INFORMATICA, 2009 Vol. 0, No. 0, 1–0

Formal Correctness Proof for DPLL Procedure *

Filip MARIĆ, Predrag JANIČIĆ,

Faculty of Mathematics, University of Belgrade
Studentski trg 16, Belgrade, Serbia
E-mail: filip@matf.bg.ac.rs, janicic@matf.bg.ac.rs

Abstract. The DPLL procedure for the SAT problem is one of the fundamental algorithms in computer science, with many applications in a range of domains, including software and hardware verification. Most of the modern SAT solvers are based on this procedure, extending it with different heuristics. In this paper we present a formal proof that the DPLL procedure is correct. As far as we know, this is the first such proof. The proof was formalized within the Isabelle/Isar proof assistant system. This proof adds to the growing body of formalized mathematical knowledge and it also provides a number of lemmas relevant for proving correctness of modern SAT and SMT solvers.

Key words: SAT problem, DPLL procedure, formal proofs, Isabelle, Isar

1. Introduction The propositional satisfiability problem (SAT) is the problem of deciding whether there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It is a canonical NP-complete problem (Cook, 1971) and it holds a central position in the field of computational complexity.

One of the first algorithms for testing satisfiability is a branch and backtracking procedure called Davis-Putnam-Logemann-Loveland (in short DPLL) procedure (Davis et al., 1960, Davis et al., 1962). Although there have been many improvements to this procedure (including techniques called backjumping, conflict-driven lemma learning, restarts, etc.) it is still a core of the majority of the state-of-the-art complete SAT solvers (e.g., zChaff (Moskewicz et al., 2001), MiniSAT (Eén et al., 2003) and their successors). Modern SAT solvers show excellent performance even for huge formulae. They are used in many practical applications (electronic design automation, hardware and software verification, scheduling, etc.). SAT solving and modifications of the DPLL procedure such as DPLL(T) (Nieuwnhuis et al., 2006) are also used for SMT (Satisfiability Modulo Theories) — the problem of deciding whether a given formula is satisfiable with respect to a background first-order theory T (Ranise et al., 2006). SMT solving

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

also has many important industrial applications. For a survey of SAT solvers, their performances, and applications see, for instance, (Gu et al., 1997, Zhang et al., 2002, Le Berre et al., 2005).

Despite their significance and wide applications, there are still no formal correctness proofs for SAT and SMT solvers, neither for the original DPLL procedure nor for its modern successors. For most of the modern SAT solvers, there are no even informal correctness proofs. In this paper we address this issue and present a first formalized correctness proof for the DPLL procedure, a proof that can be verified by an independent and reliable proof checking system. In its forty-five years history, there were no doubts about the correctness of this algorithm, but we believe that it is important to have its correctness proof formalized for, at least, the following two reasons: first, the correctness proof for the DPLL procedure will be useful for checking correctness of modern SAT and SMT solvers, some of which are still unreliable[2]; second, this proof adds to the growing body of formalized, verifiable mathematical knowledge (which is important as the rigour, reliability, and objectivity of formal proofs is vital in many computer science applications, such as software and hardware verification).

Our correctness proof for the DPLL procedure is formalized within Isabelle proof assistant and for object-level proofs we use Isar (Intelligible semi-automated reasoning) language, natively supported in Isabelle. For definitions of some functions we use primitive recursion, also natively supported in Isabelle. We also use Isabelle's built-in theory of lists (Nipkow et al., 2005; pp. 16) and, to a limited extent, Isabelle's built-in theory of sets (only for finite sets) (Nipkow et al., 2005; pp. 109).

Overview of the paper. In Section 2 we give background information on the DPLL algorithm, on formal proofs and the Isabelle/Isar system, and on program verification. In Section 3 we give basic notation, definitions and properties of propositional logic, required for our proof. In Section 4 we give a formalization of the DPLL procedure (with one concrete implementation discussed in Appendix), and in Section 5 we prove the procedure's total correctness (partial correctness and termination). In Section 6, we discuss some technical details and give some fragments of our formalization made in Isabelle/Isar. In Section 7 we briefly discuss related work, and in Section 8 we draw final conclusions and discuss future work.

[2] For instance, over the previous years, several SMT solvers turned out to be unsound according to the results from SMT competitions `http://www.smtcomp.org/`

```
Procedure DPLL(CNF formula Φ)
    if Φ is empty return yes.
    else if there is an empty clause in Φ return no.
    else if there is a pure literal l in Φ return DPLL(Φ(l)).
    else if there is a unit clause {l} in Φ return DPLL(Φ(l)).
    else
        select a variable v occurring in Φ.
        if DPLL(Φ(v))=yes
            return yes.
        else
            return DPLL(Φ(¬v)).
        end
    end
```

Fig. 1. DPLL procedure

2. Background

Davis-Putnam-Logemann-Loveland (in short DPLL) procedure. The Davis-Putnam procedure was introduced in 1960 by Martin Davis and Hilary Putnam (Davis et al., 1960). Two years later, Martin Davis, George Logemann, and Donald W. Loveland introduced a refined version of the algorithm, in which they replaced the *elimination rule* by a *splitting rule* (Davis et al., 1962). In this newer version, the splitting rule leads to two smaller subproblems (one for each truth value for a selected variable), instead of a single, possibly larger, subproblem generated by the elimination rule. Nowadays, this later version of the algorithm is often referred to as *DPLL procedure*. The algorithm is shown in Figure 1. In the algorithm, Φ is a set of propositional clauses tested for satisfiability. $\Phi(l)$ denotes the formula obtained from Φ by substituting a literal l by $\top$, by substituting the opposite literal of l by $\bot$, and by simplifying afterwards. A literal is pure if it occurs in the formula but its opposite literal does not occur. A clause is unit if it contains only one literal. A non-recursive version of the algorithm can be found in (Davis et al., 1994). There are also rule based descriptions of some more advanced versions of this algorithm (Krstić et al., 2007, Nieuwnhuis et al., 2006).

The selection of a variable v within the given algorithm is critical for its performance. Choosing a variable may be trivial — choosing a first remaining variable or a random variable, but it can also be very complex. In the original version of the procedure, the variable occurring in the first clause of minimal length was chosen. The worst case complexity for this procedure on 3-SAT (3-

SAT is a variant of the SAT problem, with all clauses consisting of exactly three literals) is $O(2^{0.762n})$ (Cook et al., 1997). For more references on selecting a split variable and on worst case complexity analysis of the DPLL procedure see, for instance, (Cook et al., 1997, Irgens et al., 2004).

