SAT Solver Verification Project*

Filip Marić and Predrag Janičić

Faculty of Mathematics, University of Belgrade,
Belgrade, Studentski Trg 16, Serbia
{filip, janicic}@matf.bg.ac.rs

Abstract. In this paper we give an overview of our SAT solver verification project. This is the first paper to present this project as a whole. We summarize the results achieved in the verification of SAT solvers described in terms of abstract state transition systems, in the Hoare-style verification of an imperative implementation of a modern SAT solver, and in generation of a trusted SAT solver based on the shallow embedding into HOL. Our formalization and verification are accompanied by a solver implemented in C++ and a trusted, automatically generated solver implemented in a functional language. One of the main final goals of our project is reaching to an both efficient and fully trusted SAT solver. Other goals include rigorous analyzes of existing SAT solving systems.

1 Introduction

One of the most important goals of computer science is reaching trusted software. This is especially important for algorithms and programs that have numerous applications, including SAT solvers — programs that test satisfiability of propositional formulae.

Spectacular improvements in the performance of SAT solvers have been achieved in the last several years and nowadays SAT solvers can decide satisfiability of propositional formulae with tens of thousands of variables and millions of clauses. However, this tremendous advance in the SAT solving technology has not been accompanied with corresponding theoretical results about solvers' correctness. Descriptions of new algorithms and techniques are usually given in terms of implementations, while correctness arguments are either not given or are given only in outlines. This gap between practical and theoretical progress needs to be filled and first steps in that direction have been made only recently, leading to the ultimate goal of having modern SAT solvers that are formally proved correct.

One approach for achieving a higher level of confidence in SAT solvers' results, successfully used in recent years, is *proof-checking*. In this approach, solvers are modified so that they output not only *sat* or *unsat* answers, but also evidences for their claims (models for satisfiable instances and proof-objects for unsatisfiable instances) which are then checked by independent proof-checkers.

* This work was partially supported by Serbian Ministry of Science grant 144030.

Proof-checking is relatively easy to implement, but it has some drawbacks. First, the evidence for every solved SAT instance has to be verified separately. Also, generating unsatisfiability proofs introduces some overhead to the solver's running time, proofs are typically large and may consume gigabytes of storage space, and proof-checking itself can be time consuming [Gel07]. Since proof-checkers have to be trusted, they must be very simple programs so they can be ,,verified" by code inspection. On the other hand, in order to be efficient, they must use specialized functionality of the underlying operating system which reduces the level of their reliability (e.g., the proof checker used in the SAT competitions uses Linux's `mmap` functionality [Gel07].)

Another approach is to verify a SAT solver itself, instead of checking each of the solver's claims. This approach is much harder to realize, since it requires formal analysis of the complete solver's behaviour. Still, we opt for this approach, as we believe that it is much more rewarding:

- Although the overheads of generating unsatisfiability proofs during solving are not unmanageable, they can still be avoided if the solver itself is trusted.
- Verification of modern SAT solvers could help in better theoretical understanding of how and why they work. A rigorous analysis and verification of modern SAT solvers may reveal some possible improvements in underlying algorithms and techniques which can influence and improve other solvers as well.
- Verified SAT solvers can serve as trusted kernel checkers for verifying results of other untrusted verifiers such as BDDs, model checkers, and SMT solvers [SV09]. Also, verification of some SAT solver modules (e.g., Boolean constraint propagation) can serve as a basis for creating both verified and an efficient proof-checkers for SAT.
- We want to demonstrate that, thanks to the recent advances in software verification technology, the time has finally come when it is possible to have a non-trivial, widely used software fully verified. Such work would contribute to the *Verification Grand Challenge* [VSTTE].

In this paper we present our ongoing project on SAT solver verification, with largest parts already completed. The project aims at producing solvers that are both efficient and fully trusted. In order to achieve the desired, highest level of trust, a fully mechanized and machine-checkable formalization (within the Isabelle proof assistant) is being developed. Within the project and in this paper we consider three ways of specifying modern SAT solvers and the corresponding verification paradigms (each with its advantages and disadvantages):

Abstract state transition systems. We have formally verified several abstract state transition systems that describe SAT solvers [NOT06,KG07]. Verification of such systems proves to be of vital importance because it serves as a key building block in other approaches to formalization.

Imperative implementation. We have made a more detailed (compared to the abstract state transition systems) description of a SAT solver in an imperative pseudo programming language. In parallel, we have developed a corresponding SAT solver ArgoSAT in C++. The solver properties have been formalized and verified within the Hoare logic.

Shallow embedding into a proof assistant. We have defined a SAT solver as a set of recursive functions within higher order logic of the system Isabelle (regarded as a pure functional language) and its correctness has been formally proved, mainly by induction and equational reasoning. Based on this specification, an executable functional program has been generated by means of the *code extraction*.

