Engineering DPLL(T) + Saturation

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{leonardo, nbjorner}@microsoft.com

Abstract. Satisfiability Modulo Theories (SMT) solvers have proven highly scalable, efficient and suitable for integrated theory reasoning. The most efficient SMT solvers rely on refutationally incomplete methods for incorporating quantifier reasoning. We describe a calculus and a system that tightly integrates Superposition and DPLL(T). In the calculus, all non-unit ground clauses are delegated to the DPLL(T) core. The integration is tight, dependencies on case splits are tracked as hypotheses in the saturation engine. The hypotheses are discharged during backtracking. The combination is refutationally complete for first-order logic, and its implementation is competitive in performance with E-matching based SMT solvers on problems they are good at.

1 Introduction

SMT solvers based on a DPLL(T) [1] framework have proven highly scalable, efficient and suitable for integrating theory reasoning. However, for numerous applications from program analysis and verification, an integration of decision procedures for the ground fragment is insufficient, as proof obligations often include quantifiers for capturing frame conditions over loops, summarizing auxiliary invariants over heaps, and for supplying axioms of theories that are not already equipped with ground decision procedures. A well known approach for incorporating quantifier reasoning with ground decision procedures is E-matching algorithm used in the Simplify theorem prover [2]. The E-matching algorithm works against an E-graph to instantiate quantified variables. Other state-of-the-art SMT solvers also use E-matching: CVC3 [3], Fx7 [4], Yices [5], and Z3 [6]. Although E-matching is quite effective for some software verification problems, it suffers from several problems: it is not refutationally complete for first-order logic, hints (triggers) are usually required, it is sensitive to the syntactic structure of the formula, and fails to prove formulas that can be easily discharged by saturation based provers.

Equational theorem provers based on Superposition Calculus are strong at reasoning with equalities, universally quantified variables, and Horn clauses. However, these provers do not perform well in the context of software verification [7, 3], as they explore a huge search space generated by a large number of axioms, most of which are irrelevant. The typical software verification problem consists of a set of axioms and a big (mostly) ground formula encoding the data and control flow of the program. This formula is usually a deeply nested and-or tree. Quantified formulas nested in this tree can be extracted by naming them with fresh propositional variables. These problems typically yield large sets with huge non-Horn clauses, which are far from the sweet spot for saturation based provers, but handled well by DPLL(T)-based solvers.

This paper describes a new calculus DPLL($\mathit{\Gamma}$), and an accompanying system, Z3($\mathcal{SP}$), that tightly integrates the strength of saturation based provers in equational reasoning and quantified formulas, with DPLL-based implementations of SMT solvers that are strong in case-analysis and combining theory solvers. The new calculus is very flexible and it can simulate different strategies used in other theorem provers that aim for integrating DPLL and/or theory reasoners as well.

On the technical side, we introduce a key ingredient for this integration; *hypotheses* that track dependencies on case splits when the saturation component performs its deductions. We first lift standard saturation deduction rules to the DPLL($\mathit{\Gamma}$) setting by simply propagating hypotheses (Section 3). It is a somewhat simple, but important, observation that the resulting system is refutationally complete even when ground non-unit clauses are only visible to the DPLL component (but invisible to the inference rules in the saturation component). The lifting becomes less trivial when it comes to using case split literals or any consequence of case splits in deletion and simplification rules. Section 4 presents a lifting that properly tracks case splits into such rules.

On the system side, we discuss the implementation of an instance of DPLL($\mathit{\Gamma}$) in the theorem prover Z3. We believe this is the first report of a really tight integration of DPLL and saturation procedures. Existing systems, to our knowledge, integrate either a black box SMT solver with a saturation prover, [8], or a black box saturation solver with an DPLL core, [9], or don't offer exchanging hypotheses as tightly.

2 Background

We employ basic notions from logic usually assumed in theorem proving. For notation, the symbol $\simeq$ denotes equality; s, u, t are terms; x, y, z are variables; f, g, h, a, b, c are constant or function symbols based on arity; p, q, r are predicate symbols; l is a literal; C and D denote clauses, that is, multi-sets of literals interpreted as disjunctions; $\Box$ is the empty clause; N is a set of clauses; and σ is used for substitutions. A term, literal, or clause is said to be *ground* if it does not contain variables.

We assume that terms are ordered by a simplification ordering $\prec$. It is extended to literals and clauses by a multiset extension. A simplification ordering is well founded and total on ground terms. The most commonly used simplification orderings are instances of the recursive path ordering (RPO) and the Knuth-Bendix ordering (KBO).

An *inference rule* γ is $n+1$-ary relation on clauses, it is written as:

$$\frac{C_1 \quad \ldots \quad C_n}{C}$$

The clauses $C_1, \dots, C_n$ are called premises, and C the conclusion of the inference. If γ is an inference rule, we denote by $\mathsf{C}(\gamma)$ its conclusion, and $\mathsf{P}(\gamma)$ its premises. An *inference system* Γ is a set of inference rules. We assume that each inference rule has a *main premise* that is "reduced" to the conclusion in the context of the other (*side*) premises.

