧ truthAssignment[i] = -1 ⟹
  (i, false) ∉ stack.contents ∧ (i, true) ∉ stack.contents)
```

trueLiteralsCount and **falseLiteralsCount** are two arrays indexed from 0 to $|clauses| - 1$ where $trueLiteralsCount[i]$ denotes the number of literals set to true in the $clause_i$. These data structures are used to identify more quickly which clauses are satisfied, which clauses are unit or which clauses are empty. For example, to check if a $clause_i$ is satisfied, we only have to check if $trueLiteralsCount[i] > 0$. This is much faster than checking every literal every time, but Dafny needs help to understand that this is equivalent to $isClauseSatisfied(tau, clause_i)$. Therefore several lemmas are required to show that the data in these structures satisfy these properties.

Invariant 5. $validTrueLiteralsCount()$

```
|trueLiteralsCount| = |clauses| ∧
∀ i • 0 ≤ i < |clauses| ⟹
  0 ≤ trueLiteralsCount[i] = countTrueLiterals(truthAssignment, clauses[i])
```

Invariant 6. $validFalseLiteralsCount()$

```
|falseLiteralsCount| = |clauses| ∧
∀ i • 0 ≤ i < |clauses| ⟹
  0 ≤ falseLiteralsCount[i] = countFalseLiterals(truthAssignment, clauses[i])
```

The arrays **positiveLiteralsToClauses** and **negativeLiteralsToClauses** are indexed from 0 to $|clauses| - 1$ and $positiveLiteralsToClauses[i]$ contains every positive variable that appears in $clause_i$. These data structures are used every time a new literal is set/unset in order to update *trueLiteralsCount* and *falseLiteralsCount*, and to do unit propagation.

Invariant 7. $validPositiveLiteralsToClauses()$

```
|positiveLiteralsToClauses| = variablesCount ∧ (

    ∀ variable • 0 ≤ variable < |positiveLiteralsToClauses| ⟹
        ghost var s := positiveLiteralsToClauses[variable];
    valuesBoundedBy(s, 0, |clauses|) ∧ orderedAsc(s) ∧

    (∀ clauseIndex • clauseIndex in s ⟹
      variable+1 in clauses[clauseIndex]) ∧

    // the clauses which do not appear, do not contain the positive literal
    (∀ clauseIndex • 0 ≤ clauseIndex < |clauses| ∧ clauseIndex ∉ s ⟹
      variable+1 ∉ clauses[clauseIndex]))
```

Invariant 8. validNegativeLiteralsToClauses()

```
|negativeLiteralsToClauses| = variablesCount ∧

    ∀ v • 0 ≤ v < |negativeLiteralsToClauses| ⟹ (
    ghost var s := negativeLiteralsToClauses[variable];
    valuesBoundedBy(s, 0, |clauses|) ∧ orderedAsc(s) ∧

