Verifying the DPLL Algorithm in Dafny

Cezar-Constantin Andrici
Alexandru Ioan Cuza University of Iaşi
cezar.andrici@gmail.com

Ştefan Ciobâcă
Alexandru Ioan Cuza University of Iaşi
stefan.ciobaca@info.uaic.ro

Modern high-performance SAT solvers quickly solve large satisfiability instances that occur in practice. If the instance is satisfiable, then the SAT solver can provide a witness which can be checked independently in the form of a satisfying truth assignment.

However, if the instance is unsatisfiable, the certificates could be exponentially large or the SAT solver might not be able to output certificates. The implementation of the SAT solver should then be trusted not to contain bugs. However, the data structures and algorithms implemented by a typical high-performance SAT solver are complex enough to allow for subtle programming errors.

To counter this issue, we build a verified SAT solver using the Dafny system. We discuss its implementation in the present article.

1 Introduction

Recent advances have enabled the formal verification and analysis of larger and larger software projects using techniques such static analysis or automated or interactive theorem proving. Two such examples are the certified C compiler CompCert [9] (using the Coq proof assistant) and the seL4 microkernel [8] (using Isabelle/HOL).

However, one problematic aspect is that verification tools themselves (e.g., static analyzers, SAT/SMT solvers, theorem provers, verification condition generators), which are usually highly sophisticated and relatively large pieces of software, might also include bugs. Such bugs can be rather embarassing, as they may lead the verification tool to prove correct a wrong program.

In fact, many verification tools contain bugs. For example, Brummayer and others [3] have shown using fuzz testing that many state-of-the-art SAT solvers contained bugs, including soundness bugs. Since 2016, in order to mitigate this issue, the anual SAT competiton requires solvers competing in the main track to output UNSAT certificates [1]; these certificates are independently checked in order to ensure soundness. Other contents such as the Automated Theorem Proving competition [14] or SMT competition [4] also contain various soundness check at various points in the competition timeline.

An approach to help ensuring correctness is to *verify the verification tools* themselves. In this article, we propose to do just that for a SAT solver. A SAT solver solves instances of the well-known Boolean satisfiability problem (SAT), which has many applications in software and hardware verification, as well as in combinatorial optimization. Relatively recently, high-performance SAT solvers based on the DPLL and CDCL algorithms have emerged and can handle practical SAT instances with millions of variables in reasonable running time.

However, SAT solver implementations contain complex data structures and algorithms and can therefore contain subtle bugs. This is less of an issue for satisfiable instances, since a satisfiability certificate can be checked easily. To counter this possible soundness issue, the SAT Competition started to require solvers to output certificates even in the case when the formula is unsatisfiable. Unsurprisingly, these certificates can be exponential and some tools cannot output certificates.

M. Marin, A. Crăciun (Eds.):
Working Formal Methods Symposium 2019 (FROM 2019)
EPTCS 303, 2019, pp. 3–15, doi:10.4204/EPTCS.303.1

In this paper, we propose to instead implement a *verified SAT solver* using the Dafny system. Dafny is a high-level imperative language with support for object oriented features. Dafny features methods with preconditions, postconditions and invariants which are checked at compilation time by relying on the Z3 SMT solver. If a postcondition cannot be established (either due to a timeout or due to the fact that it does not hold), compilation fails. Therefore, we can place a high degree of trust in a program verified using the Dafny system.

A modern, high-performance SAT solver essentially consists of the following optimizations over a backtracking approach to solving the satisfiability question: 1. unit propagation; 2. fast data structures to identify unit clauses; 3. variable ordering heuristics; 4. backjumping; 5. conflict analysis; 6. clause learning; 7. restart strategy.

The first three items are usually refered to as the DPLL algorithm, while the last four items make up the CDCL algorithm. We have implemented and verified the first two items, and the other points remain for future work. In particular, we use a fixed variable ordering. We note that our verification ensures soundness, completeness and termination of the solver. We do not verify input handling.

In Section 2, we discuss related work. In Section 3, we briefly explain the DPLL algorithm, as presented in the literature. In Section 4, we present the Dafny data structures and their invariants, as well as the implementation of the core algorithm, together with the verified guarantees that it provides. In Section 5, we briefly benchmark the performance of our solver and we conclude in Section 6.