A *proof* of a clause C from the set of clauses N with respect to an inference system Γ is sequence of clauses $C_1, \dots, C_n$, where $C_n = C$ and each clause C_i is either an element of N or the conclusion of an inference rule γ of Γ, where the set of premises is a subset of $\{C_1, \dots, C_{i-1}\}$. A proof of the empty clause $\Box$ is said to be a refutation. An inference system Γ is *refutationally complete* if there is a refutation by Γ from any unsatisfiable set of clauses. A set of clauses N is *saturated* with respect to Γ if the conclusion of any inference by Γ from N is an element of N. Let I be a mapping, called a *model functor*, that assigns to each set of ground clauses N not containing the empty clause an interpretation I_N, called the *candidate model*. If I_N is a model for N, then N is clearly satisfiable. Otherwise, some clause C in N is false in I_N (i.e., C is a counterexample for I_N), then N must contain a *minimal* counterexample with respect to $\prec$. An inference system has the *reduction property for counterexamples*, if for all sets N of clauses and minimal counterexamples C for I_N in N, there is an inference in Γ from N with main premise C, side premises that are true in I_N, and conclusion D that is a smaller counterexample for I_N than C. A clause C is called *redundant* with respect to a set of clauses N if there exists $C_1, \dots, C_n$ in N such that $C_1, \dots, C_n \models C$ and $C_i \prec C$ for all $i \in [1, n]$. A set of clauses N is *saturated up to redundancy* with respect to Γ if the conclusion of any inference by Γ with non redundant premises from N is redundant. If Γ is an inference system that satisfies the reduction property for counterexamples, and N a set of clauses saturated up to redundancy, then N is unsatisfiable if, and only if, it contains the empty clause [10].

2.1 Superposition Calculus

The superposition calculus ($\mathcal{SP}$) [11] is a rewriting-based inference system which is refutationally complete for first-order logic with equality. It is based on a simplification order on terms. Figure 1 contains the $\mathcal{SP}$ inference rules. The inference rules restrict generating inferences to positions in maximal terms of maximal literals. More constraints can be imposed, if a clause C contains negative literals, then it is possible to restrict generating inferences to arbitrarily selected negative literals [11].

3 DPLL(Γ)

We will adapt the proof calculus for DPLL(Γ) from an exposition of DPLL(T) as an abstract transition system [1]. DPLL(Γ) is parameterized by a set of inference rules Γ. States of the transition system are of the form $M \parallel F$, where M is a sequence of *decided and implied literals*, and F a set of *hypothetical clauses*. Intuitively, M represents a partial assignment to ground literals and their justifications. During conflict resolution, we also use states of the form

Equality Resolution

$$\frac{s \not\simeq t \vee C}{\sigma(C)} \qquad \textbf{if} \quad \sigma = mgu(s,t), \;\; \text{for all } l \in C, \; \sigma(s \not\simeq t) \not\prec l$$

Equality Factoring

$$\frac{s \simeq t \vee u \simeq v \vee C}{\sigma(t \not\simeq v \vee u \simeq v \vee C)} \qquad \textbf{if} \quad \begin{cases} \sigma = mgu(s,u), \; \sigma(s) \not\preceq \sigma(t), \\ \text{for all } l \in (u \simeq v \vee C), \; \sigma(s \simeq t) \not\prec l \end{cases}$$

Superposition Right

$$\frac{s \simeq t \vee C \qquad u[s'] \simeq v \vee D}{\sigma(u[t] \simeq v \vee C \vee D)} \qquad \textbf{if} \quad \begin{cases} \sigma = mgu(s,s'), \; s' \text{ is not a variable}, \\ \sigma(s) \not\preceq \sigma(t), \; \sigma(u[s']) \not\preceq \sigma(v), \\ \text{for all } l \in C, \; \sigma(s \simeq t) \not\preceq l \\ \text{for all } l \in D, \; \sigma(u[s'] \simeq v) \not\preceq l \end{cases}$$

Superposition Left

$$\frac{s \simeq t \vee C \qquad u[s'] \not\simeq v \vee D}{\sigma(u[t] \not\simeq v \vee C \vee D)} \qquad \textbf{if} \quad \begin{cases} \sigma = mgu(s,s'), \; s' \text{ is not a variable}, \\ \sigma(s) \not\preceq \sigma(t), \; \sigma(u[s']) \not\preceq \sigma(v), \\ \text{for all } l \in C, \; \sigma(s \simeq t) \not\preceq l \\ \text{for all } l \in D, \; \sigma(u[s'] \not\simeq v) \not\preceq l \end{cases}$$

Fig. 1. Superposition Calculus: Inference Rules

$M \parallel F \parallel C$, where C is a ground clause. A decided literal represents a guess, and an implied literal l_C a literal l that was implied by a clause C. A decided or implied literal in M is said to be an assigned literal. No assigned literal occurs twice in M nor does it occur negated in M. If neither l or $\overline{l}$ occurs in M, then l is said to be undefined in M. We use *lits*(M) to denote the set of assigned literals. We write $M \models_P C$ to indicate that M propositionally satisfies the clause C. If C is the clause $l_1 \vee \ldots \vee l_n$, then $\neg C$ is the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$. A hypothetical clause is denoted by $H \triangleright C$, where H is a set of assigned ground literals (hypotheses) and C is a general clause. The set of hypotheses should be interpreted as a conjunction, and a hypothetical clause $(l_1 \wedge \ldots \wedge l_n) \triangleright (l'_1 \vee \ldots l'_m)$ should be interpreted as $\neg l_1 \vee \ldots \vee \neg l_n \vee l'_1 \vee \ldots \vee l'_m$. The basic idea is to allow the inference rules in Γ to use the assigned literals in M as premises, and hypothetical clauses is an artifact to track the dependencies on these assigned literals. We say the conclusions produced by Γ are hypothetical because they may depend on guessed (decided) literals. We use *clauses*(F) to denote the set $\{C \mid H \triangleright C \in F\}$, and *clauses*$(M \parallel F)$ to denote *clauses*$(F) \cup$ *lits*(M). We also write C instead of $\emptyset \triangleright C$.

The interface with the inference system Γ is realized in the following way: assume γ is an inference rule with n premises, $\{H_1 \triangleright C_1, \ldots, H_m \triangleright C_m\}$ is a set of hypothetical clauses in F, $\{l_{m+1}, \ldots, l_n\}$ is a set of assigned literals in M, and $\mathsf{H}(\gamma)$ denotes the set $H_1 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$, then the inference rule γ is applied to the set of premises $\mathsf{P}(\gamma) = \{C_1, \ldots, C_m, l_{m+1}, \ldots, l_n\}$, and the conclusion $\mathsf{C}(\gamma)$ is added to F as the hypothetical clause $\mathsf{H}(\gamma) \triangleright \mathsf{C}(\gamma)$. Note that the hypotheses of the clauses $H_i \triangleright C_i$ are hidden from the inference rules in Γ.