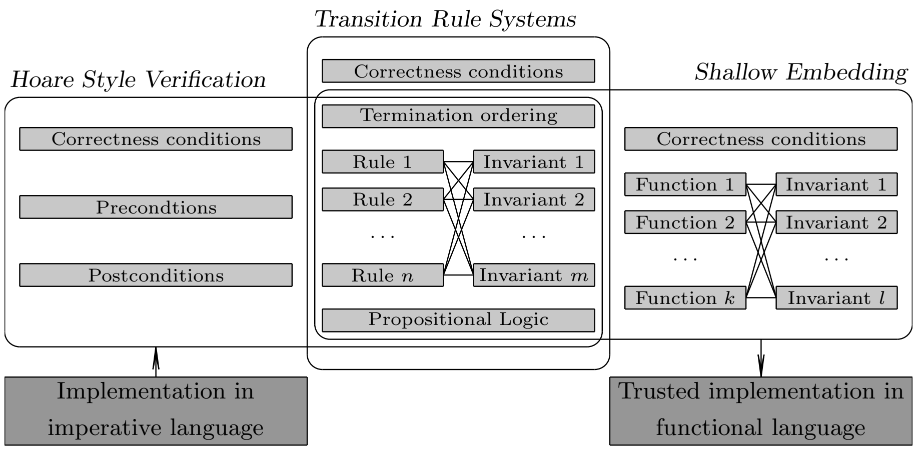

Fig. 1. Overall structure of the SAT solver verification project

An overall structure of our project is illustrated in Fig. 1. The basic component is a formalization of propositional logic and notions used in SAT solving. Notice that the confidence in our correctness proofs for a SAT solver, in bottom line, relies on the definition of satisfiability of propositional formulae. Fortunately, this definition is rather simple and can be checked by human inspection.

In the rest of the paper, we just briefly explain the modern SAT solving technology and algorithms and we refer the interested reader to other sources on these matters (e.g, [BHM$^+$09]). Due to the lack of space, we also give just a very few used definitions and just briefly comment only the central theorems and proofs. All the definitions, conjectures, and proofs have been completely formalized and verified within the Isabelle/Isar system [NPW02] and the complete proof documents are available in [Mar08]. Parts of the described project have been already described elsewhere [MJ09a,Mar09a,Mar09b,Mar09c,MJ09b], but in this paper the project is described as a whole for the first time.

2 Background

SAT Problem and SAT Solvers. SAT is the problem of deciding if there is a valuation of propositional variables under which a given propositional formula (in conjunctive normal form) is true. It is one of the central problems in computer science. SAT is the first problem that was proved to be NP-complete [Coo71] and it still holds a central position in the field of computational complexity. There is a number of *SAT solvers* — procedures that solve the SAT problem. The majority of the state-of-the-art complete SAT solvers are based on the branch and backtrack algorithm called Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DP60,DLL62] and we consider only them. Spectacular improvements in their performance achieved in the last several years are due to (i) several conceptual enhancements on the original DPLL procedure, (ii) smart heuristic components, and (iii) better implementation techniques. Thanks to these advances, modern SAT solvers can handle more and more practical problems in areas such as electronic design automation, software and hardware verification, artificial intelligence, operations research.

Program Verification. Program verification is the process of proving that a computer program meets its specification (that describes the expected program behaviour). Early results date back to 1950's and to the pioneers in this field, including Alan Turing, John von Neumann, and John McCarthy. In the late 1960's, Robert Floyd introduced reasoning on flowcharts for proving program correctness and Anthony Hoare introduced the Hoare logic — an axiomatic semantics for programming constructs. Many of early results in mechanical proving of program properties were carried out by Robert Boyer and J Moore using their theorem prover. Following lessons from major software failures, an increasing amount of effort has been being invested in this field. To achieve the highest level of trust, correctness proofs must be mechanically checkable by proof assistants. Many fundamental algorithms and properties of data structures have been formalized and verified in this way. Lot of efforts has also been invested into formalization of programming language semantics, compilers, communication protocols, security protocols, etc. Proof assistants that are most commonly used for program verification nowadays are Isabelle, HOL, Coq, PVS, Nuprl, etc.

3 Formalization of Logic of Propositional CNF Formulae

Central notions in SAT solving are propositional formulae in conjunctive normal form (CNF) and their satisfiability. In this section, it will be briefly described how are these notions formalized within the higher order logic of Isabelle[1].

Syntax of CNF formulae. The syntax of propositional logic of CNF formulae is based on the following types.

Definition 1. *A* Variable *is identified with a natural number. A* Literal *is either a positive variable (*Pos *vbl) or a negative variable (*Neg *vbl). A* Clause *is a list of literals. A* CNF Formula *is a list of clauses.*

[1] Our theory builds upon the built-in theory `Main` which includes that basic notions of HOL as well as the theory of lists.

Several basic operations on these types are introduced[2]. For example, the *variable of a literal* l is denoted by ($\mathsf{var}\ l$) and the set of all variables that occur in a formula F is denoted by ($\mathsf{vars}\ F$). *The opposite literal of a literal* l is denoted by $\overline{l}$, and for $\mathsf{Pos}\ vbl$ the opposite literal is $\mathsf{Neg}\ vlb$, and for $\mathsf{Neg}\ vbl$ the opposite literal is $\mathsf{Pos}\ vbl$.