2 Related Work

The SAT solver `versat` [12] was implemented and verified in the Guru programming language using dependent types. As our solver, it also implements efficient data structures. However, it relies on a translation to C where data structures are implemented imperatively by using reference counting and a statically enforced read/write discipline. Unlike our approach, the solver is only verified to be sound: if it produces an `UNSAT` answer, then the input formula truly is unsatisfible. However, termination and completeness (if the solver produces SAT, then the formula truly is satisfiable) are not verified. Another small difference is the verification guarantee: `versat` is verified to output `UNSAT` only if a resolution proof of the empty clause exists, while in our approach we use a semantic criterion: our solver always terminates and produces UNSAT only if there is no satisfying model of the input formula. Of course, in the case of propositional logic these criteriors are equivalent and therefore this difference is mostly a matter of implementation. Unlike our solver, some checks are not proved statically and must be checked dynamically, so they could be a source of incompleteness. An advantage of `versat` over our approach is that is implements more optimizations, like conflict analysis and clauses learning, which enable it to be competitive. Blanchette and others [2] present a certified SAT solving framework verified in the Isabelle/HOL proof assistant. The proof effort is part of the *Isabelle Formalization of Logic* project. The framework is based on refinment: at the highest level sit several calculi like CDCL and DPLL, which are formally proved. Depending on the strategy, the calculi are also shown to be terminating. The calculi are shown to be refined by a functional program. Finally, at the lowest level is an imperative implementation in Standard ML, which is shown to be a refinement of the functional implementation. Emphasis is also placed on meta-theoretical consideration. The final solver can still two orders of magnitude slower than a state-of-the-art C solver and therefore additional optimizations [5] are desirable. In contrast, in our own work we do not investigate any metatheoretical properties of the DPLL/CDCL frameworks; we simply concentrate on obtaining a verified SAT solver. We investigate to what extent directly proving the imperative algorithm is possible in an autoactive manner. We have shown that this is possible for

a restricted algorithm. However, we have reached a point where Dafny proofs take a lot of time (tens of minutes). In order to go further and verify the entire DPLL algorithm, additional techniques to bring down Dafny verification time are required, as dicussed in the conclusion. Another SAT solver formalized in Isabelle/HOL is by Marić [11]. In contrast to the previous formalization, the verification methodology is not based on refinement. Instead, a shallow embedding of the algorithm is expressed as a set of recursive functions. This style of algorithm is of course not as high-performance as an imperative one. Another formalization of a SAT solver (extended with linear arithmetic) is by Lescuyer [10], who verifies a DPLL-based decision procedure for propositional logic in Coq and exposes it as a reflexive tactic. Finally, a decision procedure based on DPLL is also verified by Shankar and Vaucher [13] in the PVS system. For the proof, they rely on subtyping and dependent types.

3 The Davis-Putnam-Logemann-Loveland Algorithm

The DPLL procedure is an optimization of backtracking. The main improvement is called unit propagation. A unit clause has the property that its literals are all *false* except one, which has no value yet. If this literal would be set to *false*, the clause would not be satisfied; therefore, the literal must necessarily be *true*. This process of identifying unit clauses and settings the unknown literal to true is called unit propagation.

Example 1. We consider a formula with 7 variables and 5 clauses:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_2 \vee x_4 \vee x_5) \wedge (x_5 \vee x_6 \vee x_7)$$

The formula is satisfiable, as witness by the truth assignment *(true, false, false, true, true, false, true)*.

Algorithm 1 describes the DPLL procedure which we implement and verify. We describe how the algorithm works on this example: first, the algorithm choses the literal x_1 and sets it to *true* (arbitrarily; if *true* would not work out, then the algorithm would backtrack here and try *false*). At the next step, it finds that the second clause is unit and sets $\neg x_2$ to *true*, which makes the third clause unit, so $\neg x_3$ is set to *true*. After unit propagation, the next clause not satisfied yet is the fourth one, and the first unset literal is x_4. At the branching step, x_4 is assigned to *true*. Furthermore, only one clause is not satisfied yet, and the next decision is to choose x_5 and set it to *true*, which makes the formula satisfied, even if x_6 and x_7 are not set yet.