Definition 1. *The basic DPLL(Γ) system consists of the following rules:*

Decide

$$M \parallel F \Longrightarrow M\ l \parallel F \quad \textbf{if} \begin{cases} l \textit{ is ground}, \\ l \textit{ or } \bar{l} \textit{ occurs in } F, \\ l \textit{ is undefined in } M. \end{cases}$$

UnitPropagate

$$M \parallel F, H \rhd (C \vee l) \Longrightarrow M\ l_{H \rhd (C \vee l)} \parallel F, H \rhd (C \vee l) \quad \textbf{if} \begin{cases} l \textit{ is ground}, \\ M \models_P \neg C, \\ l \textit{ is undefined in } M. \end{cases}$$

Deduce

$$M \parallel F \Longrightarrow M \parallel F, \mathsf{H}(\gamma) \rhd \mathsf{C}(\gamma) \quad \textbf{if} \begin{cases} \gamma \in \Gamma, \\ \mathsf{P}(\gamma) \subseteq \textit{clauses}(M \parallel F), \\ \mathsf{C}(\gamma) \notin \textit{clauses}(F) \end{cases}$$

HypothesisElim

$$M \parallel F, (H \wedge l) \rhd C \Longrightarrow M \parallel F, H \rhd (C \vee \neg l)$$

Conflict

$$M \parallel F, H \rhd C \Longrightarrow M \parallel F, H \rhd C \parallel \neg H \vee C \quad \textbf{if} \ M \models_P \neg C$$

Explain

$$M \parallel F \parallel C \vee \bar{l} \Longrightarrow M \parallel F \parallel \neg H \vee D \vee C \quad \textbf{if} \ l_{H \rhd (D \vee l)} \in M,$$

Learn

$$M \parallel F \parallel C \Longrightarrow M \parallel F, C \parallel C \quad \textbf{if} \ C \notin \textit{clauses}(F)$$

Backjump

$$M\ l'\ M' \parallel F \parallel C \vee l \Longrightarrow M\ l_{C \vee l} \parallel F' \quad \textbf{if} \begin{cases} M \models_P \neg C, \\ l \textit{ is undefined in } M, \\ F' = \{H \rhd C \in F \mid H \cap \textit{lits}(l'\ M') = \emptyset\} \end{cases}$$

Unsat

$$M \parallel F \parallel \square \Longrightarrow \textit{unsat}$$

We say a hypothetical clause $H \rhd C$ is in *conflict* if all literals in C have a complementary assignment. The rule Conflict converts a hypothetical conflict clause $H \rhd C$ into a regular clause by negating its hypotheses, and puts the DPLL(Γ) system in conflict resolution mode. The Explain rule unfolds literals from conflict clauses that were produced by unit propagation. Any clause derived by Explain can be learned, and added to F, because they are logical consequences of the original set of clauses. The rule Backjump can be used to transition the DPLL(Γ) system back from conflict resolution to search mode, it unassigns at least one decided literal (l' in the rule definition). All hypothetical clauses $H \rhd C$ which contain hypotheses that will be unassigned by the Backjump rule are deleted.

Figure 2 contains an example that illustrates DPLL(Γ) instantiated with a Γ that contains only the binary resolution and factoring rules. In this example, we annotate each application of the Deduce rule with the premises for the binary resolution inference rule. In this example, the rule HypothesisElim is used to replace the clause $q(a) \rhd \neg q(y) \vee r(a, y)$ with $\neg q(a) \vee \neg q(y) \vee r(a, y)$ to prevent it from being deleted during backjumping.

Consider the following initial set of clauses:

$$F = \{p(a) \vee p(b),\ \ \neg p(b) \vee p(c),\ \ \neg p(a) \vee q(a),\ \ \neg p(a) \vee q(b),$$
$$\neg r(x,b),\ \ \neg q(x) \vee \neg q(y) \vee r(x,y),\ \ p(x) \vee q(x),\ \ \neg q(a) \vee \neg p(b)\}$$

$\| F$
$\Longrightarrow$ Decide
$p(c) \| F$
$\Longrightarrow$ Decide
$p(c)\ p(a) \| F$
$\Longrightarrow$ UnitPropagate
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)} \| F$
$\Longrightarrow$ Deduce *resolution* $q(a)$ *with* $\neg q(x) \vee \neg q(y) \vee r(x,y)$
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)} \| F,\ q(a) \triangleright \neg q(y) \vee r(a,y)$
$\Longrightarrow$ UnitPropagate
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F,\ q(a) \triangleright \neg q(y) \vee r(a,y)$
$\Longrightarrow$ Deduce *resolution* $q(b)$ *with* $\neg q(y) \vee r(a,y)$
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F,\ q(a) \triangleright \neg q(y) \vee r(a,y),\ q(a) \wedge q(b) \triangleright r(a,b)$
$\Longrightarrow$ HypothesisElim
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| \underbrace{F,\ \neg q(a) \vee \neg q(y) \vee r(a,y),\ q(a) \wedge q(b) \triangleright r(a,b)}_{F'}$
$\Longrightarrow$ Deduce *resolution* $\neg r(x,b)$ *with* $r(a,b)$
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F',\ q(a) \wedge q(b) \triangleright \square$
$\Longrightarrow$ Conflict
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F',\ q(a) \wedge q(b) \triangleright \square \| \neg q(a) \vee \neg q(b)$
$\Longrightarrow$ Explain
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F',\ q(a) \wedge q(b) \triangleright \square \| \neg p(a) \vee \neg q(b)$
$\Longrightarrow$ Explain
$p(c)\ p(a)\ q(a)_{\neg p(a) \vee q(a)}\ q(b)_{\neg p(a) \vee q(b)} \| F',\ q(a) \wedge q(b) \triangleright \square \| \neg p(a)$
$\Longrightarrow$ Backjump
$\neg p(a)_{\neg p(a)} \| F,\ \neg q(a) \vee \neg q(y) \vee r(a,y)$
$\Longrightarrow$ UnitPropagate
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)} \| F,\ \neg q(a) \vee \neg q(y) \vee r(a,y)$
$\Longrightarrow$ Deduce *resolution* $\neg p(a)$ *with* $p(x) \vee q(x)$
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)} \| \underbrace{F,\ \neg q(a) \vee \neg q(y) \vee r(a,y),\ \neg p(a) \triangleright q(a)}_{F''}$
$\Longrightarrow$ UnitPropagate
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \| F''$
$\Longrightarrow$ Conflict
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \| F'' \| \neg q(a) \vee \neg p(b)$
$\Longrightarrow$ Explain
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \| F'' \| p(a) \vee \neg p(b)$
$\Longrightarrow$ Explain
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \| F'' \| p(a)$
$\Longrightarrow$ Explain
$\neg p(a)_{\neg p(a)}\ p(b)_{p(a) \vee p(b)}\ q(a)_{\neg p(a) \triangleright q(a)} \| F'' \| \square$
$\Longrightarrow$ Unsat
unsat