Semantics of CNF formulae. Semantics of CNF formulae is based on the notion of *valuation.* Although valuations are usually defined as mappings assigning Boolean values to variables, we use a definition that more closely relates to the internal working of modern SAT solvers.

Definition 2. *A* Valuation *is a list of literals*

Consistent valuations are of special importance as there is a bijective correspondence between consistent valuations and (partial) mappings of variables to Boolean values.

Definition 3. *A valuation v is* consistent*, denoted* ($\mathsf{consistent}\ v$)*, iff it does not contain both a literal and its opposite.*

The notions of truth and satisfiability are introduced by the following two definitions.

Definition 4. *A literal l is* true in a valuation v*, denoted $v \vDash l$, iff $l \in v$.*
A clause c is true in a valuation v*, denoted $v \vDash c$, iff $\exists l.\ l \in c \wedge v \vDash l$.*
A formula F is true in a valuation v*, denoted $v \vDash F$, iff $\forall c.\ c \in F \Rightarrow v \vDash c$.*
A literal l is false in a valuation v*, denoted $v \vDash\neg l$, iff $\overline{l} \in v$.*
A clause c is false in a valuation v*, denoted $v \vDash\neg c$, iff $\forall l.\ l \in c \Rightarrow v \vDash\neg l$.*
A formula F is false in a valuation v*, denoted $v \vDash\neg F$, iff $\exists c.\ c \in F \wedge v \vDash\neg c$.*

Definition 5. *A* model *of a formula F is a consistent valuation in which F is true. A formula F is* satisfiable*, denoted* ($\mathsf{sat}\ F$)*, iff it has a model i.e., $\exists v.$ ($\mathsf{consistent}\ v$) $\wedge\ v \vDash F$*

Although these logical notions are sufficient to formulate the correctness conditions for SAT solvers, in order to prove these conditions, many additional notions of propositional logic have to be introduced and their properties have to be formally proved. For example, entailment of a literal or a clause by a formula (denoted by $F \vDash l$ or $F \vDash c$), logical equivalence of two formulae (denoted by $F_1 \equiv F_2$), etc.

SAT solving related notions. Some notions specific to SAT solving are also introduced. For example, a unit clause c (denoted by $\mathsf{isUnit}\ c\ l\ v$), is a clause which contains a literal l undefined in v and whose all other literals are false in v; a reason clause c for the literal l (denoted by $\mathsf{isReason}\ c\ l\ v$), is a clause that

[2] All the presented definitions have been formalized (most of them by using primitive recursion), but, in order to simplify presentation and improve readability, we give them only informally.

contains l (true in v), whose all other literals are false in v, and their opposites precede l in v; the resolvent of two clauses (denoted by $\mathsf{resolve}\ c_1\ c_2\ l$), etc.

Modern SAT solvers slightly extend the notion of valuation by distinguishing two different kinds of literals: *decision* and *implied*.

Definition 6. *An* assertion trail (or trail) *is a list of (Literal, Bool) pairs. Decisions are assigned the value* $\top$, *and implied the value* $\bot$.

Example 1. A trail M could be $[+1, |-2, +6, |+5, -3, |-7]$. Decision literals are marked by the symbol $|$ and they split the trail into levels, so M has 4 different levels (marked by 0 to 3): $+1$, then $-2, +6$, then $+5, -3$, and -7.

There is a number of operations on assertion trails used within SAT solvers. These operations have also been formally defined within our theory and their properties have been formally proved. Some of these are the list of decisions in a trail (denoted by ($\mathsf{decisions}\ M$)), the list of decisions that precede the first occurrence of a given literal (denoted by $\mathsf{decisionsTo}\ l\ M$)), the number of levels in a trail (denoted by ($\mathsf{currentLevel}\ M$)), prefix of a trail up to the given level (denoted by ($\mathsf{prefixToLevel}\ \mathit{level}\ M$)), etc.

4 Verification of the State Transition Systems

Modern DPLL-based SAT solvers can be modelled as state transition systems. Such systems can define the top-level architecture of SAT solvers as mathematical objects that can be rigorously reasoned about and whose correctness is expressed in pure mathematical terms. During the last few years two such systems have been proposed [NOT06,KG07]. Both systems are accompanied by informal pen-and-paper correctness proofs.

Decide:
$$\frac{l \in F_0 \qquad l, \bar{l} \notin M}{M := M\ |l}$$

UnitPropagate:
$$\frac{c \in F \qquad \mathsf{isUnit}\ c\ l\ M}{M := M\ l}$$

Conflict:
$$\frac{\mathit{conflict} = \bot \qquad c \in F \qquad M \vDash \neg c}{\mathit{conflict} = \top \qquad C := c}$$

Explain:
$$\frac{\mathit{conflict} = \top \quad l \in C \quad c \in F \quad \mathsf{isReason}\ c\ \bar{l}\ M}{C := \mathsf{resolve}\ C\ c\ l}$$

Backjump:
$$\frac{\mathit{conflict} = \top \quad C \in F \quad C = l \vee l_1 \vee \ldots \vee l_k \quad \mathsf{level}\ \bar{l} > m \geq \mathsf{level}\ \overline{l_i}}{\mathit{conflict} = \bot \qquad M := (\mathsf{prefixToLevel}\ m\ M)\ l}$$