4 A Verified Implementation of the DPLL Algorithm

In this section, we describe the structure of our verified solver.

4.1 Data Structures and Their Invariants

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

The attribute *variablesCount* is the number of variables in the formula. The array *stack* has size *variablesCount* and contains at most this many *layers*, which are explained below. A propositional variable is represented in the stack by a value between 0 and *variablesCount* $-$ 1. The attribute *size* represents the number of layers on the *stack*.

The *stack* contains the trail of assignments made up to the current state, divided into layers. A new layer is created at every branching step, where a new unset variable v is chosen and set. The first

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 1: Presented in Satisfiability solvers, 2017 [7], slightly modified in order to match our implementation.

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

Figure 1: Data Structure - Stack.

element is $(v, boolean\ value)$; the rest of the sequence in the layer contains assignments performed by 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 shown in Figure 2 for Example 1, where it can be seen that after the first iteration, the variable chosen 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 [6]. 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 order does not matter.

To represent class invariants, Dafny encourages a methodology of defining a **valid()** predicate, which is used as a pre-condition for all class members. In our case **valid()** is a conjunction of the following invariants.

Invariant 1. *Stack* contains assignments only on the used 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. Each variable occurs at most 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)
```

Formula:

1) $x_1 \vee x_2 \vee x_3$

2) $\neg x_1 \vee \neg x_2$

3) $x_2 \vee \neg x_3$

4) $x_2 \vee x_4 \vee x_5$

5) $x_5 \vee x_6 \vee x_7$

Stack:

$(x_1, true)$, $(x_2, false)$, $(x_3, false)$
$(x_4, true)$
$(x_5, true)$

Figure 2: Stack representation for Example 1.

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

  var truthAssignment : array<int>;

  var trueLiteralsCount : array<int>;
  var falseLiteralsCount : array<int>;

  var positiveLiteralsToClauses : array< seq<int> >;
  var negativeLiteralsToClauses : array< seq<int> >;
}
```

Figure 3: Data Structure - Formula.

A **Formula** is a tuple $(variablesCount, clauses, stack)$. Because a variable is represented by a value between 0 and $variablesCount - 1$, a positive literal is denoted in *clauses* by $variable + 1$ and a negative literal by $-variable - 1$. Based on the tuple, we have created 5 more efficient data structures that contain the same information, which have the following invariants:

The array **truthAssignment** 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 it would have been defined based on the layers, several additional universal quantifiers would have been needed. The function $getLiteralValue(tau, \ell)$ returns the value of the literal ℓ in the truth assignment *tau*. The invariant for **truthAssignment** is:

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)
```

The variables **trueLiteralsCount** and **falseLiteralsCount** are two arrays indexed from 0 to $|clauses| - 1$, where $trueLiteralsCount[i]$ denotes the number of literals set to true in $clause_i$ and $falseLiteralsCount[i]$

the number of false literals in $clause_i$. These are used to quickly identify which clauses are satisfied, which clauses are unit or which clauses are false. For example, to check whether a $clause_i$ is satisfied, we only evaluate $trueLiteralsCount[i] > 0$. The following invariants are true for these arrays:

Invariant 5. *validTrueLiteralsCount*() (analogously for *validFalseLiteralsCount*())

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

The arrays **positiveLiteralsToClauses** and **negativeLiteralsToClauses** are two arrays indexed from 0 to $variablesCount - 1$, where $positiveLiteralsToClauses[i]$ contains the indices of the clauses where a given variable occurs and $negativeLiteralsToClauses[i]$ the indices of the clauses where its negation occurs. 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. They satisfy the following invariants:

Invariant 6. *validPositiveLiteralsToClauses*() (analogously for *validNegativeLiteralsToClauses*())