Fig. 2. DPLL(Γ): example

3.1 Soundness and Completeness

We assume that the set of inference rules Γ is sound and refutationally complete. Then, it is an easy observation that all transition rules in DPLL(Γ) preserve satisfiability. In particular, all clauses added by conflict resolution are consequences of the original set of clauses F.

Theorem 1 (Soundness). *DPLL(Γ) is sound.*

From any unsatisfiable set of clauses, a fair application of the transition rules in DPLL(Γ) will eventually generate the *unsat* state. This is a direct consequence of the refutational completeness of Γ. That is, Deduce alone can be used to derive the empty clause without using any hypothesis.

Theorem 2 (Completeness). *DPLL(Γ) is refutationally complete.*

Unrestricted use of the Deduce rule described in Defintion 1 is not very effective, because it allows the inference rules in Γ to use arbitrary ground clauses as premises. We here introduce a refinement of Deduce, called Deduce$^\sharp$, which applies on fewer cases than Deduce, but still maintains refutational completeness. In Deduce$^\sharp$, the set of premises for inference rules in Γ are restricted to non-ground clauses and ground unit clauses. That is, $\mathsf{P}(\gamma) \subseteq \mathit{premises}(M \parallel F)$, where we define $\mathit{nug}(N)$ to be the subset of non unit ground clauses of a set of clauses N, and $\mathit{premises}(M \parallel F) = (\mathit{clauses}(F) \setminus \mathit{nug}(\mathit{clauses}(F))) \cup \mathit{lits}(M)$. The refined rule is then:

Deduce$^\sharp$

$$M \parallel F \implies M \parallel F, \mathsf{H}(\gamma) \triangleright \mathsf{C}(\gamma) \qquad \textbf{if} \begin{cases} \gamma \in \Gamma, \quad \mathsf{P}(\gamma) \subseteq \mathit{premises}(M \parallel F), \\ \mathsf{C}(\gamma) \notin \mathit{clauses}(F) \end{cases}$$

The idea is to use DPLL to handle all case-analysis due to ground clauses. The refined system is called DPLL$(\Gamma)^\sharp$. A state $M \parallel F$ of DPLL$(\Gamma)^\sharp$ is said to be *saturated* if any ground literal in F is assigned in M, there is no ground clause C in $\mathit{clauses}(F)$ such that $M \models_P \neg C$, and if the conclusion of any inference by Γ from $\mathit{premises}(M \parallel F)$ is an element of $\mathit{clauses}(F)$.

Theorem 3. *If $M \parallel F$ is a saturated state of DPLL$(\Gamma)^\sharp$ for an initial set of clauses N and Γ has the reduction property for counterexamples, then N is satisfiable.*

Proof. Since all transitions preserve satisfiability, we just need to show that $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is satisfiable. The set of clauses $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is not saturated with respect to Γ because clauses in $\mathit{nug}(\mathit{clauses}(F))$ were not used as premises for its inference rules, but the set is saturated up to redundancy. Any clause C in $\mathit{nug}(\mathit{clauses}(F))$ is redundant because there is a literal l of C that is in $\mathit{lits}(M)$, and clearly $l \models C$ and $l \prec C$. Since Γ has the reduction property for counterexamples, the set $\mathit{clauses}(F) \cup \mathit{lits}(M)$ is satisfiable.

We assign a *proof depth* to any clause in $\mathit{clauses}(F)$ and literal in $\mathit{lits}(M)$. Intuitively, the proof depth of a clause C indicates the depth of the derivation needed to produce C. More precisely, all clauses in the original set of clauses have proof depth 0. If a clause C is produced using the Deduce rule, and n is the maximum proof depth of the premises, then the proof depth of C is $n+1$. The proof depth of a literal l_C in M is equals to the proof depth of C. If l is a decided literal, and n is the minimum proof depth of the clauses in F that contain l, then the proof depth of l is n. We say DPLL$(\mathit{\Gamma})^\sharp$ is k-bounded if $\mathsf{Deduce}^\sharp$ is restricted to premises with proof depth $< k$. Note that, the number of ground literals that can be produced in a k-bounded run of DPLL$(\mathit{\Gamma})^\sharp$ is finite.