Formal proofs and Isabelle. Over the last years, in all areas of mathematics and computer science, with a history of huge number of flawed published mathematical proofs and also flawed software and hardware components, formal proofs (machine verifiable, given in object-level form, in terms of axioms and inference rules) have gained more and more importance. There are growing efforts in this direction, with many extremely complex mathematical theorems formally proved[3] and with many software tools producing and checking formal proofs. Isabelle is a generic theorem prover that supports a variety of logics, with Gentzen's natural deduction as the basic built-in logic (Paulson, 1994). Distinctive Isabelle's features include representation of logics within a meta-logic and the use of higher-order unification to combine inference rules. Isabelle can be applied to reasoning in pure mathematics or verification of computer systems. Isabelle is one of the most popular theorem proving systems nowadays.

Readable formal proofs and Isar. Theorem proving system supporting both interactive proof development and some degree of automation have become quite successful in sizable applications in recent years. Most of them are based on traditional proof scripts which explicitly list all axioms and inference rules used in every single proof step. Despite success of semi-automated proving systems based on such scripts in formalizing fragments of mathematics and computer science, they are still not accepted by a wide range of researchers. The Intelligible semi-automated reasoning (Isar) (Wenzel, 2007) approach to readable formal proof documents aims to bridge the semantic gap between internal notions of proof given by state-of-the-art interactive theorem proving systems and an appropriate level of abstraction for user-level work. Isar is an alternative proof language interface layer, beyond traditional formal proof tactic scripts, which is much more readable for the users. The Isabelle/Isar system provides an interpreter for the Isar formal proof document language, and readable Isar proof documents are converted and executed as series of low-level inference steps. It allows users to express proofs in a human-friendly way but still have proofs that are automatically formally verified by an underlying proof system and that rely only on valid axioms and inference rules.

[3] For a list of selected formally proved theorems see, for instance, `http://www.cs.ru.nl/~freek/100/`.

Program verification. Program verification is the process of formally proving that a computer program meets its specification. Program verification is old, but very much active field. Following the lessons from major software failures in recent years, more and more efforts have been invested in this field. Many fundamental algorithms and properties of data structures have been formalized. Also, a lot of work has been devoted to formalization of compilers, program semantics, communication protocols, security protocols, etc. Formal verification is vital for SAT and SMT solvers and first steps in this direction have been made. For a short overview of results in program verification see Section 7.

3. Notation and Definitions In this section, we introduce notation, definitions, and basic propositions used in our formalized correctness proof for the DPLL procedure. Our proof is almost self-contained, so here we also define notions (and notation) of literals, clauses, formulae, satisfiability, etc. All notions introduced here are also formalized within Isabelle's higher order logic (Isabelle/HOL). Some of them are defined by primitive recursion, supported in Isabelle/HOL.

Formulae and logical connectives of this meta-logic ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) are written in the usual way. Ternary $if{-}then{-}else$ connective is also used: $if\ f\ then\ f_1\ else\ f_2$ denotes $f \Rightarrow f_1 \wedge \neg f \Rightarrow f_2$. The symbol $=$ denotes syntactical identity of two expressions.[4]

The theory that we built for expressing correctness of the DPLL procedure uses Isabelle's built-in theory of lists and Isabelle's built-in theory of sets (only for finite sets). Figure 2 informally describes the notions from these theories that we use.

We assume that all meta-logic formulae in the following text are implicitly universally quantified, if not stated otherwise. We use typed logic, but for better readability, when printing formulae we omit types, and use the following convention:

- vbl denotes a variable and has the type *nat*;
- $l, l', l_1, l_2, \ldots$ denote literals and have the type *Literal*;
- $c, c', c_1, c_2, \ldots$ denote clauses and have the type *Clause*;
- $F, F', F_1, F_2, \ldots$ denote formulae and have the type *Formula*;
- $v, v', v_1, v_2, \ldots$ denote valuations and have the type *Valuation*.

Definition 1. *A* variable *is identified with a natural number.*

[4]Note that in this presentation we make a clear distinction between syntactical identity and logical equivalence and use different symbols for them, $=$ and $\Leftrightarrow$ (while in Isabelle/HOL these two notions are denoted by the same symbol, $=$).

$[]$	the empty list.
$[e_1, \ldots, e_n]$	the list of n given elements $e_1, \ldots, e_n$.
$e \in list$	e is a member of the list $list$.
$e \# list$	the list obtained by prepending element e to the list $list$.
$list_1 @ list_2$	the list obtained by appending lists $list_1$ and $list_2$.
$head(list)$	the first element of the list $list$ (assuming the list is non empty).
$tail(list)$	the list obtained by removing the first element of the list $list$
$list \setminus e$	the list obtained by removing all occurrences of the element e from the list $list$
$list_1 \setminus list_2$	the list obtained from the list $list_1$ by removing from it all elements of the list $list_2$.
$list_1 \subseteq list_2$	all elements of $list_1$ are also elements of $list_2$.
$\|list\|$	the length of the list $list$.
$\{\}$	the empty set.
$e \in set$	e is a member of the set set.
$set_1 \cup set_2$	the union of the sets set_1 and set_2.

Fig. 2. Notions from the theory of lists and the theory of sets that are used.

Definition 2. *A* literal *is either a positive variable (denoted by $+vbl$) or a negative variable (denoted by $-vbl$).*

Definition 3. *A* clause *is a list of literals.*

Definition 4. *A* formula *is a list of clauses.*

Definition 5. *A* valuation *is a list of literals.*

Definition 6. *A* variable of a literal, *denoted $var(l)$, is defined in the following way: $var(+vbl) = var(-vbl) = vbl$.*

Definition 7. *A* opposite literal of a literal, *denoted $\overline{l}$, is defined in the following way: $\overline{+vbl} = -vbl$, $\overline{-vbl} = +vbl$.*

Notice that we abuse the notation and overload some symbols. For example, the symbol $\in$ denotes both set membership and list membership. It is also used to denote that a literal occurs in a formula.

Definition 8. *A formula F* contains a literal *l (and the literal l* occurs in the formula *F), denoted $l \in F$, iff $(\exists c)(c \in F \wedge l \in c)$.*

Symbol $vars$ is also overloaded and denotes the set of variables occurring in a clause, formula, valuation, defined by primitive recursion.