Learn:
$$\frac{C \notin F}{F := F \cup C}$$

Forget:
$$\frac{\mathit{conflict} = \bot \quad c \in F \quad F \setminus c \vDash c}{F := F \setminus c}$$

Restart:
$$\frac{\mathit{conflict} = \bot}{M := \mathsf{prefixToLevel}\ 0\ M}$$

Fig. 2. Abstract state transition system for a DPLL-based SAT solver

The system presented in [NOT06] is very coarse and it can capture many different state-of-the art SAT solvers, at a price that it is far from the actual implementations. The system presented in [KG07] is somewhat more specific and it gives a more detailed description of some parts of the solving process (especially the conflict analysis phase). We used this system as a starting point and developed a slightly modified transition rule system shown in Fig. 2. The system models the solver's behaviour as transitions between states that represent values of the global variables of the solver. Transitions between states are performed only by using the transition rules. The rules have guarded assignment form: above the line is a condition that enables the application of the rule, below the line is an update to the state variables. The solving process is finished when no transition rule applies and a final state is reached.

A *state* of the rule-based SAT solver $(F, M, C, conflict)$ consists of the formula F being tested for satisfiability, a trail M, a conflict analysis clause C, and a Boolean variable $conflict$ that flags if the current formula is false in the current valuation (i.e., if the *conflict analysis* is under way).

The transition rules given above informally, have been formalized using relations over states. For instance,

$$\begin{aligned}\textsf{unitPropagate}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2) \iff \\ \exists c\ l.\ c \in F_1\ \wedge\ \textsf{isUnit}\ c\ l\ M_1\ \wedge \\ M_2 = M_1 @ l\ \wedge\ F_2 = F_1\ \wedge\ C_2 = C_1\ \wedge\ conflict_1 = conflict_2\end{aligned}$$

We say that two states are in relation $\rightarrow$, if they are in one of the relations describing the transition rules.

Definition 7. *A state* $([\,], F_0, [\,], \bot)$ *is an* initial state *for the input formula* F_0.

A state s *is* final state *with respect to* $\rightarrow$*, if and only if it is its minimal element, i.e., if there is no state* s' *such that* $s \rightarrow s'$.

A final state is an accepting state *if it holds that* $conflict = \bot$.

A final state is a rejecting state *if it holds that* $conflict = \top$.

Our correctness proof for the above system is based on formulating a set of suitable invariants and a well-founded ordering defined on states that ensures termination (as illustrated in Fig. 1). We formulated suitable invariants and proved that they hold for each state reached from an initial state (i.e., we proved that each invariant holds for initial states and that each invariant holds after each rule application).

Definition 8.

$Invariant_{consistent}$:	$\textsf{consistent}\ M$
$Invariant_{distinct}$:	$\textsf{distinct}\ M$
$Invariant_{varsM}$:	$\textsf{vars}\ M\ \subseteq\ \textsf{vars}\ F_0$
$Invariant_{varsF}$:	$\textsf{vars}\ F\ \subseteq\ \textsf{vars}\ F_0$
$Invariant_{equiv}$:	$F \equiv F_0$
$Invariant_{impliedLiterals}$:	$\forall l.\ l \in M \implies F\ @\ (\textsf{decisionsTo}\ l\ M) \vDash l$
$Invariant_{Cfalse}$:	$conflict \implies M \vDash \neg C$
$Invariant_{Centailed}$:	$conflict \implies F \vDash C$
$Invariant_{reasonClauses}$:	$\forall\ l.\ l \in M\ \wedge\ l \notin (\textsf{decisions}\ M) \implies \exists\ c.\ (\textsf{isReason}\ c\ l\ M)\ \wedge\ F \vDash c$

For introducing a required well-founded ordering over states, we had to define several auxiliary orderings, as follows.

Definition 9.

$$\begin{aligned}
l_1 \prec^{lit} l_2 &\iff (\mathsf{isDecision}\ l_1) \wedge \neg(\mathsf{isDecision}\ l_2)\\
M_1 \succ_{trail} M_2 &\iff M_1 \prec^{lit}_{lex} M_2\\
M_1 \succ^{restrict}_{trail} M_2 &\iff (\mathsf{distinct}\ M_1)\ \wedge\ (\mathsf{vars}\ M_1) \subseteq (\mathsf{vars}\ F_0) \wedge\\
& (\mathsf{distinct}\ M_2)\ \wedge\ (\mathsf{vars}\ M_2) \subseteq (\mathsf{vars}\ F_0) \wedge\\
& M_1 \succ_{trail} M_2\\
F_1 \succ^{C}_{formula} F_2 &\iff C \notin F_1 \wedge C \in F_2\\
C_1 \succ^{M}_{cclause} C_2 &\iff \langle \mathsf{remdups}\ \overline{C_2}\rangle \prec^{M}_{mult} \langle \mathsf{remdups}\ \overline{C_1}\rangle\\
conflict_1 \succ_{cflag} conflict_2 &\iff conflict_1 = \bot\ \wedge\ conflict_2 = \top.
\end{aligned}$$

where $\prec^{lit}_{lex}$ *denotes the lexicographic extension of the ordering* $\prec^{lit}$ *and* $\prec^{M}_{mult}$ *is the multiset extension of the ordering of literals* $\prec^{M}$ *induced by their order in the list* M*.*

The ordering $\succ$, defined as lexicographic combination of four orderings as follows, is a well-founded ordering on states.