    (∀ clauseIndex • clauseIndex in s ⟹
      -variable-1 in clauses[clauseIndex]) ∧

    // the clauses which do not appear, do not contain the negative literal
    (∀ clauseIndex • 0 ≤ clauseIndex < |clauses| ∧ clauseIndex ∉ s ⟹
      -variable-1 ∉ clauses[clauseIndex]))
```

All of these predicates and some other more low-level predicates that we omit for brevity, are incorporated in a single predicate *valid()* which is used as an invariant at all steps. This way, it is guaranteed that the data structures will are consistent.

An issue in Dafny is that if you change anything in the heap, no matter how small, it has to recheck if all the predicates still hold, which sometimes can make the proof a lot longer. A lot of times, we try to prove that some data structure equals to its old version, so it could use what already knows about it. A way to avoid this is to use *modifies* as much as possible. However this is not very expressive. For example, it does not support stating that you modify only one position of an array, so it is necessary to prove that all positions contain the old/valid data.

Figure 2 shows the dependencies between the data structures. This means that the smallest change in one of them impacts what is known about all the other that depend on it. In the current implementation, *variablesCount* and *clauses* never change.

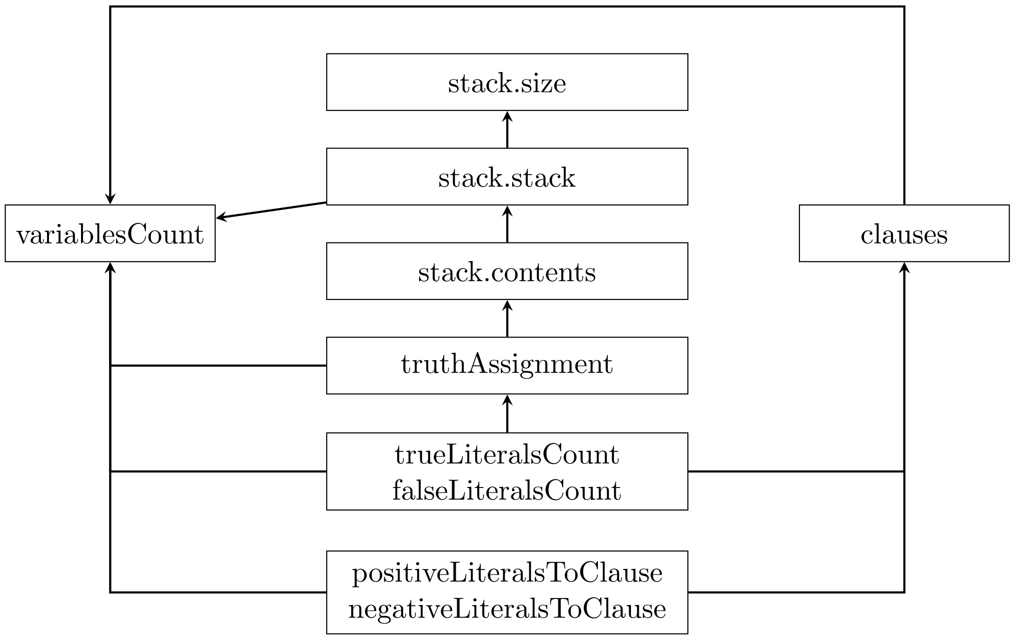


Figure 2: The dependencies between the data structures.

3.3.2 The correctness of the code

The code is considered correct if it goes only through valid states. From the initial valid state, we are able to do 4 actions: create a new layer on the stack, set a variable, set a literal and do unit propagation, and undo the last layer on the stack. Before analyzing why the code always returns the correct result, we have to define what these 4 methods do and how they modify the current state using post-conditions.

The method **newLayerOnStack()** just increments the size of the stack by one, but for that the following conditions must be met: the stack must not be full and the last layer must not be empty. The method guarantees that the new state is valid, and nothing changes except the size of the stack.

```
method newLayerOnStack()
    requires valid();
    requires 0 ≤ stack.size < |stack.stack|;
    requires stack.size > 0 ⟹ |stack.stack[stack.size-1]| > 0;

    modifies stack;

    ensures valid();
    ensures stack.size = old(stack.size) + 1;
    ensures 0 < stack.size ≤ |stack.stack|;
    ensures ∀ i • 0 ≤ i < |stack.stack| ⟹ stack.stack[i] = old(stack.stack[i]);
    ensures stack.contents = old(stack.contents);
```

The method **setVariable(variable : int, value : bool)** requires a variable that is not set in the current valid state and guarantees that only one position in the new truth assignment was changed: the position for *variable*. Figure 2 shows, because *stack.stack* and *truthAssignment* were changed, *trueLiteralsCount* and *falseLiteralsCount* had to be updated. We used *positiveLiteralsToClauses* and *negativeLiteralsToClauses* to efficiently update the two, and prove that the ones that are not contained in those are not impacted. The last ensure clause, is used later to prove that the program terminates, and this is true because the number of unset variables decreases at every branching step of the algorithm.

```
method setVariable(variable : int, value : bool)
    requires valid();
    requires 0 ≤ variable < variablesCount;
    requires truthAssignment[variable] = -1;
    requires 0 < stack.size ≤ |stack.stack|;

    modifies truthAssignment, stack, stack.stack, trueLiteralsCount,
             falseLiteralsCount;