Definition 9. *A* set of variables that occur in a clause c, *denoted* $vars(c)$, *is defined in the following way:*

$$\begin{aligned} vars([\,]) &= \{\} \\ vars(l \# c) &= var(l) \cup vars(c) \end{aligned}$$

A set of variables that occur in a formula F, *denoted* $vars(F)$, *is defined in the following way:*

$$\begin{aligned} vars([\,]) &= \{\} \\ vars(c \# F) &= vars(c) \cup vars(F) \end{aligned}$$

A set of variables that occur in a valuation v, *denoted* $vars(v)$, *is defined in the following way:*

$$\begin{aligned} vars([\,]) &= \{\} \\ vars(l \# v) &= var(l) \cup vars(v) \end{aligned}$$

The semantics is introduced by the following definitions.

Definition 10. *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* $(\forall c)(c \in F \Rightarrow v \vDash c)$.

Definition 11. *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 12. *A valuation* v *is* inconsistent *iff it contains both literal and its opposite i.e.,* $(\exists l)(v \vDash l \wedge v \vDash \overline{l})$. *A valuation is* consistent *iff it is not inconsistent.*

Definition 13. *A* model *of a formula* F *is a consistent valuation* v *under which* F *is true i.e.,* $model(v, F)$ *iff* $consistent(v) \wedge v \vDash F$. *A formula* F *is* satisfiable, *denoted* $sat(F)$ *iff it has a model i.e.,* $(\exists v)(model(v, F))$

The following proposition gives some basic properties of the notions we have just introduced. These properties were formulated and proved in Isabelle/Isar and used in the correctness proof for the DPLL procedure.

Proposition 1.

	Basic properties of opposite literals:
$(A1)$	$\overline{\overline{l}} = l$
$(A2)$	$\overline{l_1} = l_2 \Leftrightarrow \overline{l_2} = l_1$
$(A3)$	$\overline{l_1} = \overline{l_2} \Leftrightarrow l_1 = l_2$
$(A4)$	$\overline{l} \neq l$
$(A5)$	$var(\overline{l}) = var(l)$
$(A6)$	$var(l_1) = var(l_2) \;\Leftrightarrow\; l_1 = l_2 \vee l_1 = \overline{l_2}$

	Basic properties of $vars$:
$(B1)$	$l \in c \Rightarrow var(l) \in vars(c)$
$(B2)$	$l \in F \Rightarrow var(l) \in vars(F)$
$(B3)$	$c \in F \Rightarrow vars(c) \subseteq vars(F)$
$(B4)$	$l \in v \Rightarrow var(l) \in vars(v)$
$(B5)$	$var(l) \in vars(c) \Leftrightarrow l \in c \vee \overline{l} \in c$
$(B6)$	$var(l) \in vars(F) \Leftrightarrow l \in F \vee \overline{l} \in F$
$(B7)$	$var(l) \in vars(v) \Leftrightarrow v \vDash l \vee v \vDash \neg\, l$

	Basic properties of consistent valuations:
$(C1)$	$consistent([\,])$
$(C2)$	$inconsistent(v \setminus l) \;\Rightarrow\; inconsistent(v)$

	Basic properties of the relation $\vDash$:
$(D1)$	$v \vDash c \setminus l \;\Rightarrow v \vDash c$
$(D2)$	$var(l) \notin vars(F) \wedge v \vDash F \;\Rightarrow\; v \setminus [l, \overline{l}] \vDash F$

	Basic properties of models and formula satisfiability:
$(E1)$	$model(v, F) \;\wedge\; vbl \notin vars(F) \Rightarrow$ $(\exists v')(model(v', F) \;\wedge\; vbl \notin vars(v'))$
$(E2)$	$F \subseteq F' \;\Rightarrow\; (sat(F') \Rightarrow sat(F))$
$(E3)$	$sat([\,])$
$(E4)$	$[\,] \in F \;\Rightarrow\; \neg sat(F)$

4. Formalization of the DPLL Procedure In this section, we present a formalization of the DPLL procedure and all required notions. We give a special attention to the notions of *pure literal* and *unit clause*, essential for certain steps of the procedure. All the given proofs are rigorously formulated and verified within Isabelle/Isar.

4.1. Substitution One of the basic steps of the DPLL procedure is substitution of a literal by logical constants $\top$ and $\bot$, and simplification of the obtained formula. This operation is formalized by the following definition.

Definition 14. *$F[l \to \top]$ is the formula that is obtained from F by deleting all clauses that contain l and deleting all occurences of the literal $\overline{l}$. It is defined by primitive recursion:*

$$
\begin{array}{lcl}
[\,]\,[l \to \top] & = & [\,] \\
(c \# F)\,[l \to \top] & = & \textit{if } l \in c \textit{ then} \\
 & & \quad F[l \to \top] \\
 & & \textit{else if } \overline{l} \in c \textit{ then} \\
 & & \quad (c \setminus \overline{l}) \# F[l \to \top] \\
 & & \textit{else} \\
 & & \quad c \# F[l \to \top]
\end{array}
$$

$F[l \to \bot]$ denotes $F[\overline{l} \to \top]$.

The following proposition (proved in Isabelle/Isar) gives some basic properties of of this operation.

Proposition 2.

(1)	$var(l) \notin vars(F[l \to \top])$
(2)	$var(l) \notin vars(F) \;\Rightarrow\; F[l \to \top] = F$
(3)	$l \notin F \wedge \overline{l} \notin F \;\Rightarrow\; F[l \to \top] = F[l \to \bot] = F$
(4)	$model(v, F) \;\wedge\; l \in v \;\Rightarrow\; model(v, F[l \to \top])$
(5)	$model(v, F) \;\wedge\; var(l) \notin vars(v) \;\Rightarrow\; model(v, F[l \to \top])$
(6)	$v \vDash F[l \to \top] \;\Rightarrow\; (l \# v) \vDash F$
(7)	$[l] \in F \;\Rightarrow\; \neg sat(F[l \to \bot])$
(8)	$l \in F \wedge \overline{l} \notin F \;\Rightarrow\; F[l \to \top] \subseteq F[l \to \bot]$

The following lemma suggests that the satisfiability of a formula can be, by using substitution, checked by testing the satisfiability of two smaller formulae. Since this is a fundamental lemma in the proof of the DPLL correctness, we give a sketch of its proof. This sketch is still very close to its formal, Isabelle/Isar counterpart. It also illustrates the use of the listed properties.