Definition 10. $(M_1, F_1, C_1, conflict_1) \succ (M_2, F_2, C_2, conflict_2) \iff$

$$\begin{aligned}
&M_1 \succ^{restrict}_{trail} M_2\ \vee\\
&M_1 = M_2\ \wedge\ conflict_1 \succ_{cflag} conflict_2 \vee\\
&M_1 = M_2\ \wedge\ conflict_1 = conflict_2\ \wedge\ C_1 \succ^{M_1}_{cclause} C_2 \vee\\
&M_1 = M_2\ \wedge\ conflict_1 = conflict_2\ \wedge\ C_1 = C_2 \vee\ F_1 \succ^{C_1}_{formula} F_2
\end{aligned}$$

It has been proved that the system is terminating, using the fact that for any two states s_1 and s_2 reachable from an initial state it holds that $s_1 \rightarrow s_2$ implies $s_1 \succ s_2$. By the given invariants, it has been also proved that the system is sound and complete.

Theorem 1 (Correctness). *For any satisfiable input formula, the system consisting of the given rules terminates in an accepting state, and for any unsatisfiable formula, it terminates in an rejecting state.*

The main advantage of the abstract state transition systems is that they are mathematical objects, so it is relatively easy to make their formalization within higher order logic and to formally reason about them. Also, their verification can be a key building block for other verification approaches. Disadvantages are that the transition systems do not specify many details present in modern solvers' implementation and that they are not directly executable.

More details on the verification of the abstract state transition systems for SAT are given in [MJ09b].

5 Hoare-style Verification

Verification of imperative programs is usually done in the Floyd-Hoare logic [Hoa69]. It is a formal system that provides a set of logical rules for reasoning about the correctness of computer programs with the rigor of mathematical logic. The central object in the Hoare logic is a *Hoare triple* that describes how the execution of a piece of code changes the state of a computation. A Hoare triple is of the form $\{P\}$ `code` $\{Q\}$, where P (the *precondition*) and Q (the *postcondition*) are logic formulae and `code` is a programming language code. Hoare triple should be read as: "given that the assertion P holds at the point before `code` is executed and the `code` execution terminates, the assertion Q will hold at the point after `code` was executed". Hoare triples are manipulated by the inference rules that are formulated for each construct of the used programming language. For example, the inference rule for the `while` statement is:

$$\frac{\{P \wedge B\}\ \ S\ \ \{P\}}{\{P\}\ \ \texttt{while}\ B\ \texttt{do}\ S\ \ \{\neg B \wedge P\}}$$

Using this approach, we have verified the core of our solver ArgoSAT[3] implemented in C++[4]. Its implementation [Mar09b] supports all standard modern SAT solving techniques, but in the same time it closely follows the abstract state transition system given in Sect. 4. For example, the rule UnitPropagate is applied by using the function applyUnitPropagate, implemented as:

```
void Solver::applyUnitPropagate() {
    assertLiteral(_Q.front(), false);
    _Q.pop_front();
}
```

However, using the Hoare logic for the language complex as C++ was out of our reach. Therefore, we designed a pseudo language rich enough to support the implementation of our SAT solver, but simple enough to formulate a convenient Hoare logic axioms for all its constructs. The whole of the solver's core has been expressed within this pseudo language[5] and, for example, the function applyUnitPropagate is expressed as:

```
function applyUnitPropagate() : Boolean
begin
  assertLiteral ((head Q), false);
  Q := (tail Q);
end
```

As already said, although abstract state transition systems give quite clear descriptions of modern DPLL-based SAT solvers, they are still relatively far from the actual implementations. For example, the UnitPropagate rule does not

[3] The web page of ArgoSAT is `http://argo.matf.bg.ac.yu`.

[4] The core of ArgoSAT, implementing the rules given in Fig. 2 in an efficient way, counts around 1500 lines, while the whole system counts around 5000 lines.

[5] The description of the solver in the pseudo language is somewhat shorter then in C++, because of the simplified syntax.

specify how are the unit literals found. In the implementation given above, the unit literals are placed in a unit propagation queue Q from where they are taken and asserted to M by using the assertLiteral function. Therefore, a precondition for the applyUnitPropagate function is that all literals of Q are unit literals. The following example Hoare triple states that this property is preserved after the function call:

$$\begin{array}{c}\{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \wedge\ \mathsf{isUnit}\ c\ l\ M\}\\ \mathsf{applyUnitPropagate}()\\ \{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \wedge\ \mathsf{isUnit}\ c\ l\ M\}\end{array}$$

There is a number of features like this one that are not covered by the state transition systems, but are present in the implementation that was verified. These include most techniques used in state-of-the-art SAT solvers (e.g., MiniSAT [ES04]), and the most significant ones are the following.