    ensures valid();
    ensures stack.size = old(stack.size);
    ensures 0 < stack.size ≤ |stack.stack|;
    ensures |stack.stack[stack.size-1]| = |old(stack.stack[stack.size-1]| + 1;
    ensures stack.contents = old(stack.contents) + {(variable, value)};
    ensures ∀ i • 0 ≤ i < |stack.stack| ∧ i ≠ stack.size-1 ⟹
            stack.stack[i] = old(stack.stack[i]);
    ensures value = false ⟹ old(truthAssignment[variable := 0]) = truthAssignment;
    ensures value = true ⟹ old(truthAssignment[variable := 1]) = truthAssignment;
    ensures countUnsetVariables(truthAssignment) + 1 =
            countUnsetVariables(old(truthAssignment));
```

The method **setLiteral(literal : int, value : bool)** uses *setVariable*, so the pre- and post-conditions are similar, but the difference is that after it sets the first literal, it does unit propagation. This means that it calls *setLiteral* again with new values. So, at the end of a call, the *truthAssignment* might change at several positions. We discuss the last *ensures* clause in Section 3.3.3.

```
method setLiteral(literal : int, value : bool)
    requires valid();
    requires validLiteral(literal);
    requires getLiteralValue(truthAssignment, literal) = -1;
    requires 0 < stack.size ≤ |stack.stack|;

    modifies truthAssignment, stack, stack.stack, trueLiteralsCount,
             falseLiteralsCount;
    ensures valid();
    ensures 0 < stack.size ≤ |stack.stack|;
    ensures stack.size = old(stack.size);
    ensures |stack.stack[stack.size-1]| > 0;
    ensures ∀ i • 0 ≤ i < |stack.stack| ∧ i ≠ stack.size-1 ⟹
            stack.stack[i] = old(stack.stack[i]);
    ensures ∀ x • x in old(stack.contents) ⟹ x in stack.contents;
    ensures stack.contents = old(stack.contents) + stack.getLastLayer();
    ensures countUnsetVariables(truthAssignment) <
            old(countUnsetVariables(truthAssignment));
    ensures (
      ghost var (variable, val) := convertLVtoVI(literal, value);
      ghost var oldTau := old(truthAssignment[variable := val]);

      isSatisfiableExtend(oldTau) ⟺ isSatisfiableExtend(truthAssignment)
    );
    decreases countUnsetVariables(truthAssignment);
```

The method **undoLayerOnStack()** reverts the assignments from the last layer by changing the value of the literals to *unset*. As *setVariable*, this method needs a lot of proofs that confirm that the data structures are updated correctly and that the state is valid. For a fast update to *trueLiteralsCount* and *falseLiteralsCount*, we used *positiveLiteralsToClauses* and *negativeLiteralsToClauses*, and proved that the ones that are not in those remain unchanged.

```
  method undoLayerOnStack()
    requires valid();
    requires 0 < stack.size ≤ |stack.stack|;
    requires |stack.stack[stack.size-1]| > 0;

    modifies truthAssignment, stack, stack.stack, trueLiteralsCount,
             falseLiteralsCount;

    ensures valid();
    ensures stack.size = old(stack.size) - 1;
    ensures 0 ≤ stack.size < |stack.stack|;