```
|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]))
```

The conjunction of the above invariants, plus a few other low-level predicates that we omit for brevity, are incorporated in a single predicate *valid()* which is used as a data structure invariant for all methods. This way, it is guaranteed that the data structures are consistent.

4.2 Proof of the Data Structure Invariants

From the initial valid state, we can to do one of four 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. We show that these four methods preserve the data structure invariants above.

The method **newLayerOnStack()** increments the size of the stack by one, but it has the following preconditions: 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*. Because *stack.stack* and *truthAssignment* were changed, *trueLiteralsCount* and *falseLiteralsCount* have to be updated. We use *positiveLiteralsToClauses* and *negativeLiteralsToClauses*

to efficiently update them, and prove that the ones that are not contained in those are not impacted. To prove termination, we use as a variant the number of unset variables, which 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 also performs 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.

```
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 isSatisfiableExtend(oldTau[literal := value]) ⟺
              isSatisfiableExtend(truthAssignment);
    decreases countUnsetVariables(truthAssignment);
```

Finally, the method **undoLayerOnStack()** reverts the assignments from the last layer by changing the value of the literals to *unset*. As *setVariable*, this method needs several proofs that confirm that the data structures are updated correctly and that the state is valid. To quickly update *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;
```

4.3 Proof of Functional Correctness

The entry point called to solve the SAT instance is *solve*:

```
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);
```

It implements the *DPLL-procedure* using recursion, but the data structures are 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* can be extended to satisfy the formula, and if returns UNSAT it means that no truth assignment extending the current *truthAssignment* satisfies it. We made use of the following predicate:

Predicate 1. *isSatisfiableExtend*(*tau*, *clauses*): A set of clauses are satisfiable by a partial truth assignment *tau* if there exists a complete assignment that extends *tau* and that satisfies the formula.

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.

A flowchart that shows every command in the *solve* 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 $state = old(state)$.