False and unit clause detection. One of the most important aspects for efficient implementation is how to detect whether there are false or unit clauses in F wrt. the current trail M, i.e., how to detect whether the rules Conflict and UnitPropagate (shown in Fig. 2) are applicable. This is achieved by using the *two-watch unit propagation scheme*, and the unit propagation queue Q from the example above is introduced as a part of this technique.

Special treatment of single-literal clauses. Clauses that contain only one literal are not stored in the current set of clauses F, but their literals are directly asserted to the level zero of the trail M. This simplifies the implementation of the two-watch propagation scheme (although it complicates some other parts of the implementation and the corresponding correctness proofs).

Strategy and implementation of the conflict analysis. During the conflict analysis process (modelled by the Conflict and Explain rules shown in Fig. 2), after the Conflict rule has been applied, there is a number of possible ways to apply the Explain rule. A strategy often employed by SAT solvers (also used in our implementation) is the *first unique implication point (firstUIP)*. With it, the Explain rule is always applied to the last falsified literal of C in M and the explaining is performed until the first point when the Backjump rule becomes applicable. Also, efficient data-structures are used for storing the conflict analysis clause C.

Concerning heuristic components, instead of proving correctness of their implementation, functions that implement them have been specified by Hoare triples. This way, for any implementation of a heuristic it suffices to prove that it meets the corresponding Hoare triple. For example, the selection of a literal for the Decide rule is specified as follows:

$$\{\mathsf{vars}\ M \neq \mathsf{vars}\ F_0\}\ \ \mathsf{selectLiteral}()\ \ \{\mathsf{var}\ ret \in \mathsf{vars}\ F_0 \wedge \mathsf{var}\ ret \notin \mathsf{vars}\ M\}$$

Once the solver has been described in the pseudo programming language, the preconditions and postconditions for each fragment of the code are manually specified and joint together, following a suitable Hoare logic for our pseudo language. These correctness conditions express only partial correctness and not termination of the program. Termination could be proved using the ordering similar to the one described in Sect. 4. The entry point to the solver is the solve function which, if terminates, sets the value of $satFlag$ (either to SAT or $UNSAT$). The main verification result is the following theorem.

Theorem 2 (Partial correctness). *The SAT solver satisfies the Hoare triple:* $\{\top\}$ solve*(F_0)* $\{(satFlag = UNSAT \wedge \neg \mathsf{sat}\ F_0) \vee (satFlag = SAT \wedge M \vDash F_0)\}$

The main benefit of using Hoare style verification is that it enabled us to address imperative code which is the way that most real-world SAT solvers are implemented. In addition, all verification conditions have formally been proved within a proof assistant. Thanks to this, the confidence in our solver ArgoSAT is higher compared to other C/C++ implementations. On the other hand, there is still a gap between our correctness proof and the C++ implementation. First, there is no formal link between C++ and our pseudo language implementation. Second, there has been a number of manual steps in formulating correctness conditions and joining them together. Isabelle supports automatic conditions generation of Hoare conditions, but it is for a very simple programming language that is not powerful enough to express a complex SAT solver implementation.

More details on our description of a solver in an imperative language and its Hoare-style verification are given in [Mar09a].

6 Shallow Embedding

When using the *shallow embedding into HOL* approach for verification, a program (a SAT solver in our case) is implemented within higher order logic which is, for this purpose, treated as a pure functional programming language, i.e., the program is expressed as a set of recursive functions in HOL. Then, the properties of these functions are proved mainly by induction and equational reasoning.

Although a programming paradigm had to be changed from imperative to pure functional, we tried to make an implementation that closely follows the one described in Sect. 5 and that is the core of our solver ArgoSAT. All aspects of the implementation that are present in the imperative implementation verified by the Hoare-style approach (and that are not covered by the state transition systems) are also present in our functional implementation within Isabelle[6].

In an imperative or object-oriented language, state of the solver is represented by using global or class variables. The solver functions access and change the state variables as their side-effects. In HOL, functions cannot have side-effects, so the solver state must be wrapped up in a record and passed around with each function call. In our implementation, the state of the solver is represented by the following Isabelle record:

```
record State =
  "getF" :: Formula
```

[6] Formal definitions of the solver functions count over 500 lines of Isabelle code.

```
  "getM" :: LiteralTrail
  "getC" :: Clause
  "getConflictFlag" :: Boolean
  ...
```

Notice that this record directly correspond to the state of the abstract state transition systems described in Sect. 4. However, in order to implement more advanced techniques, the state had to be extended, and in our final definition it contains 14 components.

All functions in our functional implementation receive the current solver state as their parameter and return the modified state as their result. For example, the function applyUnitPropagate is implemented as follows:

```
definition applyUnitPropagate :: "State ⇒ State"
where
"applyUnitPropagate state =
  (let state' = assertLiteral (hd (getQ state)) False state in
   state'⦇ getQ := tl (getQ state') ⦈)"
```

This explicit state passing can be hidden if standard monadic bind and return combinators are used. This support has been recently added to Isabelle along with a convenient Haskell-like do-syntax [BKH$^+$08]. In this syntax, the applyUnitPropagate function becomes:

```
definition applyUnitPropagate :: "State ⇒ State"
where
"applyUnitPropagate =
do
  Q ← readQ; assertLiteral (hd Q) False;
  Q' ← readQ; updateQ (tl Q')
done"
```

The function `readQ` gets the current Q from the state, and `updateQ` sets the current Q in the state to the specified value. This way the code is much easier to read and resembles the code in our imperative pseudo language (p9).

