# 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 a 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 (usually given in conjunctive normal form). The SAT problem 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. The majority of the state-of-the-art complete SAT solvers are based on the backtracking algorithm called Davis-Putnam-Logemann-Loveland (DPLL) [DP60,DLL62]. Spectacular improvements in the performance of SAT solvers have been achieved in the last decade and nowadays SAT solvers can decide satisfiability of propositional formulae with tens of thousands of variables and millions of clauses. 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.

The 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.

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

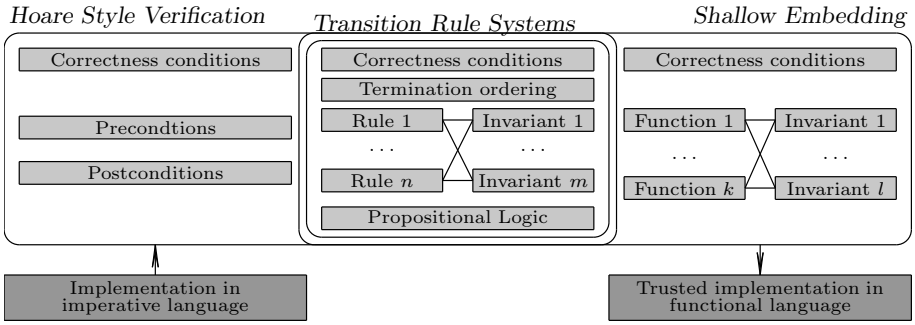


**Fig. 1.** Overall structure of the SAT solver verification project

One approach for achieving a higher level of confidence in SAT solvers' results, successfully used in recent years, is *proof-checking*. Solvers are modified so that they output not only *SAT* or *UNSAT* answers, but also evidences for their claims (models for satisfiable and proof-objects for unsatisfiable instances) which are then checked by independent checkers. Proof-checking is relatively easy to implement, but it has some drawbacks. Generating proof-objects introduces some runtime and storage overheads (proofs are typically large and may consume gigabytes of storage space) [Gel07]. Since proof-checkers have to be trusted, they must be very simple programs so they can be ,,verified" by code inspection.

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) but 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 and their rigorous analysis may reveal some possible improvements in underlying algorithms and techniques.
- Verified SAT solvers can serve as trusted kernel checkers for verifying results of other untrusted verifiers (e.g., BDDs, model checkers, SMT solvers) [SV09]. Also, verification of some SAT solver modules (e.g., BCP) can serve as a basis for creating both verified and an efficient proof-checkers for SAT.
- In addition to the above benefits, 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*.

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 is being developed. An overall structure of our project is illustrated in Fig. 1. Within the

project, 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++. Solver's 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. Based on this specification, an executable functional program has been generated by means of the *code extraction*.

In the rest of this paper, due to the lack of space, we 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. For a detailed survey of modern SAT solving technology and algorithms we refer the interested reader to other sources on these matters (e.g, [BHM$^+$09]).

## 2 Formalization of Logic of Propositional CNF Formulae

The syntax 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.*

Several basic operations on these types are introduced (e.g., *variable of a literal* $l$, denoted by (var $l$), the set of all variables that occur in a formula $F$, denoted by (vars $F$), *the opposite literal of a literal* $l$, denoted by $\overline{l}$).

The semantics of CNF formulae is based on the notion of *valuation*.

**Definition 2.** *A* Valuation *is a list of literals. For a given valuation* $v$*, a* literal $l$ is true *(denoted* $v \models l$*) iff* $l \in v$*, and is* false *(denoted* $v \models \neg l$*) iff* $\overline{l} \in v$*. A* clause $c$ is true*, denoted* $v \models c$*, iff* $\exists l.\ l \in c \wedge v \models l$*, and is* false *(denoted* $v \models \neg c$*) iff* $\forall l.\ l \in c \Rightarrow v \models \neg l$*. A* formula $F$ is true *(denoted* $v \models F$*) iff* $\forall c.\ c \in F \Rightarrow v \models c$*, and is* false *(denoted* $v \models \neg F$*) iff* $\exists c.\ c \in F \wedge v \models \neg c$*. A valuation* $v$ *is* consistent*, denoted* (consistent $v$)*, iff it does not contain both a literal and its opposite. A* model *of a formula* $F$ *is a consistent valuation in which* $F$ *is true. A formula* $F$ *is* satisfiable*, denoted* (sat $F$)*, iff it has a model.*

Note that, although total valuations are usually defined as Boolean variable assignments, the given definition that covers also partial valuations and more closely relates to the internal working of modern SAT solvers. The confidence in our correctness proofs for a SAT solver, in bottom line, relies on the given definitions. Fortunately, they are rather simple and can be checked by human inspection. Still, in order to prove correctness conditions, many additional notions have to be introduced and their properties have to be formally proved (e.g., entailment of a literal or a clause by a formula, denoted by $F \models l$ or $F \models c$, logical equivalence of two formulae, denoted by $F_1 \equiv F_2$).

*SAT solving related notions.* Some notions specific to SAT solving are also introduced within our formalization. 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 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*. For example, 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 (labelled 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 $(\mathsf{prefixToLevel}\ level\ M)$), etc.

## 3 Verification of the State Transition Systems

Modern DPLL-based SAT solvers can be modelled as abstract state transition systems (ASTS). Such systems 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 accompanied by informal, pen-and-paper correctness proofs. We used ASTS from [KG07] as a starting point and developed a slightly modified ASTS 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 are performed only by using the transition rules. The rules have guarded assignment form: above the line is a condition that enables the rule application, below the line is an update to the state variables. The solving process is finished when no transition rule applies.