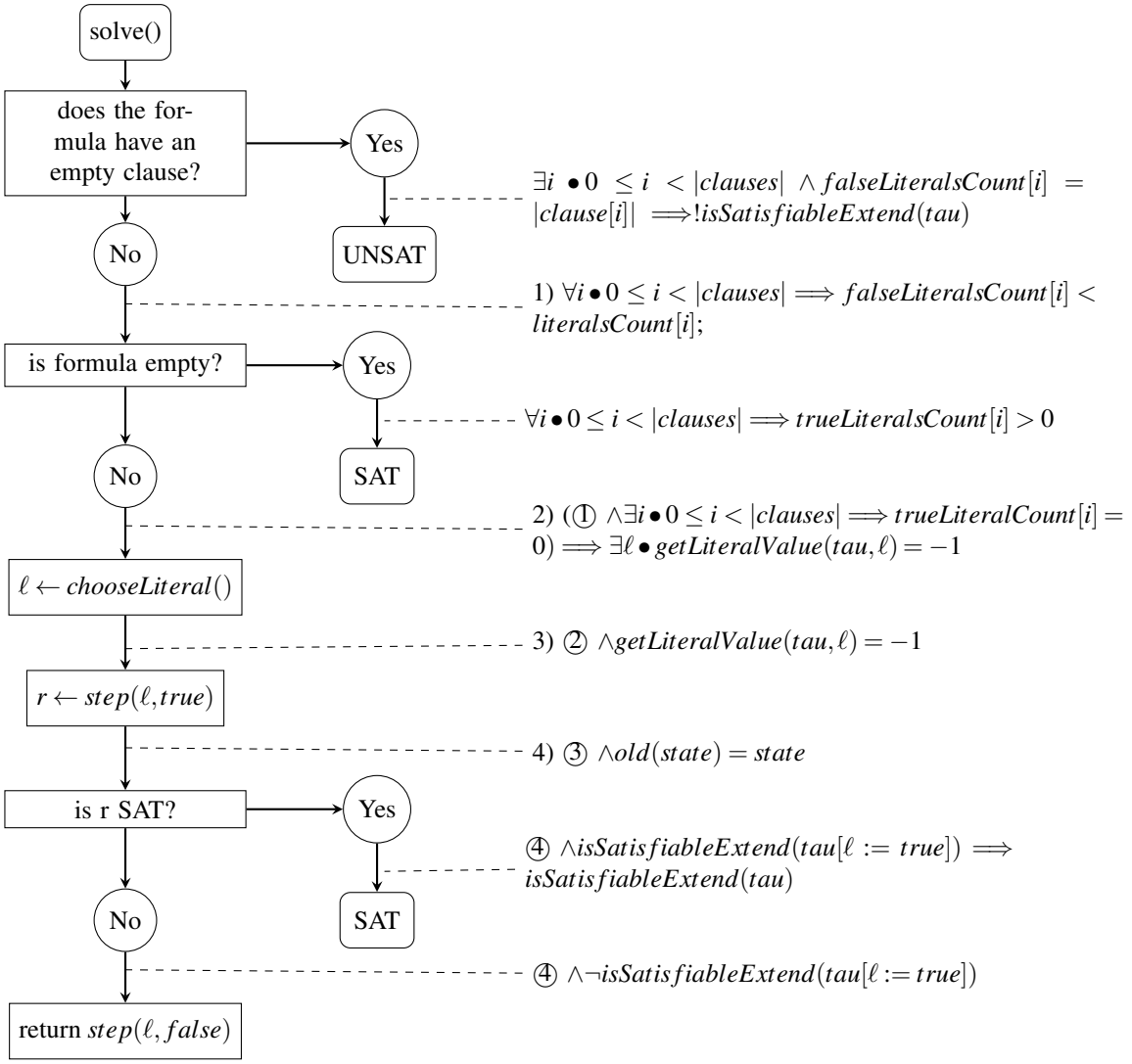

Figure 4: Flowchart of method *solve*.

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 2 that the current truth assignment could not be extended to satisfy the formula.

By putting together the methods described in Section 4.2 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;