Once the solver has been defined in HOL, its properties are formally proved. Again, it has been proved that all states that are reached during the code execution (this time these are the states that are returned by the functions of the solver) satisfy a given set of invariants (as illustrated in Fig. 1). These invariants include all invariants formulated for the abstract state transition systems (Def. 8). Many functions in the implementation explicitly match the high-level transition rules, so a number of proofs that the code preserves basic invariants simply rely on the proven properties of the abstract state transition system. However, since the implementation employs a number of advances techniques that are not covered by the state transition systems (most notably the two-watch unit propagation scheme), the set of invariants is significantly extended, counting 24 invariants in total. Therefore, it had to be proved that the code preserves all the additional invariants and it turned out that this task was equally hard (if not harder) as proving the properties of the abstract state transition system.

The entry point to the solver is the `solve` function, and, for termination it was required to prove that this function is total. Only three functions have been defined by general recursion and their termination is not trivial. Since the function solve is the only entry point to our solver, all these three functions are called only indirectly by the function solve and all parameters that are passed to them are computed by the solver. Therefore, it was sufficient to show that they terminate for those values of their input parameters that could actually be passed to them during a solver's execution starting from an initial state. We have used Isabelle's built-in features to model this kind of partiality [Kra08]. Once these partial correctness lemmas have been formulated, they are easily proved using termination orderings given in Def. 9.

When it has been proved that the invariants are preserved throughout the code, and that the function solve is total, the main correctness theorem is very easily formulated and proved.

Theorem 3 (Correctness). solve F_0 = sat F_0

Unlike the Hoare-style approach which starts with an existing solver implementation, when using the shallow embedding approach, the executable code in one of the leading functional languages (Haskell, SML, or OCaml) can be exported by using the *code extraction*, supported by Isabelle. When applying the code extraction, the term language of logic within a proof assistant is identified with the term language of the target language and the verified program correctness is transferred to the exported program, up to the simple transformation rules. As an example, the extracted code in Haskell for the function applyUnitPropagate is given bellow.

```
applyUnitPropagate :: State_ext_type () -> State_ext_type ();
applyUnitPropagate state =
  let {
    state' = assertLiteral (hd (getQ state)) False state;
  } in getQ_update (\ x -> tl (getQ state')) state';
```

Advantages of using the shallow embedding are that, once the solver is defined within the proof assistant, it is possible to perform its verification directly inside the logic and a formal model of the operational or denotational semantics of the language is not required. Also, executable code can be extracted and it can be trusted with a very high level of confidence. On the other hand, it is required to build a fresh implementation of a SAT solver within the logic. Also, special techniques must be used to have mutable data-structures and consequently, an efficient generated code.

More details on the verification by shallow embedding are given in [Mar09c]. We also used this approach for verification of the classic DPLL procedure, and details are given in [MJ09a].

7 Proof Management

Although it is hard to quantify the efforts invested in formally proving correctness conditions described in this work, we estimate it to be around one man-year.

The proof scripts make around 30000 lines of Isabelle code and the generated PDF proof documents have around 700 pages (these numbers are, of course, heavily dependent on the indentation style used). Proof-checking time by Isabelle is under 5 minutes on a 1.6GHz/512Mb RAM machine running Linux. We estimate that careful investigation of the proof text and its reorganization mainly by extracting some common parts of different proofs into lemmas could lead to 10-20 percent reductions.

During this verification effort, some interesting technical issues arose. In order to make such a large-scale verification effort possible, it was necessary to introduce some kind of modularity to the formalization. The crucial step in this direction was to prove the properties of the abstract state transition systems of [NOT06,KG07] and then use these proofs in the correctness proofs of the low-level implementations (either in the Hoare style or by means of the shallow embedding into HOL). A good direction to follow would be to define internal data-structures (for example the assertion trail) as abstract data-types (ADT) with some desired properties given axiomatically. Although, unfortunately, this has not been explicitly done in our formalization, this idea has been followed to some extent. Namely, after introducing basic definitions, we proved lemmas that could be regarded as axioms of the ADT and all further proofs relied only on those lemmas, without using the low-level properties of the implementation. This enables changing the low-level implementation into a more efficient one without changing much of the whole correctness proof.

When proving properties about recursively defined functions we had a dilemma whether to repeat the same induction scheme in proofs of many similar lemmas (one for each property of the recursive function) or to formulate one bigger lemma that groups all assumptions and conclusions for several properties that are being shown. We took the second approach and reduced the total number of lemmas and the total size of proofs, but the price that had to be payed is that we lost track of which assumptions are effectively used for proving a specific conclusion. For example, most of our high-level lemmas that state that invariants are preserved by the function calls assume that all invariants hold before the function call and show that all invariants hold after the function call. Precise mutual relationships between invariants can be determined only by analyzing the proof texts which can be very tedious.