Our system has been formalized in the following way. A *state* $(F, M, C, conflict)$ consists of a formula $F$ being tested for satisfiability, a trail $M$, a conflict analysis clause $C$, and a Boolean variable $conflict$ that flags if the current for-

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

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

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

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

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

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

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

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

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

mula is false in the current valuation (i.e., if the *conflict analysis* is under way). The rules have been formalized using relations over states. For instance,

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

Two states are in relation $\rightarrow$ iff they are in one of the relations describing the transition rules. A state $([\,], F_0, [\,], \bot)$ is an *initial state* for the input formula $F_0$. A state $s$ is *final state* with respect to $\rightarrow$, iff 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 $conflict = \top$.

**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.*

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). For example, we proved that the following invariants hold for each state reached from an initial state.

$Invariant_M$: $\mathsf{consistent}\ M\ \wedge\ \mathsf{distinct}\ M$

$Invariant_{vars}$: $\mathsf{vars}\ M\ \cup\ \mathsf{vars}\ F \subseteq\ \mathsf{vars}\ F_0$

$Invariant_{equiv}$: $F \equiv F_0$

$Invariant_{impliedLiterals}$: $\forall l.\ l \in M\ \Longrightarrow\ F\ @\ (\mathsf{decisionsTo}\ l\ M) \vDash l$

$Invariant_C$: $conflict\ \Longrightarrow\ M \vDash \neg C\ \wedge\ F \vDash C$

$Invariant_{reasonClauses}$: $\forall\ l.\ l \in M\ \wedge\ l \notin (\mathsf{decisions}\ M)\ \Longrightarrow$
$\exists\ c.\ (\mathsf{isReason}\ c\ l\ M)\ \wedge\ F \vDash c$

The main advantage of the ASTSs 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' implementations and that they are not directly executable. More details on the verification of the ASTSs for SAT are given in [MJ09b].

## 4 Hoare-style Verification

Verification of imperative programs is usually done in the Floyd-Hoare logic [Hoa69]. Its central object is a *Hoare triple* of the form $\{P\}$ `code` $\{Q\}$. Hoare triple should be read as: "given that the *precondition* $P$ holds before `code` is executed and the `code` execution terminates, the *postcondition* $Q$ 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 programming language.

Using this approach, we have verified the core of our solver ArgoSAT[1] implemented in C++[2]. Its implementation [Mar09b] closely follows the abstract state transition system given in Sect. 3, but also supports all standard techniques present in a modern SAT solver (e.g., MiniSAT [ES04]) not covered by the ASTS. Most important of these are the *two-watched literal unit propagation scheme* used for efficient detection of false and unit clauses in $F$ wrt. the current trail $M$, *special treatment of single-literal clauses* which are directly asserted to the decision level zero of the trail $M$ instead of adding them to $F$, and *efficient implementation of conflict analysis* using specialized data-structures for storing the conflict clause $C$.

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[3]. As an example, we list one function:

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

Following the two-watched literal scheme, all unit literals are placed in a unit propagation queue $Q$ from where they are taken and asserted to $M$. 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:

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

Heuristic components were specified only by Hoare triples. This way, for any implementation of a heuristic it suffices to prove that it meets the corresponding triple. For example, the selection of a literal for the Decide rule is specified as:

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

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

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

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

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. The entry point to the solver is the solve function which, if terminates, sets the value of $satFlag$ (either to $SAT$ or $UNSAT$).

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

The main benefit of using the Hoare style verification is that it enabled us to address imperative code which is the way that most real-world SAT solvers are implemented. 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. More details on our description of a solver in an imperative language and its Hoare-style verification are given in [Mar09a].

## 5 Shallow Embedding into HOL

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

Although a programming paradigm had to be changed from imperative to pure functional, our implementation closely follows the one described in Sect. 4 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 are also present in our functional implementation within Isabelle[4].

In an imperative or object-oriented language, the 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. For example, the following Isabelle record directly correspond to the state of the abstract state transition systems described in Sect. 3:

```
record State =
  "getF" :: Formula
  "getM" :: LiteralTrail
  "getC" :: Clause
  "getConflictFlag" :: Boolean
```

However, in order to have more advanced techniques implemented, 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. This explicit

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

state passing can be hidden if standard *monadic 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"
```

Functions `readQ` and `updateQ` modify the $Q$ component of the current state.

Once the solver has been defined in HOL, its properties are formally proved. The main result is the following correctness theorem.

**Theorem 3 (Correctness).** $\mathsf{solve}\ F_0 = \mathsf{sat}\ F_0$

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, but also include additional ones (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 ASTS. For termination, it was required to prove that the function `solve` is total. Only three functions called by it have been defined by general recursion and their termination is not trivial. Since the function solve is the only entry point to our solver, it was sufficient to prove termination of these functions only 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] and reused the orderings defined for abstract state transition systems to prove termination.

Unlike the Hoare-style approach that 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.

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 used this approach also for verification of the classic DPLL procedure, and details are given in [MJ09a].

## 6 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.

## 7 Conclusions and Future Work

In this paper we gave an overview of our ongoing project on the modern SAT solver verification. 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. All three formalizations have been verified using appropriate paradigms. Each of them has its own advantages and disadvantages, making them in some aspects complementary and in some aspects overlapping. The complete formalization has been made within Isabelle/Isar proof assistant and is publicly available[5]. Although it is hard to quantify the efforts invested in formally proving correctness conditions described in this work, we estimate that we have, so far, invested around 1.5 man-years into this project. 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.

One of the main remaining tasks in our project is to increase the efficiency of the code exported from the shallow embedding specification. Implementation of some heuristic components 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., VSIDS) should be used. Also, several low-level algorithmic improvements have to be made. 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

[5] The proof scripts make around 30000 lines of Isabelle code.

and comparable to state-of-the-art SAT solvers and this is the subject of our current work.

## 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. In preparation.

[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.