    ensures ∀ i • 0 ≤ i < |stack.stack| ∧ i ≠ stack.size ⟹
            stack.stack[i] = old(stack.stack[i]);
    ensures |stack.stack[stack.size]| = 0;
    ensures ∀ x • x in old(stack.contents) ∧ x ∉ old(stack.stack[stack.size-1]) ⟹
            x in stack.contents;
    ensures ∀ x • x in old(stack.stack[stack.size-1]) ⟹ x ∉ stack.contents;
    ensures stack.contents = old(stack.contents) - old(stack.getLastLayer());
    ensures stack.size > 0 ⟹ |stack.stack[stack.size-1]| > 0;
```

3.3.3 The correctness of the result

The only function that has an *ensures* clause that states something about the result of the program is *setLiteral*$(\ell, value)$. Using the predicate $isSatisfiableExtend(tau)$, it states that the formula can be satisfiable under the returned truth assignment (let us denote it by *finalTau*) if and only if it can be satisfiable under the initial truth assignment with $\ell \leftarrow value$ (*tau*). The difference between *tau* and *finalTau* is that after the initial assignment is made, we do unit propagation. This means that they could be equal, or they could be different at multiple positions with a sequence of ‘if and only if‘ between them.

To do the unit propagation, we search in $negativeLiteralsToClauses[\ell]$ for unit clauses, and when we find one (ℓ'), we call *setLiteral* again to set it to *true*. Let us call the intermediate truth assignment tau'. The truth assignment tau' ranges from *tau* to *tauFinal*. Calling the method again returns that $isSatisfiableExtend(tau'[\ell' := true]) \iff isSatisfiableExtend(finalTau)$, and to prove the condition we want, we have to prove that $isSatisfiableExtend(tau') \iff isSatisfiableExtend(tau'[\ell' := true])$. If the formula could be satisfiable under *finalTau*, then is clear that it could be under tau' too, because the complete extensions of *finalTau* will apply to tau' as well. If the formula can not be satisfiable under $tau'[\ell' := true]$, then we know from Lemma 2 that $tau'[\ell' := false]$ does not satisfy at least a clause, and therefore, we can prove that the formula can not be satisfied by tau' with any extension by using Lemma 3.

Lemma 2. For a truth assignment tau and a unit clause c where the literal ℓ is not set, $tau[\ell := false]$ does not satisfy the formula.

Proof. This is easy to prove by assuming that a complete tau' that extends $tau[\ell := false]$ and satisfies the formula exists. But all literals in c evaluate to *false* under tau and therefore under tau' as well. The truth assignment tau' does not satisfy clause c, and therefore does not satisfy the formula either, resulting in a contradiction. □

Lemma 3. For a truth assignment tau and knowing that for a literal ℓ, $tau[\ell := false]$ and $tau[\ell := true]$ could not satisfy the formula if extended, then tau could not either.

Proof. Let us assume that tau if extended could satisfy the formula, therefore there exists a complete extension tau' that satisfies the formula. But $tau'[\ell]$ must be *true* or *false*, which makes it an extension of $tau[\ell := true/false]$ which can not be extend to satisfy the formula, resulting in a contradiction. □

The main method called to solve the SAT instance is *solve*. It implements the $DPLL-$*procedure* using recursion, but the data is kept in the instance of a class instead of being passed as arguments. The pre- and post-conditions of *solve* can be summed up by the following: it starts in a valid state and it ends up in the exact same state, and if it returns SAT then the current *truthAssignment* satisfies the formula, and if returns UNSAT it means the contrary. Starting and ending in the same state means that we chose to undo the changes we made even if we found a solution, and this is because otherwise we would have had to add a condition to every *ensures* clause with the type of the result, which would have doubled the number of post-conditions. For simplicity, we chose to revert to the initial state every time.

```
method solve() returns (result : SAT_UNSAT)
    requires valid();
    requires 0 ≤ formula.stack.size ≤ formula.stack.stack.Length;
    requires formula.stack.size > 0 ⟹
      |formula.stack.stack[formula.stack.size-1]| > 0;

    modifies formula.truthAssignment, formula.stack, formula.stack.stack,
             formula.trueLiteralsCount, formula.falseLiteralsCount;

    ensures valid();
    ensures old(formula.stack.size) = formula.stack.size;
    ensures ∀ i • 0 ≤ i < |formula.stack.stack| ⟹
      formula.stack.stack[i] = old(formula.stack.stack[i]);
    ensures old(formula.stack.contents) = formula.stack.contents;
    ensures formula.stack.size > 0 ⟹
      |formula.stack.stack[formula.stack.size-1]| > 0;

    ensures result.SAT? ⟹ formula.isSatisfiableExtend(formula.truthAssignment);
    ensures result.UNSAT? ⟹
      ¬formula.isSatisfiableExtend(formula.truthAssignment);