Lemma 1 (Split rule lemma)**.**

$$sat(F) \;\Leftrightarrow\; sat(F[l \to \top]) \;\vee\; sat(F[l \to \bot])$$

Proof:

($\Rightarrow$) : Let us assume $sat(F)$. This means that there is a valuation v such that $model(v, F)$, i.e., $consistent(v)$ and $v \vDash F$. We consider two cases:

1. $var(l) \notin vars(v)$: from Proposition 2(5) it follows that v is a model for $F[l \rightarrow \top]$, and, therefore, it holds that $sat(F[l \rightarrow \top])$.
2. $var(l) \in vars(v)$: from Proposition 1(B7), either $v \vDash l$ or $v \vDash \neg\, l$ holds.
(a) $v \vDash l$: from Proposition 2(4) it follows that v is a model for $F[l \rightarrow \top]$, and, therefore, it holds that $sat(F[l \rightarrow \top])$.
(b) $v \vDash \overline{l}$: from Proposition 2(4), it holds that v is a model for $F[\overline{l} \rightarrow \top]$, and, therefore, it holds that $sat(F[l \rightarrow \bot])$.

($\Leftarrow$) : Let us assume $sat(F[l \rightarrow \top]) \ \vee \ sat(F[l \rightarrow \bot])$.

Consider the case when $sat(F[l \rightarrow \top])$ holds. This means that there is a valuation v such that $model(v, F)$, i.e., $consistent(v)$ and $v \vDash F[l \rightarrow \top]$. From Proposition 2(1), it holds that $var(l) \notin vars(F[l \rightarrow \top])$. From Proposition 1(E1) applied to the formula $F[l \rightarrow \top]$, variable $var(l)$, and the valuation v, it follows that there is a valuation v' such that $consistent(v')$, $v' \vDash F[l \rightarrow \top]$ and $var(l) \notin vars(v')$. Then, from Proposition 2(6) applied to the valuation v', it follows that $(l \# v') \vDash F$. Since $var(l) \notin vars(v')$, it follows $\overline{l} \notin v'$ and therefore $consistent(l \# v')$. Finally, $sat(F)$ holds since $l \# v'$ is a model of F.

The case when $sat(F[l \rightarrow \bot])$ holds is analogous to the previous case, with the literal l replaced by $\overline{l}$. □

This lemma inspires a naive, but still sound and complete, procedure for satisfiability checking. In some situations, one of the two formulae $sat(F[l \rightarrow \top])$ and $sat(F[l \rightarrow \bot])$ from the above lemma does not need to be considered. For instance, if the first disjunct is satisfied, then the second one does not need to be checked (as indicated by the algorithm shown in Figure 1). Also, in some special cases discussed below it suffices to consider just one of these disjuncts.

4.2. Unit Clauses One sort of optimization of the mentioned naive procedure for satisfiability checking is based on exploiting *unit clauses*.

Definition 15. *A clause c is a* unit clause *iff it has only one literal, i.e., $c = [l]$. Then we also say that l is a* unit literal.

The following lemma shows that when a formula contains a unit clause, checking its satisfiability can be reduced to checking satisfiability of just one smaller formula (in constrast to Lemma 1).

Lemma 2 (Unit clause rule lemma).

$$[l] \in F \;\Rightarrow\; (sat(F) \Leftrightarrow sat(F[l \rightarrow \top]))$$

Proof: By Proposition 2(7) it holds that $[l] \in F \;\Rightarrow\; \neg sat(F[l \rightarrow \bot])$, so the lemma is a direct consequence of the split rule lemma (Lemma 1). □

4.3. Pure Literals Another sort of optimizations of the naive procedure for satisfiability checking is based on *pure literals*.

Definition 16. *A literal l is a* pure literal *in F iff $l \in F$ and $\overline{l} \notin F$.*

Lemma 3 (Pure literal rule lemma).

$$l \in F \wedge \overline{l} \notin F \;\Rightarrow\; (sat(F) \;\Leftrightarrow\; sat(F[l \rightarrow \top]))$$

Proof: By Proposition 2(8) it holds that

$$l \in F \wedge \overline{l} \notin F \;\Rightarrow\; F[l \rightarrow \top] \subseteq F[l \rightarrow \bot],$$

so the lemma is a simple consequence of the Proposition 1(E2) applied to $F[l \rightarrow \top]$ and $F[l \rightarrow \bot]$, and the Split rule lemma (Lemma 1). □

4.4. Definition of the DPLL Procedure A recursive definition of the DPLL procedure is given in the following definition.

Definition 17.

$dpll(F) \Leftrightarrow$
  if $F = [\,]$ *then*
    $\top$
  else if $[\,] \in F$ *then*
    $\bot$
  else if $hasPureLiteral(F)$ *then*
    $dpll(F[getPureLiteral(F) \rightarrow \top])$
  else if $hasUnitLiteral(F)$ *then*
    $dpll(F[getUnitLiteral(F) \rightarrow \top])$
  else if $dpll(F[selectLiteral(F) \rightarrow \top]$ *then*
    $\top$
  else $dpll(F[selectLiteral(F) \rightarrow \bot]$

Notice that the functions $getUnitLiteral$, $getPureLiteral$ and $selectLiteral$ returning literals and Boolean functions $hasUnitLiteral$, $hasPureLiteral$,

must be effectively defined in order to have an effective DPLL procedure. As said in Section 2, this can be done in many ways. The choice of a specific implementation of these functions, can affect the procedure performance but does not affect its correctness, as long as they meet the following specification (their sorts are obvious from the context):

(1)	$hasUnitLiteral(F) \Rightarrow$
	$[getUnitLiteral(F)] \in F$
(2)	$hasPureLiteral(F) \Rightarrow$
	$getPureLiteral(F) \in F \wedge \overline{getPureLiteral(F)} \notin F$
(3)	$F \neq [] \wedge [] \notin F \Rightarrow$
	$selectLiteral(F) \in F$

One simple way to define these functions is given in Appendix.

5. Termination and Correctness of the DPLL procedure In this section we prove termination and, finally, correctness of the DPLL procedure. Our proof roughly follows the informal proof given in (Davis et al., 1994). In Isabelle spirit, termination is ensured by defining a measure that is decreased by each recursive call of the procedure. This property is ensured by proving several propositions corresponding to different recursive calls.