Theorem 4. *A k-bounded DPLL$(\mathit{\Gamma})^\sharp$ always terminates.*

A similar result can be obtained by bounding the term depth. The theorem above is also true for DPLL$(\mathit{\Gamma})$. In another variation of DPLL$(\mathit{\Gamma})$, the restriction on $\mathsf{Deduce}^\sharp$ used in DPLL$(\mathit{\Gamma})^\sharp$ is disabled after k steps. This variation is also refutationally complete, since all transition rules preserve satisfiability and $\mathit{\Gamma}$ is refutationally complete.

3.2 Additional Rules

SMT solvers implement efficient theory reasoning for conjunctions of ground literals. One of the main motivations of DPLL$(\mathit{\Gamma})$ is to use these efficient theory solvers in conjunction with arbitrary inference systems. Theory reasoning is incorporated in DPLL$(\mathit{\Gamma})$ using transition rules similar to the one described in [1]. We use $F \models_T G$ to denote the fact that F *entails* G in theory T.

T-Propagate

$$M \parallel F \quad \Longrightarrow M\ l_{(\neg l_1 \vee \ldots \vee \neg l_n \vee l)} \parallel F \quad \textbf{if} \begin{cases} l \text{ is ground and occurs in } F, \\ l \text{ is undefined in } M, \\ l_1, \ldots, l_n \in \mathit{lits}(M) \\ l_1, \ldots, l_n \models_T l, \end{cases}$$

T-Conflict

$$M \parallel F \quad \Longrightarrow M \parallel F \parallel \neg l_1 \vee \ldots \vee \neg l_n \quad \textbf{if} \begin{cases} l_1, \ldots, l_n \in \mathit{lits}(M), \\ l_1, \ldots, l_n \models_T \mathit{false} \end{cases}$$

4 Contraction Inferences

Most modern saturation theorem provers spend a considerable amount of time simplifying and eliminating redundant clauses. Most of them contain *simplifying* and *deleting inferences*. We say these are *contraction inferences*. Although these inferences are not necessary for completeness, they are very important in practice. This section discusses how contraction inferences can be incorporated into DPLL$(\mathit{\Gamma})$. We distinguish between contraction inferences taking 1 premise, such as deletion of duplicate and resolved literals, tautology deletion, and destructive equality resolution, and rules that take additional clauses besides the one being simplified or eliminated (e.g., subsumption). Assume the contraction rules in a saturation system are described as $\gamma_{d_1}, \gamma_{s_1}, \gamma_d$, and γ_s. Common to these rules is they take a set of clauses F, C and either *delete* C, producing F or *simplify* C to C', producing F, C'. We call C the *main premise*, and other premises are called *side premises*. Thus, we will here be lifting rules of the form to DPLL(Γ):

$$\frac{F, C}{F}\ \gamma_{d_1}(C) \qquad\qquad \frac{F, C, C_2, \ldots, C_n}{F, C_2, \ldots, C_n}\ \gamma_d(C, C_2, \ldots, C_n),\ n \geq 2$$

$$\frac{F, C}{F, C'}\ \gamma_{s_1}(C, C') \qquad\qquad \frac{F, C, C_2, \ldots, C_n}{F, C', C_2, \ldots, C_n}\ \gamma_s(C, C_2, \ldots, C_n, C'),\ n \geq 2$$

Contraction rules Any contraction inference γ_{d_1} or γ_{s_1} that contains only one premise can be easily incorporated into DPLL(Γ). Given a hypothetical clause $H \triangleright C$, the contraction rules are just applied to C. For contraction rules with more than one premise (e.g., subsumption), special treatment is needed. For example, consider the following state:

$$p(a) \parallel p(a) \triangleright p(b), \quad p(b) \vee p(c), \quad p(a) \vee p(b)$$

In this state, the clause $p(b)$ subsumes the clause $p(b) \vee p(c)$, but it is not safe to delete $p(b) \vee p(c)$ because the subsumer has a hypothesis. That is, after backjumping decision literal $p(a)$, $p(a) \triangleright p(b)$ will be deleted and $p(b) \vee p(c)$ will not be subsumed anymore. A naïve solution consists in using the HypothesisElim to transform hypothetical clauses $H \triangleright C$ into regular clauses $\neg H \vee C$. This solution is not satisfactory because important contraction rules, such as demodulation, have premises that must be unit clauses. Moreover, HypothesisElim also has the unfortunate side-effect of eliminating relationships between different hypotheses. For example, in the following state:

$$p(a)\ p(b)_{\neg p(a) \vee p(b)} \parallel p(a) \triangleright p(c), \quad p(b) \triangleright p(c) \vee p(d)$$

it is safe to use the clause $p(c)$ to subsume $p(c) \vee p(d)$ because $p(a) \triangleright p(c)$ and $p(b) \triangleright p(c) \vee p(d)$ will be deleted at the same time during backjumping. To formalize this approach, we assign a *scope level* to any assigned literal in M. The scope level of a literal l ($level(l)$) in $M\ l\ M'$ equals to the number of decision literals in $M\ l$. For example, in the following state:

$$p(a)_{p(a)}\ p(b)\ p(c)_{\neg p(b) \vee p(c)}\ p(d) \parallel \ldots$$

The scope levels of $p(a)$, $p(b)$, $p(c)$ and $p(d)$ are 0, 1, 1 and 2, respectively. The level of a set of literals is the supremum level, thus for a set H, $level(H) = max\{level(l) \mid l \in H\}$. Clearly, if literal l occurs after literal l' in M, then $level(l) \geq level(l')$. In the Backjump rule we go from a state $M\ l'\ M' \parallel F \parallel C \vee l$ to a state $M\ l_{C \vee l} \parallel F'$, and we say the scope level $level(l')$ was *backjumped*.