    decreases countUnsetVariables(formula.truthAssignment);
```

A flowchart [3] that shows every command in the *solve* method, and the propositions that hold after each line, is presented in Figure 3. For simplicity, when the initial state is reached, we used the notation *state* = *old*(*state*).

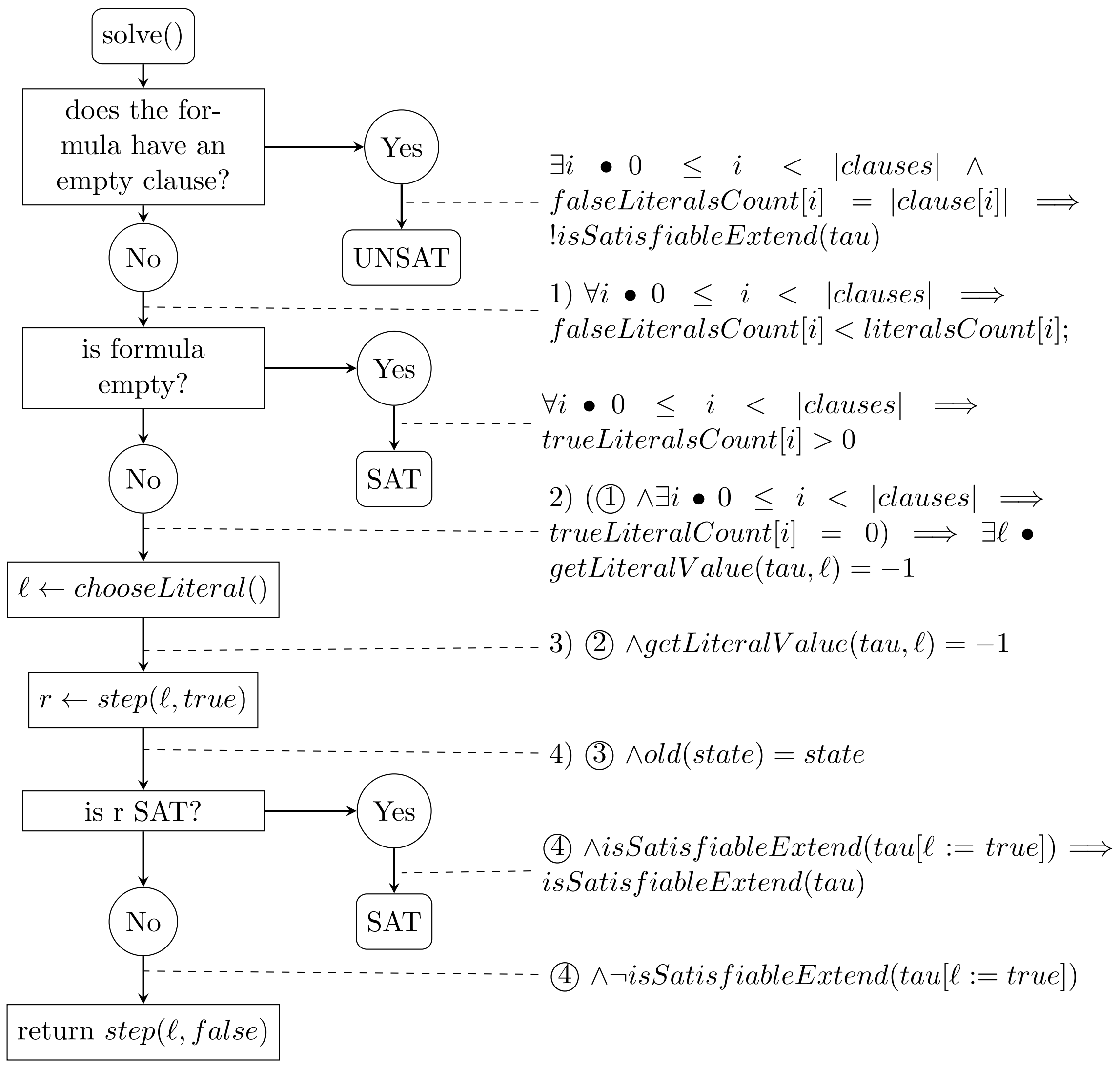

solve()
does the formula have an empty clause?
Yes
UNSAT
No
is formula empty?
Yes
SAT
No
$\ell \leftarrow chooseLiteral()$
$r \leftarrow step(\ell, true)$
is r SAT?
Yes
SAT
No
return $step(\ell, false)$
$\exists i \bullet 0 \leq i < |clauses| \wedge falseLiteralsCount[i] = |clause[i]| \implies !isSatisfiableExtend(tau)$
1) $\forall i \bullet 0 \leq i < |clauses| \implies falseLiteralsCount[i] < literalsCount[i];$
$\forall i \bullet 0 \leq i < |clauses| \implies trueLiteralsCount[i] > 0$
2) $(① \wedge \exists i \bullet 0 \leq i < |clauses| \implies trueLiteralCount[i] = 0) \implies \exists \ell \bullet getLiteralValue(tau, \ell) = -1$
3) $② \wedge getLiteralValue(tau, \ell) = -1$
4) $③ \wedge old(state) = state$
$④ \wedge isSatisfiableExtend(tau[\ell := true]) \implies isSatisfiableExtend(tau)$
$④ \wedge \neg isSatisfiableExtend(tau[\ell := true])$

Figure 3: Flowchart of method *solve*.

The modifications are extracted to the method $step(\ell, value)$ to be easier to prove that we modify the data structures and that at the end we revert the changes to reach to the initial state. Most of the pre- and post-conditions are exactly the same as the ones in *solve*, with small differences. First, *step* takes an unset literal and returns SAT if $isSatisfiableExtend(tau[\ell := value])$ and UNSAT if not. With these *ensures* clauses, *solve* can find a solution or prove using Lemma 3 that the current truth assignment could not be extended to satisfy the formula.

A flowchart [3] that shows every command in the step method, and the propositions that hold after each line is presented in Figure 4. For simplicity, when the initial state is reached, we used the notation $state = old(state)$. By putting together the methods described in Section 3.3.2 it is easy to see how they fit and how the proof is build.