5.1. Termination In order to prove termination of the specified procedure, we show that the total number of literals in all clauses of F is decreased by each recursive call.[5] This number, denoted by $numLiterals(F)$, is defined by primitive recursion.

Definition 18.

$$\begin{aligned} numLiterals([]) &= 0 \\ numLiterals(c \# F) &= |c| + numLiterals(F) \end{aligned}$$

From the following proposition it follows that the total number of literals in F is reduced by each recursive call. Because of that, the total number of literals in the formula can be used as a decreasing measure suitable for proving termination of the DPLL procedure. This measure and the following proposition are used by Isabelle/Isar for the automatic proof of termination.

[5] There are other suitable termination measures that can be used as well (e.g., the number of occuring variables).

Proposition 3.

(1)	$l \in F \;\Rightarrow\; numLiterals(F[l \rightarrow \top]) < numLiterals(F)$
(2)	$l \in F \;\Rightarrow\; numLiterals(F[l \rightarrow \bot]) < numLiterals(F)$
(3)	$F \neq [] \wedge [] \notin F \;\Rightarrow$ $numLiterals(F[selectLiteral(F) \rightarrow \top]) < numLiterals(F)$
(4)	$F \neq [] \wedge [] \notin F \;\Rightarrow$ $numLiterals(F[selectLiteral(F) \rightarrow \bot]) < numLiterals(F)$
(5)	$hasUnitLiteral(F) \;\Rightarrow$ $numLiterals(F[getUnitLiteral(F) \rightarrow \top]) < numLiterals(F)$
(6)	$hasPureLiteral(F) \;\Rightarrow$ $numLiterals(F[getPureLiteral(F) \rightarrow \top]) < numLiterals(F)$

5.2. Correctness Finally, we can prove the correctness of the procedure defined by Definition 17.

Theorem 1.

$$dpll(F) \;\Leftrightarrow\; sat(F)$$

Proof: As a base of the inductive proof, we consider the cases in which the function does not perform a recursive call. There are two such branches:

- If $F = []$ then, by Proposition 1(E3), $dpll(F) = \top$ and $sat(F) = \top$, so the conjecture trivially holds.
- If $F \neq []$ and $[] \in F$ then, by Proposition 1(E4), $dpll(F) = \bot$ and $sat(F) = \bot$, so the conjecture trivially holds.

Now, let us assume that the conjecture holds for each recursive call, and let us show that the conjecture holds for the top level procedure call. Therefore, let us assume the following inductive hypotheses.

$$(F \neq [] \;\wedge\; [] \notin F) \;\Rightarrow\; (hasPureLiteral(F) \;\Rightarrow\; dpll(F[getPureLiteral(F) \rightarrow \top]) \Leftrightarrow sat(F[getPureLiteral(F) \rightarrow \top]))$$

$$(F \neq [] \;\wedge\; [] \notin F \;\wedge\; \neg hasPureLiteral(F)) \;\Rightarrow\; (hasUnitLiteral(F) \;\Rightarrow\; dpll(F[getUnitLiteral(F) \rightarrow \top]) \Leftrightarrow sat(F[getUnitLiteral(F) \rightarrow \top]))$$

$$(F \neq [\,] \ \wedge\ [\,] \notin F \ \wedge\ \neg hasPureLiteral(F) \ \wedge\ \neg hasUnitLiteral(F)) \Rightarrow$$
$$dpll(F[selectLiteral(F) \rightarrow \top]) \Leftrightarrow sat(F[selectLiteral(F) \rightarrow \top])$$

$$(F \neq [\,] \ \wedge\ [\,] \notin F \ \wedge\ \neg hasPureLiteral(F) \ \wedge\ \neg hasUnitLiteral(F)) \Rightarrow$$
$$(\neg dpll(F[selectLiteral(F) \rightarrow \top]) \Rightarrow$$
$$dpll(F[selectLiteral(F) \rightarrow \bot]) \Leftrightarrow sat(F[selectLiteral(F) \rightarrow \bot]))$$

Let us consider different branches of $if\text{–}then\text{–}else$ in the definition of $dpll$ function:

- If $F \neq [\,]$ and $[\,] \notin F$, and $hasPureLiteral(F)$, then by the $dpll$ definition:

  $$dpll(F) \ \Leftrightarrow\ dpll(F[getPureLiteral(F) \rightarrow \top])$$

  Also, by the inductive hypothesis, it holds:

  $$dpll(F[getPureLiteral(F) \rightarrow \top]) \Leftrightarrow sat(F[getPureLiteral(F) \rightarrow \top])$$

  From the specification of $getPureLiteral$ and the assumption $hasPureLiteral(F)$, it holds that $getPureLiteral(F) \in F \ \wedge\ \overline{getPureLiteral(F)} \notin F$. Then, by Lemma 3:

  $$sat(F[getPureLiteral(F) \rightarrow \top]) \ \Leftrightarrow\ sat(F)$$

  Therefore, $dpll(F) \ \Leftrightarrow\ sat(F)$.
- If $F \neq [\,]$ and $[\,] \notin F$ and $\neg hasPureLiteral(F)$, and $hasUnitLiteral(F)$, then by the $dpll$ definition:

  $$dpll(F) \ \Leftrightarrow\ dpll(F[getUnitLiteral(F) \rightarrow \top])$$

  Also, by the inductive hypothesis, it holds:

  $$dpll(F[getUnitLiteral(F) \rightarrow \top]) \Leftrightarrow sat(F[getUnitLiteral(F) \rightarrow \top])$$

From the specification of $getUnitLiteral$ and the assumption $hasUnitLiteral(F)$, it holds that $[getUnitLiteral(F)] \in F$. Then, by Lemma 2:

$$sat(F[getUnitLiteral(F) \rightarrow \top]) \;\Leftrightarrow\; sat(F)$$

Therefore, $dpll(F) \;\Leftrightarrow\; sat(F)$.