    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[literal := value]);
    ensures result.UNSAT? ⟹
      ¬formula.isSatisfiableExtend(formula.truthAssignment[literal := value]);
{
  formula.newLayerOnStack();
  // stack.size = old(stack.size) + 1
  ghost var tau' := formula.truthAssignment[literal := value];
  formula.setLiteral(literal, value);
  // isSatisfiableExtend(tau') ⟺ isSatisfiableExtend(formula.truthAssignment)
  result := solve();
  // isSatisfiableExtend(formula.truthAssignment) ∨
  //  ¬isSatisfiableExtend(formula.truthAssignment)
  formula.undoLayerOnStack();
  // old(state) = state ∧ (isSatisfiableExtend(tau') ∨ ¬isSatisfiableExtend(tau'))
  return result;
}
```

The method *setLiteral*$(\ell, value)$ also ensures 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 *finalTau* also contain the assignments performed during unit propagation.

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 the unset literal to *true*.

We use the following two lemmas (formally verified by Dafny) to show that this is sound.

Lemma 1. 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. We assume 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 2. Given a truth assignment *tau*, if for a literal ℓ, *tau*$[\ell :=$ *false*$]$ and *tau*$[\ell :=$ *true*$]$ do not satisfy the formula when extended, then *tau* does not satisfy the formula 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. □

Setting a variable and performing unit propagation is separated as method *step*$(\ell, value)$, in order to be make the development more modular and therefore easier to prove. Most of the pre- and post-conditions are exactly the same as the ones in *solve*.

5 Benchmarks

Dafny code can be extracted to C# and compiled. We used a few tests from the latest SAT competitions and we ran them side by side against the SAT Solver MiniSat[1].

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[2]. 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 (number of variables, number of clauses)	Our verified Dafny SAT Solver	MiniSat v2.2.0
Hole6 (42, 133)	UNSAT / 3.21s	UNSAT / 0.02s
Zebra (155, 1135)	SAT / 1.09s	SAT / 0.00s
Hanoi4 (718, 4934)	timed out	SAT / 0.03s
Queens16 (256, 6336)	SAT / 4.64s	SAT / 0.00s

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

As expected, a CDCL solver outperforms our solver. However, the soundness guarantee offered by our verifier solver is higher than the unverified C code of MiniSat.

6 Conclusion and Further Work

We have developed a formally verified implementation of the DPLL algorithm in the Dafny programming language. Our implementation incorporates data structures to quickly identify unit clauses and perform unit propagation. However, it uses a fixed variable ordering.

The formalization consists of about 3200 lines of Dafny code. The project was developed in a year of part-time work, as part of the BsC thesis of the first author. The code was written by the first author, who also lerned Dafny during that time. The ratio between lines of proofs and code is 5 to 1. The function *undoLayerOnStack* has the biggest proof-to-code ratio: 27 lines of actual code and 280 lines of annotations. The entire Dafny implementation of the solver is available at `https://github.com/andricicezar/sat-solver-dafny`.

The solver is not currently competitive against state-of-the-art CDCL solvers, but since Dafny compiles to C#, we conjecture that it is possible in theory to obtain performance close to the state-of-the-art by implementating the rest of the optimizations present in CDCL. We base our conjecture on a quick test that shows that Dafny code for enumerating permutations is roughly as performant as hand-written C#code for the same task.

Development of the verified solver was challenging, since the verification time is prohibitive for certain methods. Here is a summary of the verification time required for the various methods implemented as part of the solver:

[1] `http://minisat.se/`

[2] `https://www.comp.nus.edu.sg/~gregory/sat/`

Function / Method / Lemma	Time (seconds)
Formula.setLiteral(ℓ, *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

As further work, the main priority is to lower the verification time. This will enable experimenting with additional SAT solving optimizations. Another possible extension is to port the data structures and algorithms, together with their invariants, to a verifier for C, such as VCC or Frama-C. This should give C-like performance to the verified SAT solver.

Acknowledgement. This work was supported by a grant of the *Alexandru Ioan Cuza University* of Iaşi within the Research Grants program *Grant UAIC*, code GI-UAIC-2018-07.

References

[1] Tomás Balyo, Marijn J. H. Heule & Matti Järvisalo (2017): *SAT Competition 2016: Recent Developments*. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pp. 5061–5063.

[2] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich & Christoph Weidenbach (2018): *A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality*. *J. Autom. Reasoning* 61(1-4), pp. 333–365, doi:10.1007/s10817-018-9455-7.

[3] Robert Brummayer, Florian Lonsing & Armin Biere (2010): *Automated Testing and Debugging of SAT and QBF Solvers*. In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pp. 44–57, doi:10.1007/978-3-642-14186-7_6.

[4] David R. Cok, David Déharbe & Tjark Weber (2014): *The 2014 SMT Competition*. *JSAT* 9, pp. 207–242.

[5] Mathias Fleury (2019): *Optimizing a Verified SAT Solver*. In: *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, pp. 148–165, doi:10.1007/978-3-030-20652-9_10.

[6] Richard L. Ford & K. Rustan M. Leino (2017): *Dafny Reference Manual*.

[7] Carla P. Gomes, Henry A. Kautz, Ashish Sabharwal & Bart Selman (2008): *Satisfiability Solvers*. In: *Handbook of Knowledge Representation*, Elsevier, pp. 89–134, doi:10.1016/S1574-6526(07)03002-7.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: formal verification of an OS kernel*. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pp. 207–220, doi:10.1145/1629575.1629596.

[9] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[10] Stephane Lescuyer (2011): *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Theses, Université Paris Sud - Paris XI.

[11] Filip Marić (2009): *Formalization and Implementation of Modern SAT Solvers*. *J. Autom. Reasoning* 43(1), pp. 81–119, doi:10.1007/s10817-009-9127-8.

[12] Duckki Oe, Aaron Stump, Corey Oliver & Kevin Clancy (2012): *versat: A Verified Modern SAT Solver*. In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pp. 363–378, doi:10.1007/978-3-642-27940-9_24.

[13] Natarajan Shankar & Marc Vaucher (2011): *The Mechanical Verification of a DPLL-Based Satisfiability Solver*. *Electr. Notes Theor. Comput. Sci.* 269, pp. 3–17, doi:10.1016/j.entcs.2011.03.002.

[14] Geoff Sutcliffe (2018): *The 9th IJCAR Automated Theorem Proving System Competition - CASC-J9*. *AI Commun.* 31(6), pp. 495–507, doi:10.3233/AIC-180773.