We now have the sufficient background to formulate how deletion rules with multiple premises can be lifted in DPLL(Γ). Thus, we seek to lift γ_d to a main clause of the form $H \triangleright C$, and side clauses $H_2 \triangleright C_2, \ldots, H_m \triangleright C_m, l_{m+1}, \ldots, l_n$ taken from F and $lits(M)$. The hypothesis of the main clause is H and the hypotheses used in the side clauses are $H_2 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$ (called H' for short). So assume the premise for deletion holds, that is $\gamma_d(C, C_2, \ldots, C_m, l_{m+1}, \ldots, l_n)$. If $\mathit{level}(H) \geq \mathit{level}(H')$, we claim it is safe to delete the main clause $H \triangleright C$. This is so as backjumping will never delete side premises before removing the main premise. Thus, it is not possible to backjump to a state where one of the side premises was deleted from F or M, but a main premise from H is still *alive*: a deleted clause would not be redundant in the new state. In contrast, if $\mathit{level}(H) < \mathit{level}(H')$, then it is only safe to *disable* the clause $H \triangleright C$ until $\mathit{level}(H')$ is backjumped.

A *disabled* clause is not deleted, but is not used as a premise for any inference rule until it is re-enabled. To realize this new refinement, we maintain one array of disabled clauses for every scope level. Clauses that are disabled are not deleted, but moved to the array at the scope level of H'. Clauses are moved from the array of disabled clauses back to the set of main clauses F when backjumping pops scope levels. We annotate disabled clauses as $[H \triangleright C]_k$, where k is the level at which the clause can be re-enabled.

Contraction rules summary We can now summarize how the contraction rules lift to DPLL(Γ). In the contraction rules below, assume $H_2 \triangleright C_2, \ldots, H_m \triangleright C_m \in F$, $l_{m+1}, \ldots, l_n \in \mathit{lits}(M)$, and $H' = H_2 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$.

Delete

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F \qquad \textbf{if} \; \begin{cases} \gamma_d(C, C_2, \ldots, l_n), \; n \geq 2 \\ \mathit{level}(H) \geq \mathit{level}(H') \end{cases}$$

Disable

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, [H \triangleright C]_{\mathit{level}(H')} \qquad \textbf{if} \; \begin{cases} \gamma_d(C, C_2, \ldots, l_n), \; n \geq 2 \\ \mathit{level}(H) < \mathit{level}(H') \end{cases}$$

Simplify

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, (H \cup H') \triangleright C' \qquad \textbf{if} \; \gamma_s(C, C_2, \ldots, l_n, C'), \; n \geq 2$$

We also use Delete_1 and $\mathsf{Simplify}_1$ as special cases (without side conditions) of the general rules for γ_{d_1} and γ_{s_1}.

5 System Architecture

We implemented DPLL(Γ) (and DPLL(Γ)$^\sharp$) in the Z3 theorem prover [6] by instantiating the calculus with the $\mathcal{SP}$ inference rules. Z3 is a state of the art SMT solver which previously used E-matching exclusively for handling quantified formulas. It integrates a modern DPLL-based SAT solver, a core theory solver that handles ground equalities over uninterpreted functions, and *satellite solvers* (for arithmetic, bit-vectors, arrays, etc). The new system is called Z3($\mathcal{SP}$). It uses perfectly shared expressions as its main data structure. This data structure is implemented in the standard way using hash-consing. These expressions are used by the DPLL(T) and $\mathcal{SP}$ engines.

DPLL(T) engine. The rules Decide, UnitPropagate, Conflict, Explain, Learn, Backjump, and Unsat are realized by the DPLL-based SAT solver in Z3($\mathcal{SP}$)'s core. During exploration of a particular branch of the search tree, several of the assigned literals are not relevant. Relevancy propagation [12] keeps track of which truth assignments in M are essential for determining satisfiability of a formula. In our implementation, we use a small refinement where only literals that are marked as relevant have their truth assignment propagated to theory solvers and are made available as premises to the Deduce rule. The rules T-Propagate and T-Conflict are implemented by the congruence closure core and satellite solvers. The congruence closure core processes the assigned literals in M. Atoms range over equalities and theory specific atomic formulas, such as arithmetical inequalities. Equalities asserted in M are propagated by the congruence closure core using a data structure that we will call an E-graph following [2]. Each expression is associated with an E-node that contains the extra fields used to implement the Union-find algorithm, congruence closure, and track references to theory specific data structures. When two expressions are merged, the merge is propagated as an equality to the relevant theory solvers. The core also propagates the effects of the theory solvers, such as inferred equalities that are produced and atoms assigned to true or false. The theory solvers may also produce new ground clauses in the case of non-convex theories. These ground clauses are propagated to F.

$\mathcal{SP}$ engine. The rules Deduce, Deduce$^\sharp$, Delete, Disable and Simplify are implemented by the new $\mathcal{SP}$ engine. It contains a superset of the $\mathcal{SP}$ rules described in Figure 1, that includes contraction rules such as: forward/backward rewriting, forward/backward subsumption, tautology deletion, destructive equality resolution and equality subsumption. Z3($\mathcal{SP}$) implements Knuth-Bendix ordering (KBO) and lexicographical path ordering (LPO). Substitution trees [13] is the main indexing data structure used in the $\mathcal{SP}$ engine. It is used to implement most of the inference rules: forward/backward rewriting, superposition right/left, binary resolution, and unit-forward subsumption. Feature vector indexing [14] is used to implement non-unit forward and backward subsumption. Several inference rules assume that premises have no variables in common. For performance reasons, we do not explicitly rename variables, but use *expression offsets* like the Darwin theorem prover [15]. Like most saturation theorem provers, we store in each clause the set of *parent clauses* (premises) used to infer it. Thus, the set of hypotheses of an inferred clause is only computed during conflict resolution by following the pointers to *parent clauses*. This is an effective optimization because most of the inferred clauses are never used during conflict resolution.