- If $F \neq [\,]$ and $[\,] \notin F$ and $\neg hasPureLiteral(F)$, and $\neg hasUnitLiteral(F)$ then, from the specification of $dpll$ and the definition of $if - then - else$ connective, it holds that

$$\begin{aligned} dpll(F) \;\Leftrightarrow\; & (dpll(F[selectLiteral(F) \rightarrow \top]) \Rightarrow \top) \;\wedge \\ & (\neg dpll(F[selectLiteral(F) \rightarrow \top]) \Rightarrow \\ & \quad dpll(F[selectLiteral(F) \rightarrow \bot]) \end{aligned}$$

Therefore, it holds that

$$\begin{aligned} dpll(F) \;\Leftrightarrow\; & dpll(F[selectLiteral(F) \rightarrow \top]) \;\vee \\ & dpll(F[selectLiteral(F) \rightarrow \bot]). \end{aligned}$$

If $dpll(F[selectLiteral(F) \rightarrow \top])$ then, by the inductive hypothesis, it holds that $sat(F[selectLiteral(F) \rightarrow \top])$. Otherwise, if $\neg dpll(F[selectLiteral(F) \rightarrow \top])$ and $dpll(F[selectLiteral(F) \rightarrow \bot])$ hold then, by the inductive hypothesis, $sat(F[selectLiteral(F) \rightarrow \bot])$ holds. Therefore:

$$\begin{aligned} dpll(F) \;\Leftrightarrow\; & sat(F[selectLiteral(F) \rightarrow \top]) \;\vee \\ & sat(F[selectLiteral(F) \rightarrow \bot]). \end{aligned}$$

Then, by Lemma 1, it holds that $dpll(F) \;\Leftrightarrow\; sat(F)$.

□

This proof, together with the termination argument, proves the total correctness of the $dpll$ function.

6. Formalization in Isabelle/Isar Our formalization of the DPLL procedure and its correctness proof in Isabelle/Isar[6] faithfully follow the definitions

[6] All proof documents are available from `http://argo.matf.bg.ac.rs`.

given in the previous sections. Using this formalization, an effective, operational ML implementation of the DPLL procedure is automatically generated from Isabelle, yielding a formally verified (although not quite efficient) SAT solver that is guaranteed to be correct (Haftmann, 2008). As an example, here we give some fragments of Isabelle/Isar code that formalizes some of the content given in the previous sections.

Definitions 1 and 2:

```
types    Variable  = nat
datatype Literal   = Pos Variable | Neg Variable
```

Definitions 6 and 7:

```
text{* The variable of a literal *}
consts var      :: "Literal => Variable"
primrec
"var (Pos v) = v"
"var (Neg v) = v"

text{* The opposite of a given literal *}
consts opposite :: "Literal => Literal"
primrec
"opposite (Pos v) = (Neg v)"
"opposite (Neg v) = (Pos v)"
```

The DPLL procedure, as defined in Section 4.4:

```
function dpll::"Formula => bool"
where
"(dpll formula) =
 (if (formula = []) then
       True
  else if ([] mem formula) then
       False
  else if (hasPureLiteral formula) then
      (dpll (setLiteralTrue
                (getPureLiteral formula) formula))
  else if (hasUnitLiteral formula) then
      (dpll (setLiteralTrue
                (getUnitLiteral formula) formula))
  else if (dpll (setLiteralTrue
                    (selectLiteral formula) formula)) then
       True
  else
      (dpll (setLiteralTrue
```

```
                (opposite (selectLiteral formula)) formula))
 )"
by pat_completenesss auto
termination
by (relation "measure (% formula. (numLiterals formula))")
   (auto simp add: dpllTermination_1 dpllTermination_2
                   dpllTermination_3 dpllTermination_4)
```

The proof of correctness of the DPLL procedure, corresponding to the outlined proof given in Section 5:

```
lemma dpllCorrectness: "(dpll F) = (satisfiable F)"
proof (induct F rule: dpll.induct)
  case (inductiveStep formula)
  note inductive_hypothesis = this
  show ?case
  proof (cases "formula = []")
    case True
    thus ?thesis
      by (simp add:emptyFormulaIsSatisfiable)
  next
    case False
    show ?thesis
    proof (cases "[] mem formula")
      case True
      with ‘formula ~= []‘ show ?thesis
        by (simp add:formulaWithEmptyClauseIsUnsatisfiable)
    next
      case False
      show ?thesis
      proof (cases "hasPureLiteral formula")
        case True
        let ?pl = "getPureLiteral formula"
        hence "?pl el formula" and "~opposite ?pl el formula"
          by (auto simp add: getPureLiteralIsPure)
        with ‘formula ~= []‘ ‘~[] mem formula‘
             ‘hasPureLiteral formula‘
             inductive_hypothesis
             pureLiteraRule [of "?pl" "formula"]
          show ?thesis
          by auto
      next
        case False
        show ?thesis
        proof (cases "hasUnitLiteral formula")
          case True
          let ?ul = "getUnitLiteral formula"
          hence "[?ul] mem formula"
            by (simp add: getUnitLiteralIsUnit)
          with ‘formula ~= []‘ ‘~[] mem formula‘
```

```
              ‘~hasPureLiteral formula‘ ‘hasUnitLiteral formula‘
              inductive_hypothesis
              unitLiteralRule [of "?ul" "formula"]
          show ?thesis
            by auto
        next
          case False
          with ‘formula ~= []‘ ‘~[] mem formula‘
              ‘~hasPureLiteral formula‘ ‘~hasUnitLiteral formula‘
              inductive_hypothesis
          show ?thesis
            using split_rule[of "formula" "selectLiteral formula"]
            by auto
        qed
      qed
    qed
  qed
qed
```

7. Related Work There is a large and growing body of formalized mathematical knowledge. In this section we briefly overview formalized knowledge and proofs relevant for computer science, especially those formalized in Isabelle, and those relevant for automated reasoning and SAT and SMT solving.

Archive of formal proofs[7] is a collection of proof libraries, examples, and larger scientific developments, mechanically checked in the theorem prover Isabelle. A range of algorithms and data structures have been formalized and verified in Isabelle and similar proof assistant tools. These algorithms include Quicksort, Binary Search, AVL Trees, Binary Search Trees, Depth First Search, Fast Fourier Transform, File Refinement, Cryptographic algorithms (Lindenberg et al., 2006), a range of distributed and parallel algorithms (Disk Paxos, Peterson' s algorithm).

Flaws were detected in many security protocols (e.g., (Li et al., 2007)). Even if security protocols are accompained with correctness proofs, they can still be flawed if these proofs are not formally verifiable (e.g., (Choo, 2006)). Proof assistant tools have been used for formal verification of properties of various protocols (e.g., (Nipkow, 2006, Barsotti et al., 2007)).