```
method step(literal : int, value : bool) returns (result : SAT_UNSAT)
    requires valid();
    requires 0 ≤ formula.stack.size < |formula.stack.stack|;
    requires formula.stack.size > 0 ⟹
      |formula.stack.stack[formula.stack.size-1]| > 0;
    requires ¬formula.hasEmptyClause();
    requires ¬formula.isEmpty();
    requires formula.validLiteral(literal);
    requires formula.getLiteralValue(formula.truthAssignment, literal) = -1;
```

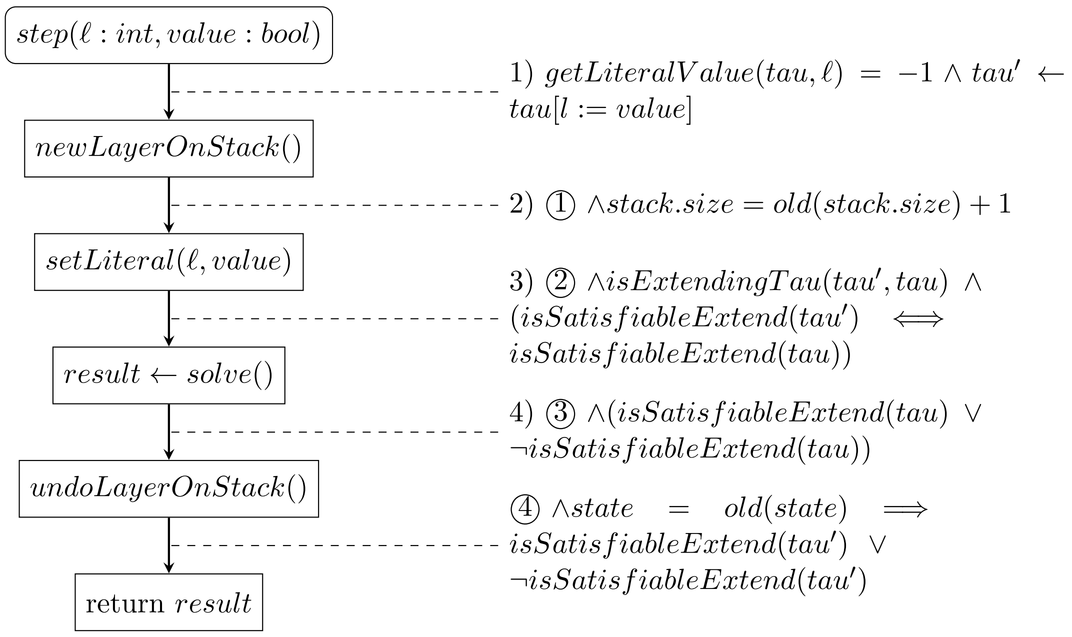

$step(\ell : int, value : bool)$
$newLayerOnStack()$
$setLiteral(\ell, value)$
$result \leftarrow solve()$
$undoLayerOnStack()$
return $result$
1) $getLiteralValue(tau, \ell) = -1 \land tau' \leftarrow tau[l := value]$
2) ① $\land stack.size = old(stack.size) + 1$
3) ② $\land isExtendingTau(tau', tau) \land (isSatisfiableExtend(tau') \iff isSatisfiableExtend(tau))$
4) ③ $\land(isSatisfiableExtend(tau) \lor \neg isSatisfiableExtend(tau))$
④ $\land state = old(state) \implies isSatisfiableExtend(tau') \lor \neg isSatisfiableExtend(tau')$

Figure 4: Flowchart of method step.