8 Related Work

First steps towards verification of SAT solvers have been made only recently. The authors of two transition rule systems for SAT informally proved their correctness [NOT06,KG07]. Zhang and Malik have informally proved correctness of a modern SAT solver [ZM03]. Lescuyer and Conchon have formalized, within the system Coq, a SAT solver based on the classic DPLL procedure [LS08]. Shankar and Vaucher have formally and mechanically verified a high level description of a modern DPLL-based SAT solver within the system PVS [SV09]. Although these approaches include most state-of-the art SAT algorithms, lower-level implementation techniques (e.g., two-watch unit propagation scheme) are not covered by any of these descriptions. Our project provides fully mechanized correctness proofs for modern SAT solvers within three verification paradigms with both higher and lower level state-of-the-art SAT techniques, and, as we are aware of, it is the only such formalization.

9 Future Work

One of the main remaining tasks in our project is to increase the efficiency of the code exported from the shallow embedding specification. First, there are several low-level algorithmic improvements that have to be made. For example, in the current implementation, checking if a literal is true in a trail M requires performing a linear-time scan through the list, while real-world solvers cache truth values of all literals in an array and so allow a constant time check. Also, implementation of some higher-level heuristics has to be more involved. For example, currently we have implemented only a trivial decision heuristic that picks a random undefined literal, but in order to have a practically usable solver, an advanced decision heuristic (e.g., the MiniSat one) should be used. It would also be useful to implement forgetting and restarting techniques [KG07,NOT06]. Although these modifications require more work, we believe that they are rather straightforward. However, the most problematic issue is the fact that because of the pure functional nature of HOL no side-effects are possible and there can be no *destructive updates* of data-structures. To overcome this problem, we are planning to instruct the code generator to generate monadic Haskell and imperative ML code which would lead to huge efficiency benefits since it allows mutable references and arrays [BKH+08]. We hope that with these modifications, the generated code could become practically usable and comparable to state-of-the-art SAT solvers and this is the subject of our current work.

10 Conclusions

In this paper we gave an overview and the current status of our ongoing project on the SAT solver verification. The central part of the project is the formalization of modern SAT solvers. The complete formalization has been made within Isabelle/Isar proof assistant and is publicly available. We have, so far, invested around 1.5 man-years into this project (this work includes solver developments, their verification, and writing accompanying documents) and we estimate that 0.5 more man-years will be necessary to complete the project. SAT solvers have been formalized in three different ways: as abstract state transition systems, as imperative pseudo programming language code, and as a set of recursive HOL functions and all three formalizations have been verified using different paradigms. Each of them has its own advantages and disadvantages, making them in some aspects complementary and in some aspects overlapping.

Although there are other attempts at proving correctness of modern SAT solvers, to our best knowledge, our project gives the most detailed formalized and fully verified descriptions of a modern SAT solver so far. We are planning to work on improving efficiency of our trusted, generated solver and we hope its efficiency will be comparable to those of modern SAT solvers.

References

[BHM$^+$09] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability.* IOS Press, 2009.

[BKH$^+$08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkok, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOLs '08*, LNCS 5170, Montreal, 2008.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd STOC*, New York, 1971.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM* 5(7), pp. 394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3), pp. 201–215, 1960.

[ES04] N. Een and N. Sorensson. An Extensible SAT Solver. In *SAT '03*, LNCS 2919, S. Margherita Ligure, 2003.

[Gel07] A. Van Gelder. Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In *SAT '07*, LNCS 4501, Lisbon, 2007.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12(10), pp. 576–580, 1969.

[Kra08] A. Krauss. Defining recursive functions in Isabelle/HOL. `http://isabelle.in.tum.de/documentation.html`, 2008.

[KG07] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCos '07*, LNCS 4720, Liverpool, 2007.

[LS08] S. Lescuyer and S. Conchon A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs'08: Emerging Trends*, Montreal, 2008.

[Mar08] F. Marić, SAT Solver Verification. *The Archive of Formal Proofs,* `http://afp.sf.net/entries/SATSolverVerification.shtml`.

[Mar09a] F. Marić. Formalization and Implementation of SAT solvers. *J. Autom. Reason.* To appear. 2009.

[Mar09b] F. Marić. Flexible Implementation of SAT solvers. Manuscript submitted.

[Mar09c] F. Marić. Formal Verification of a Modern SAT Solver. Manuscript submitted.

[MJ09a] F. Marić and P. Janičić. Formal Correctness Proof for DPLL Procedure. *Informatica.* To appear. 2009.

[MJ09b] F. Marić, P. Janičić. Formalization of Abstract State Transition Systems for SAT. In preparation.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM* 53(6), pp. 937–977, 2006.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.

[SV09] N. Shankar and M. Vaucher. The mechanical verification of a DPLL-based satisfiability solver. In preparation.

[ZM03] L. Zhang and S. Malik. Validating SAT Solvers Using Independent Resolution-Based Checker. In *DATE '03*, Münich, 2003.

[VSTTE] *Verified Software: Theories, Tools, Experiments.* Conference. `http://vstte.ethz.ch/`