A lot of efforts have been invested in verifying programming language semantics and compilers. For example, Klein and Nipkow introduced Jinja (Klein et al., 2006), a Java-like programming language with a formal semantics designed to exhibit core features of the Java language architecture. A model of the language, virtual machine and a compiler are then formally verified. Berghofer described

[7] http://afp.sourceforge.net

a formally verified, fully executable compiler which was extracted from a proof assistant (Berghofer et al., 2003). Blech and Glesner developed a formal semantics for static single assignment (SSA) phase of compilation (Blech et al., 2004). Qian and Xu used iterative abstraction refinement and automated theorem proving for automatically verifying C programs against safety specifications (Qian et al. 2007).

Clark Barrett formally proved correctness of Stanford Framework for Cooperating Decision Procedures, but this proof, although quite detailed, was not verified using a proof assistant (Barret, 2003).

Tom Ridge presented an efficient, mechanically verified sound and complete theorem prover for first order logic (Ridge, 2004). After formalization in Isabelle, OCaml code is generated, yielding a directly executable program.

Chaieb and Nipkow formalized and verified quantifier elimination based decision procedures for Presburger arithmetic (Chaieb et al., 2005).

Isabelle has been combined with different tools to achieve a higher degree of automation. For instance, Weber described integration of SAT solvers zChaff and MiniSat with Isabelle (Weber, 2005, Weber, 2006). Both SAT solvers generate resolution-style proofs of unsatisfiability of their input formulae. These proofs are verified by the theorem prover. Fontaine et al. (Fontaine et al., 2006) used Isabelle to verify the correctness of proof traces generated by the SMT solver Harvey. Barsotti at al. experimented in combining the theorem prover Isabelle with automatic first-order arithmetic provers to increase automation on the verification of distributed protocols (Barsotti et al., 2007). As a case study for the experiment, they verified several clock synchronization algorithms.

Abstract descriptions of the DPLL algorithm and its extensions for ground Satisfiability Modulo Theory (SMT) have been developed. In (Nieuwnhuis et al., 2006, Tinelli, 2002, Krstić et al., 2007), rule based presentations of these algorithms and their informal correctness proofs are given. Informal correctness proofs of the DPLL procedure can be found in many mathematical logic textbooks (e.g., (Davis et al., 1994)). However, as far as we know, our proof is the first formalized correctness proof for the DPLL procedure.

8. Conclusions and Future Work In this paper we presented the first formal proof of correctness of the forty-five years old DPLL algorithm, one of the most fundamental algorithms in computer science. In its history, there were no doubts about the correctness of this algorithm. So, our proof does not resolve a long-standing mystery, but rather: *(i)* it adds to the growing body of formalized, verifiable mathematical knowledge, knowledge that can be verified by independent and reliable proof checkers; *(ii)* it serves as a first building block of formal-

ized correctness proofs for modern SAT and SMT solvers. That task — formally proving correctness of state-of-the-art SAT and SMT solvers, very important for many applications, is in the focus of our current work.

Appendix: One Concrete Implementation In this section we give very simple definitions of the functions used in the DPLL definition given in subsection 4.4. They give a concrete, instantiated procedure and enable obtaining an effectively executable ML implementation. In order to have a more efficient SAT solver, these functions should be defined in a more sophisticated way.

A formula has a unit literal iff it has a clause with only one literal. We define $hasUnitLiteral(F)$ function by primitive recursion.

Definition 19.

$$\begin{aligned} &\neg hasUnitLiteral([\,]) \\ &hasUnitLiteral(c \# F) \Leftrightarrow (|c| = 1) \vee hasUnitLiteral(F) \end{aligned}$$

$getUnitLiteral(F)$ is the first literal l such that $[l] \in F$. It is also defined by primitive recursion.

Definition 20.

$$\begin{aligned} &getUnitLiteral(c \# F) = \\ &\quad \textit{if } |c| = 1 \textit{ then} \\ &\qquad head(c) \\ &\quad \textit{else} \\ &\qquad getUnitLiteral(F) \end{aligned}$$

Procedures that find and select a pure literal from a formula are defined using a series of auxiliary functions.

The functions $literals(F)$ is a list that contains all literals that occur in the formula F. It is defined by primitive recursion.

Definition 21.

$$\begin{aligned} literals([\,]) &= [\,] \\ literals(c \# F) &= c \mathbin{@} literals(F) \end{aligned}$$

The function $hasPureLiteralAux(c_1, c_2)$ checks if there is a literal from the list c_1 whose opposite literal does not occur in the list c_2. It is defined by primitive recursion.

Definition 22.

$$\begin{array}{l}\neg hasPureLiteralAux([\,], c)\\ hasPureLiteralAux(l \# c', c) \;\Leftrightarrow\\ \quad \textit{if } \overline{l} \notin c \textit{ then}\\ \quad\quad \top\\ \quad \textit{else}\\ \quad\quad hasPureLiteralAux(c', c)\end{array}$$

Using this auxiliary function, we define $hasPureLiteral$:

Definition 23.

$$hasPureLiteral(F) \;\Leftrightarrow\; hasPureLiteralAux(literals(F), literals(F))$$

The function $getPureLiteralAux(c_1, c_2)$ finds the literal from the list c_1 whose opposite literal does not occur in the list c_2. It is defined by primitive recursion.

Definition 24.

$$\begin{array}{l}getPureLiteralAux(l \# c', c) \;\Leftrightarrow\\ \quad \textit{if } \overline{l} \notin c \textit{ then}\\ \quad\quad l\\ \quad \textit{else}\\ \quad\quad getPureLiteralAux(c', c)\end{array}$$

Finally, we can define the function $getPureLiteral$.

Definition 25.

$$getPureLiteral(F) = getPureLiteralAux(literals(F), literals(F))$$

$selectLiteral(F)$ is used to select an arbitrary literal of F. For example, it can be the first literal of the first clause of F.

Definition 26.

$$selectLiteral(F) = head(head(F))$$

It was proved that the functions defined in the above way meet the specification given in Section 4.4. These proofs can also be found in `http://argo.matf.bg.ac.rs`, while we don't present them here since this simple implementation is just one of many meeting the required specification.

Acknowledgment. We are grateful to Amine Chaieb and to the anonymous reviewers for useful comments on an earlier version of this paper.