Rewriting with ground equations. Due to the nature of our benchmarks and DPLL(Γ), most of the equations used for forward rewriting are ground. It is wasteful to use substitution trees as an indexing data structure in this case. When rewriting a term t, for every subterm s of t we need to check if there is an equation $s' \simeq u$ such that s is an instance of s'. With substitution trees, this operation may consume $\mathcal{O}(n)$ time where n is the size of s. If the equations are ground, we can store the equations as a mapping G, where $s \mapsto u \in G$ if $s \simeq u$ is ground and $u \prec s$. We can check in constant time if an expression s is a key in the mapping G, because we use perfectly shared terms.

Ground equations. Ground equations are processed by the congruence closure core and $\mathcal{SP}$ engine. To avoid duplication of work, we convert the E-graph into a canonical set of ground rewrite rules using the algorithm described in [16]. This algorithm is attractive in our context because its first step consists of executing a congruence closure algorithm.

E-matching for theories. Although the new system is refutationally complete for first-order logic with equality, it fails to prove formulas containing theory reasoning that could be proved using E-matching. For example, consider the following simple unsatisfiable set of formulas:

$$\neg(f(a) > 2), \quad f(x) > 5$$

DPLL(Γ) fails to prove the unsatisfiability of this set because 2 does not unify with 5. On the other hand, the E-matching engine selects $f(x)$ as a pattern (trigger), instantiates the quantified formula $f(x) > 5$ with the substitution $[x \mapsto a]$, and the arithmetic solver detects the inconsistency. Thus, we still use the E-matching engine in Z3($\mathcal{SP}$). E-matching can be described as:

E-matching

$$M \parallel F, H \triangleright C[t] \Longrightarrow M \parallel F, H \triangleright C[t], \sigma(H \triangleright C[t]) \quad \textbf{if} \quad \begin{cases} E \models \sigma(t) \simeq s, \text{ where} \\ s \text{ is a ground term in } M, \\ E = \{s_1 \simeq r_1, \ldots, s_n \simeq r_n\} \subseteq M \end{cases}$$

Definitions. In software verification, more precisely in Spec♯ [17] and VCC (Verified C Complier), the verification problems contain a substantial number of definitions of the form: $p(\bar{x}) \Leftrightarrow C[\bar{x}]$. Moreover, most of these definitions are irrelevant for a given problem. Assuming $C[\bar{x}]$ is a clause $l_1[\bar{x}] \vee \ldots \vee l_n[\bar{x}]$, the definition is translated into the following set of clauses $Ds = \{\neg p(\bar{x}) \vee l_1[\bar{x}] \vee \ldots \vee l_n[\bar{x}],\ \ p(\bar{x}) \vee \neg l_1[\bar{x}], \ldots, p(\bar{x}) \vee \neg l_n[\bar{x}]\}$. We avoid the overhead of processing irrelevant definitions by using a term ordering that makes the literal containing $p(\bar{x})$ maximal in the clauses Ds.

Search heuristic. A good search heuristic is crucial for a theorem prover. Our default heuristic consists in eagerly applying UnitPropagate, T-Propagate, Conflict and T-Conflict. Before each Decide, we apply k times E-matching and Deduce. The value k is small if there are unassigned ground clauses in F. The E-matching rule is mainly applied to quantified clauses that contains theory symbols. The rule Deduce is implemented using a variant of the *given-clause* algorithm, where the $\mathcal{SP}$ engine state has two sets of clauses: P processed and U unprocessed. Similarly to E [18], the clause selection strategy can use an arbitrary number of priority queues.

Candidate models. Software verification tools such as Spec♯ and VCC expect the theorem prover to provide a model for satisfiable formulas. Realistically, the theorem prover produces only candidate (potential) models. In Z3($\mathcal{SP}$), M is used to produce the candidate model, and a notion of *k-saturation* is used. We say a formula is *k-maybe-sat*, if all ground literals in M are assigned, and Deduce cannot produce any new non redundant clause with proof depth $< k$.

5.1 Evaluation

Benchmarks. We considered three sets of benchmarks: NASA [7], ESC/Java, and Spec♯ [17]. The NASA benchmarks are easy even for SMT solvers based on E-matching. In the 2007 SMT competition[1] the selected NASA benchmarks were all solved in less than one sec by all competitors. The Simplify theorem prover fails on some of these benchmarks because they do not contain hints (triggers), and Simplify uses a very conservative trigger selection heuristic. The ESC/Java and Spec♯ benchmarks are similar in nature, and are substantially more challenging than the NASA benchmarks[2]. These benchmarks contain hints for provers based on E-matching. These hints can be used also by the $\mathcal{SP}$ engine to guide the literal selection strategy. If the trigger (hint) is contained in a negative literal l, we simply select l for generating inferences. Surprisingly, a substantial subset of the axioms can be handled using this approach. When the trigger is contained in a positive literal, Z3($\mathcal{SP}$) tries to make it maximal. This heuristic is particularly effective with the *frame axioms* automatically generated by Spec♯:

$$C[a, a', x, y] \vee \mathit{read}(a, x, y) = \mathit{read}(a', x, y)$$

where $C[a, a', x, y]$ contains 10 literals, and specifies which locations of the heap were not modified. In Spec♯, the heap is represented as a bidimensional array. The Spec♯ benchmarks contain the transitivity and monotonicity axioms:

$$\neg p(x, y) \vee \neg p(y, z) \vee p(x, z), \quad \neg p(x, y) \vee p(f(x), f(y))$$

These axioms are used to model Spec♯'s type system. Each benchmark contains several ground literals of the form $p(s, t)$. The transitivity axiom can be easily "tamed" by selecting one of its negative literals. Unfortunately, this simple approach does not work with the monotonicity axiom. An infinite sequence of irrelevant clauses is produced by the $\mathcal{SP}$ engine. In contrast, these axioms can be easily handled by solvers based on E-matching. At this point, we do not have a satisfactory solution for this problem besides giving preference to clauses with small proof depth. We plan to support ordered chaining [19] in the future.