```
modifies formula.truthAssignment, formula.stack, formula.stack.stack,
         formula.trueLiteralsCount, formula.falseLiteralsCount;

ensures valid();
ensures old(formula.stack.size) = formula.stack.size;
ensures ∀ i • 0 ≤ i < |formula.stack.stack| ⟹
  formula.stack.stack[i] = old(formula.stack.stack[i]);
ensures old(formula.stack.contents) = formula.stack.contents;
ensures formula.stack.size > 0 ⟹
        |formula.stack.stack[formula.stack.size-1]| > 0;

ensures result.SAT? ⟹ (
  var (variable, val) := formula.convertLVtoVI(literal, value);
  var tau := formula.truthAssignment;

  formula.isSatisfiableExtend(tau[variable := val])
);
ensures result.UNSAT? ⟹ (
  var tau := formula.truthAssignment[..];
  var (variable, val) := formula.convertLVtoVI(literal, value);

  ¬formula.isSatisfiableExtend(tau[variable := val])
);
```

3.4 Benchmarks

Even if Dafny code can be extracted in C# and compiled, the performance does not compare to equivalent C code. Therefore, performance must come from better algorithms and better heuristic, which at the moment are not implemented. For example, the heuristic of choosing the order of the literals and their values can have a huge impact on the performance, and the one that we have implemented does a lot of unnecessary searches. We downloaded tests from the last SAT competitions and we ran them side by side against the state of the art SAT Solver MiniSat[2]. Our solver could not solve most of the tests in a limited time (600 seconds), but for the ones it did, had the same response as the other

[2]http://minisat.se/

SAT Solvers. This is good news for soundness, but shows that we have to incorporate better algorithms and data structures to the project to be competitive.

For experimenting, we restricted our tests only to the tests presented in Table 1, which were collected by Gregory J. Duck and published on his website[3]. The tests were ran on an Intel Core i5-8250U 3.40GHz with 8GB of RAM, Operating System 5.1.14-arch1-1-ARCH, Dafny 2.3.0.10506, GCC 9.1.0.

Test	Our verified Dafny SAT Solver	MiniSat v2.2.0
Hole6	UNSAT / 3.21s	UNSAT / 0.02s
Zebra	SAT / 1.09s	SAT / 0.00s
Hanoi4	timed out	SAT / 0.03s
Queens16	SAT / 4.64s	SAT / 0.00s

Table 1: Response and the time required to solve the tests

4 Difficulties in working with Dafny

As we said earlier, Dafny was a great choice for this project because is expressive and easy to use, but as the project got bigger and bigger, the prove time grew and the development time increased a lot. We discuss difficulties one by one:

The huge verification time for complex code is a challenging issue. Table 2 shows that the main methods take several minutes to be verified and this is after intense efforts to minimize it. For example, the *solve* method used to take over 1000 seconds to verify; therefore we split it in *step* and *solve*. Programming is challenging when a 5 lines of code method takes over 100 seconds to verify, and there is no built in way to know how could you help it to prove it faster. We tried the Axiom Profiler[4] from Viper Project, but it took too long (or it crashed) for the logs produced by Dafny for those methods. A first step to reduce time is to use two options when running Dafny which helped us: *-proc "*func_name*"*, which tells Dafny to check only one method/function, and *-errorLimit 1* (default is 5), which tells Dafny to stop after it finds the first error instead of the first 5. So, if the code contains more than a few methods, each one of them can be proved independently. Another useful option */trace*, which instructs Dafny to output the time required to prove each method and function. However, there is none to tell how much time each line takes. In big methods, with nested loops is hard to know which invariant is the hardest to prove, and therefore it is complicated to reduce the verifying time.

One strategy we tried was to work in an iterative manner when creating big methods, which means that we added some logic, and we tried to prove some initial properties (which usually is pretty fast) and step by step add more annotations to offer more context. But with every step, the things that should be proved increased and the time grew, because every time it tries to prove this new post-condition, it has to prove the others too. We think it will be useful to add a keyword like *focus* that tells Dafny to try to prove only that/those things and take everything else for granted. This will help on adding step by step more annotations, even if the verifying time in the end will be larger.

[3]https://www.comp.nus.edu.sg/ gregory/sat/

[4]https://bitbucket.org/viperproject/axiom-profiler

Function / Method / Lemma	Time (seconds)