REFERENCES

S. A. Cook (1971). The Complexity of Theorem-Proving Procedures. In *STOC '71: Proceedings of the 3rd Annual ACM Aymposium on Theory of Computing*, ACM Press, pp. 151–158.

M. Davis, H. Putnam (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM* 7(3), pp. 201–215.

M. Davis, G. Logemann, D. Loveland (1962). A Machine Program for Theorem-Proving. *Communications of the ACM* 5(7), pp. 394–397.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik (2001). Chaff: Engineering an Efficient SAT Solver. In *DAC '01: Proceedings of the 38th conference on Design Automation*, ACM Press, pp. 530–535.

N. Eén, N. Sörensson (2003). An Extensible SAT-solver. In *SAT '03: Theory and Applications of Satisfiability Testing*, LNCS 2919, Springer, pp. 502–518.

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

S. Ranise, C. Tinelli (2006). Satisfiability Modulo Theories. *Trends and Controversies - IEEE Intelligent Systems Magazine* 21(6), pp. 71–81.

J. Gu, P. W. Purdom, J. Franco, B. Wah (1997). Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Descrete Mathematics and Theoretical Computer Science 35, AMS, pp. 19–151.

L. Zhang, S. Malik (2002). The Quest for Efficient Boolean Satisfiability Solvers. In *CAV '02: Computer Aided Verification*, LNCS 2404, Springer, pp. 17–36.

D. Le Berre, L. Simon, editors (2005). Special Volume on the SAT 2005 Competitions and Evaluations. *Journal on Satisfiability, Boolean Modeling and Computation* 2.

T. Nipkow, L. C. Paulson, M. Wenzel (2005). *Isabelle HOL: a Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer.

M. Davis, R. Sigal, E. Weyuker (1994). *Computability, Complexity, and Languages (Fundamentals of Theoretical Computer Science)*. Morgan Kaufmann/Academic Press.

S. A. Cook, D. G. Mitchell (1997). Finding Hard Instances of the Satisfiability Problem: A Survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 35, AMS, pp. 1–17.

M. Irgens, W. S. Havens (2004). On Selection Strategies for the DPLL Algorithm, In *Advances in Artificial Intelligence*, LNAI 3060, Springer, pp. 277–291.

L. C. Paulson (1994). *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer.

M. Wenzel (2007). Isabelle/Isar — a Generic Framework for Human-readable Proof Documents, *From Insight to Proof*, Studies in Logic, Grammar and Rethoric 10(23), University of Bialystok, pp. 277–298.

F. Haftmann (2008). Code Generation from Isabelle/HOL Theories. URL: `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

C. Lindenberg, K. Wirt, J. Buchmann (2006). Formal Proof for the Correctness of RSA-PSS. *Cryptology ePrint Archive*, Report 2006/011, `http://eprint.iacr.org/`.

K. R. Choo (2006). On the Security Analysis of Lee, Hwang & Lee (2004) and Song & Kim (2000) Key Exchange/Agreement Protocols. *Informatica* 17(4), pp. 467–480.

C. Li, T. Hwang, N. Lee (2007). Security Flaw in Simple Generalized Group-Oriented Cryptosystem using ElGamal Cryptosystem. *Informatica* 18(1), pp. 61–66.

T. Nipkow (2006). Verifying a Hotel Key Card System. In *ICTAC '06: Theoretical Aspects of Computing*, LNCS 4281, pp. 1–14.

D. Barsotti, L. P. Nieto, A. F. Tiu (2007). Verification of Clocksynchronization Algorithms: Experiments on a Combination of Deductive Tools. *Formal Aspects of Computing* 19(3), pp. 321–341.

G. Klein, T. Nipkow (2006). A Machine-Checked Model for a Java-like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems* 28(4), pp. 619–695.

S. Berghofer, M. Strecker (2003). Extracting a Formally Verified, Fully Executable Compiler from a Proof Assistant. *Electronic Notes in Theoretical Computer Science* 82(2), pp. 33–50.

J. O. Blech, S. Glesner (2004). A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. *Jahrestagung der Gesellschaft für Informatik*, LNI 51, GI, pp. 449–458.

J. Qian, B. Xu (2007). Formal Verification for C Program. *Informatica* 18(2), pp. 289–304.

C. Barrett (2003). *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*, Ph.D. thesis, Stanford University.

P. Fontaine, J.I. Marion, S. Merz, L. P. Nieto, A.F. Tiu (2006). Expressiveness + Automation + Soundness: Towards Combining, SMT Solvers and Interactive Proof Assistants. In *TACAS '06: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3920, pp. 167–181.

T. Ridge (2004). A Mechanically Verified, Efficient, Sound and Complete Theorem Prover for First Order Logic. In *Archive of Formal Proofs*, `http://afp.sf.net`

S. Krstić, A. Goel (2007). Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCoS '07: Frontiers of Combining Systems*, LNCS 4720, Springer, pp. 1–27.

A. Chaieb, T. Nipkow (2005). Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic. In *LPAR '05: Logic for Programming, Artificial Intelligence, and Reasoning*, LNCS 3835, Springer, pp. 367–380.

T. Weber (2005). Using a SAT Solver as a Fast Decision Procedure for Propositional Logic in an LCF-style Theorem Prover. In *TPHOLs '05: Theorem Proving in Higher Order Logics (Emerging Trends)*, research report PRG-RR-05-02, Computing Laboratory, Oxford University, pp. 180–189.

T. Weber (2006). Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL. In *Proceedings of the 6th International Workshop on the Implementation of Logics*, CEUR Workshop Proceedings 212, SUN Site Central Europe, pp. 44–62.

C. Tinelli (2002). A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *JELIA '02: 8th European Conference on Logics in Artificial Intelligence*, LNAI 2424, Springer, pp. 308–319.

Received April 2008

Filip Marić is teaching assistant at the Faculty of Mathematics, University of Belgrade. He graduated in mathematics and received a master's degree in computer science from the University of Belgrade. His main research interests are in formal and automated theorem proving, SAT and SMT solving and object oriented programming.

Predrag Janičić is associate professor and a leader of the Automated Reasoning Group at the Faculty of Mathematics, University of Belgrade. He received his PhD degree in computer science from the University of Belgrade. His main research interests are in automated reasoning, especially in SAT and SMT solving and in intelligent geometrical software.