Inconsistent axioms. A recurrent problem that was faced by Z3 users was sets of axioms that were unintentionally inconsistent. An E-matching based solver usually fails to detect the inconsistency. Even worse, in some cases, small modifications in the formula allowed the prover to transition from failure to success and vice-versa. The following unsatisfiable set of formulas were extracted from one of these problematic benchmarks:

$$a \neq b, \quad \neg p(x, c) \vee p(x, b), \quad \neg p(x, y) \vee x = y, \quad p(x, c)$$

[1] http://www.smtcomp.org

[2] All benchmarks are available at: http://www.smtlib.org.
Z3 runtime logs for NASA's and Spec♯'s benchmarks can be found at: http://research.microsoft.com/users/leonardo/IJCAR2008.

The inconsistency is not detected by E-matching because there is no ground literal using the predicate p. Now, assume the formula also contains the clause $q(a) \vee p(a, b)$. If the prover decides to satisfy this clause by assigning $q(a)$, then $p(a, b)$ is ignored by the E-matching engine and the prover fails. On the other hand, if $p(a, b)$ is selected by the prover, then it is used for instantiation and the proof succeeds. In contrast, Z3($\mathcal{SP}$) can easily detect the inconsistency.

6 Related Work

HaRVey-FOL [9] combines Boolean solving with the equational theorem prover E. Its main application is software verification. The Boolean solver is used to handle the control flow of software systems, E allows haRVey to provide support for a wide range of theories formalizing data-structures. The integration is also loose in this case, for example, a non-unit ground clause produced by E is not processed by the Boolean solver. SPASS+T [8] integrates an arbitrary SMT solver with the SPASS [20] saturation theorem prover. The SMT solver is used to handle arithmetic and free functions. The integration is loose, SPASS uses its deduction rules to produce new formulas as usual, and the ground formulas are propagated to the SMT solver. The ground formulas are processed by both systems. Moreover, the clause splitting available in SPASS cannot be used because SPASS+T has no control over the (black box) SMT solver backtracking search. SMELS [21], to our best understanding, is a method for incorporating theory reasoning in efficient DPLL + congruence closure solvers. The ground clauses are sent to the DPLL solver. The congruence closure algorithm calculates all reduced (dis)equalities. A Superposition procedure then performs an inference rule, which is called Justified Superposition, between these (dis)equalities and the nonground clauses. The resulting ground clauses are provided to the DPLL engine.

SPASS [20] theorem prover supports *splitting*. The idea is to combine the β-rule found in tableau calculi with a saturation based prover. Their formalization relies on the use of labels [22], The main idea is to label clauses with the split levels they depend on. In contrast, the main advantage of the approach used in DPLL(Γ) is better integration with the conflict resolution techniques (i.e., backjumping and lemma learning) used in DPLL-based SAT solvers. In [23], an alternative approach to case analysis for saturation theorem provers is described. The main idea is to introduce new special propositional symbols. This approach has two main disadvantages with respect to DPLL(Γ): the generated clauses can not be directly used for reductions, and efficient conflict resolution techniques used in DPLL-based solvers cannot be directly applied.

7 Conclusion

We have introduced a calculus that tightly integrates inference rules used in saturation based provers with the DPLL(T) calculus. The combination is refutationally complete for first-order logic. The calculus is particulary attractive for software verification because all non-unit ground clauses can be delegated to DPLL(T). We believe this work also provides a better and more flexible infrastructure for implementing the rewrite-based approach for decision procedures [24]. The DPLL(Γ) calculus was implemented in the Z3($\mathcal{SP}$) theorem prover, and the main design decisions were discussed.

References

1. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (2006) 937–977
2. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52** (2005) 365–473
3. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: CADE-21. LNCS (2007)
4. Moskal, M., Lopuszanski, J., Kiniry, J.R.: E-matching for fun and profit. In: Satisfiability Modulo Theories Workshop. (2007)
5. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV'06. LNCS 4144, Springer-Verlag (2006) 81–94
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 08. (2008)
7. Denney, E., Fischer, B., Schumann, J.: Using automated theorem provers to certify auto-generated aerospace software. In: IJCAR'04. Volume 3097 of LNCS. (2004)
8. Prevosto, V., Waldmann, U.: SPASS+T. In: ESCoR Workshop. (2006)
9. Deharbe, D., Ranise, S.: Satisfiability solving for software verification. International Journal on Software Tools Technology Transfer (2008) to appear.
10. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning. MIT Press (2001) 19–99
11. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. MIT Press (2001)
12. de Moura, L., Bjørner, N.: Relevancy Propagation . Technical Report MSR-TR-2007-140, Microsoft Research (2007)
13. Graf, P.: Substitution tree indexing. In: RTA '95. (1995)
14. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: ESFOR Workshop. (2004)
15. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin: A theorem prover for the model evolution calculus. In: ESFOR Workshop. (2004)
16. Gallier, J., Narendran, P., Plaisted, D., Raatz, S., Snyder, W.: An algorithm for finding canonical sets of ground rewrite rules in polynomial time. J. ACM **40** (1993)
17. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec♯ programming system: An overview. In: CASSIS'04. LNCS 3362, Springer (2005) 49–69
18. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15** (2002) 111–126
19. Bachmair, L., Ganzinger, H.: Ordered chaining calculi for first-order theories of transitive relations. J. ACM **45** (1998) 1007–1049
20. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic, D.: SPASS version 2.0. In: CADE-18. Volume 2392 of LNAI. (2002)
21. Lynch, C.: SMELS: Satisfiability Modulo Equality with Lazy Superposition (2007) Presented at the Dagsthul Seminar on Deduction and Decision Procedures.
22. Fietzke, A.: Labelled splitting. Master's thesis, Saarland University (2007)
23. Riazanov, A., Voronkov, A.: Splitting without backtracking. In: IJCAI. (2001)
24. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Transactions on Computational Logic (To appear)