Formula.setLiteral($\ell, value$)	630.59
SATSolver.solve()	332.01
Formula.setVariable($v, value$)	294.54
Formula.undoLayerOnStack()	249.16
SATSolver.step()	233.25
Formula.constructor($vC, clauses$)	91.14
Other 32	Less than 3 seconds each

Table 2: Time required to prove each method / lemma in seconds

Another issue is that **the proof/code ratio is large**. For example, the biggest method, *undoLayerOnStack*(), has 15 lines of pre- and post-conditions, its body contains 27 lines of code and 48 lines of proofs. Some of these are references to lemmas that add up to 232 more lines, without taking in consideration the definitions for the predicates used as invariants. So, for 27 lines of code, 280 lines of proofs are needed. The project has 2785 lines overall. A similar C implementation has about 600 lines.

When working with classes, another problem that becomes immediately obvious is that **it is not possible to use *old()* in lemmas**. When most of the data is kept on the instance, early on there would be a need to describe in the post-conditions some things using the *old* keyword. But because *old*(*exp*) means the value of the expression at the start of the method, the knowledge to get the answer can be passed to a lemma only by saving most of the initial state in ghost constructs and passing it as parameters (if it is possible), and by defining pre-conditions to capture the context in such a way to be useful. In practice, we wrote some of the proofs directly in the body of the method, because capturing the context would have been too complicated. This makes proofs in the methods bigger than theoretically necessary.

5 Conclusion

We developed a formally verified implementation of a SAT solving algorithm. The solver should be extended to be competitive. This is more challenging when using a program verifier, because every method needs to be proved, but we have shown that verifiers are powerful enough to help us to do that. The fact that Dafny supports writing imperative code and using object oriented programming is a great benefit for the competitiveness. SAT solving algorithms have been proved in Coq or Isabelle/HOL, which are functional languages. To our knowledge, our work is the first to prove imperative SAT solving algorithms.

Dafny was a great choice for this project, even if we faced some challenges, it was easy to use and had enough features to permit us to finish the proof of the DPLL algorithm without compromises. For our solver to be competitive, we have to implement and prove more optimizations.

5.1 Further work

The algorithm is currently slow, therefore a good start will be to change the heuristic which chooses the next literal to be set. This can be done without affecting the actual code, because the algorithm calls the method *chooseLiteral* which expects to be in a valid state and returns an unset literal. Additional data structures, or some pre-processing are needed to compute the order in which the literals will be selected, but these should not affect the existing proofs.

To further prove more complex algorithms, we have to find ways to reduce the verifying time of the main methods. It is challenging to continue to extend this project if it takes more than a few seconds to check the correctness of every change and the times presented in the previous section show that this is important.

To be competitive, we have to remember that a huge improvement in SAT solver development was gained when backtracking was replaced by the Conflict Driven Clause Learning algorithm [6]. Right now, the clauses are stored in sequences, which are immutable, and CDCL implies learning and forgetting clauses, therefore, arrays should be used instead. We expect that most of the code will be impacted by this. By looking at Figure 2, it is easy to see that adding and removing clauses means modifying most of the data structures and at every change, all predicates have to be proven again.

References

[1] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4786–4790, 2017.

[2] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[3] Robert W. Floyd. *Assigning meanings to programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[4] Richar L. Ford and K. Rustan M. Leino. Dafny Reference Manual. 08 2017.

[5] Carla P. Gomes, Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, pages 89–134. 2008.

[6] Filip Marić. Formalization and implementation of modern SAT solvers. *Journal of Automated Reasoning*, 43(1):81–119, Jun 2009.

[7] K. Rustan and M. Leino. Developing verified programs with Dafny. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, pages 82–82, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.