Deciding Effectively Propositional Logic using DPLL and substitution sets

Leonardo de Moura, Ruzica Piskac and Nikolaj Bjørner

August 15, 2008

Technical Report
MSR-TR-2008-104

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Deciding Effectively Propositional Logic using DPLL and substitution sets

Leonardo de Moura, Ruzica Piskac and Nikolaj Bjørner

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

Abstract. We introduce a DPLL calculus that is a decision procedure for the Bernays-Schönfinkel class, also known as EPR. Our calculus allows combining techniques for efficient propositional search with data-structures, such as Binary Decision Diagrams, that can efficiently and succinctly encode finite sets of substitutions and operations on these. In the calculus, clauses comprise of a sequence of literals together with a finite set of substitutions; truth assignments are also represented using substitution sets. The calculus works directly at the level of sets, and admits performing simultaneous constraint propagation and decisions, resulting in potentially exponential speedups over existing approaches.

1 Introduction

Effectively propositional logic, also known as the Bernays-Schönfinkel class, or EPR, of first-order formulas provides for an attractive generalization of pure propositional satisfiability and quantified Boolean formulas. The EPR class comprise of formulas of the form $\exists^*\forall^*\varphi$, where φ is a quantifier-free formula with relations, equality, but without function symbols. The satisfiability problem for EPR formulas can be reduced to SAT by first replacing all existential variables by skolem constants, and then grounding the universally quantified variables by all combinations of constants. This process produces a propositional formula that is exponentially larger than the original. In a matching bound, the satisfiability problem for EPR is NEXPTIME complete [11]. An advantage is that decision problems may be encoded exponentially more succinctly in EPR than with purely propositional encodings [13].

Our calculus aims at providing a bridge from efficient techniques used in pure SAT problems to take advantage of the succinctness provided for by the EPR fragment. One inspiration was [6], which uses an ad-hoc extension of a SAT solver for problems that can otherwise be encoded in QBF or EPR; and we hope the presented framework allows formulating such applications as strategies. A main ingredient is the use of sets of instantiations for both clauses and literal assignments. By restricting sets of instantiations to the EPR fragment, it is feasible to represent these using succinct data-structures, such as Binary Decision Diagrams [4]. Such representations allow delaying, and in several instances, avoiding, space overhead that a direct propositional encoding would entail.

The main contributions of this paper comprise of a calculus DPLL($\mathcal{SX}$) with substitution sets which is complete for EPR (Section 2). The standard calculus for propositional satisfiability lifts directly to DPLL($\mathcal{SX}$), allowing techniques from SAT solving to apply on purely propositional clauses. However, here, we will be mainly interested in investigating features specific to non-propositional cases. We make explicit and essential use of *factoring*, well-known from first-order resolution, but to our knowledge so far only used for clause simplifications in other liftings of DPLL. We show how the calculus lends itself to an efficient search strategy based on a simultaneous version of Boolean constraint propagation (Section 3). We exhibit cases where the simultaneous technique may produce an exponential speedup during propagation. By focusing on *sets of* substitutions, rather than substitutions, we open up the calculus to efficient implementations based on data-structures that can encode finite sets succinctly. Our current prototype uses a BDD package for encoding finite domains, and we report on a promising, albeit preliminary, empirical evaluation (Section 4). Section 5 concludes with related work and future extensions.

2 The DPLL($\mathcal{SX}$) Calculus

2.1 Preliminaries

We use $a, b, c, \triangle, \star, 0, \ldots$ to range over a finite alphabet Σ of constants, while $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ are tuples of constants, $x, y, z, x_0, x_1, x_2, \ldots$ for variables from a set $\mathcal{V}$, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are tuples of variables, and $p_1, p_2, p, q, r, s, t, \ldots$ for atomic predicates of varying arities. Signed predicate symbols are identified by the set $\mathcal{L}$. As usual, literals (identified by the letter ℓ) are either atomic predicates or their negations applied to arguments. For example, $\overline{p}(x_1, x_2)$ is the literal where the binary atomic predicate p is negated. Clauses consist of a finite set of literals, where each atomic predicate is applied to distinct variables. For example $p(x_1, x_2) \vee \overline{q}(x_3) \vee \overline{q}(x_4)$ is a (well formed) clause. We use C, C', C_1, C_2 to range over clauses. The empty clause is identified by a $\Box$. Substitutions, written θ, θ', are idempotent partial functions from $\mathcal{V}$ to $\mathcal{V} \cup \Sigma$. In other words, $\theta \in \mathcal{V} \overset{\sim}{\to} \mathcal{V} \cup \Sigma$ is a substitution if for every $x \in \mathbf{Dom}(\theta)$ we have $\theta(x) \in \mathcal{V} \;\Rightarrow\; \theta(x) \in \mathbf{Dom}(\theta) \wedge \theta(\theta(x)) = \theta(x)$. Substitutions are lifted to homomorphisms in the usual way, so that substitutions can be applied to arbitrary terms, e.g., $\theta(f(c, x)) = f(c, \theta(x))$. Substitutions that map to constants only (Σ) are called *instantiations*. We associate with each substitution θ a set of instances, namely, $\mathit{instancesOf}(\theta) = \{\theta' \in (\mathbf{Dom}(\theta) \to \Sigma) \mid \forall x \in \mathbf{Dom}(\theta) \;.\; \theta'(x) = \theta'(\theta(x))\}$. Suppose θ is a substitution with domain $\{x_1, \ldots, x_n\}$; then the corresponding set $\mathit{instancesOf}(\theta)$ can be viewed as a set of n-entry records, or equivalently as an n-ary relation over $x_1, \ldots, x_n$. In the following, we can denote a set of substitutions as a set of instances, but will use the terminology *substitution set* to reflect that we use representations of such sets using a mixture of instantiations, which map variables to constants, as well as substitutions that map variables to variables.

We discuss in more detail how substitution sets are represented in Section 3.3. As shorthand for *sets* of substitutions we use Θ, $\Theta', \Theta_1, \Theta_2$. A substitution-set constrained clause is a pair $C \cdot \Theta$ where C is a clause and Θ is a substitution set. The denotation of a constrained clause is the set of ground clauses $\theta(C)$, for every instantiation $\theta \in \Theta$. Notice that there are no ground clauses associated with $C \cdot \emptyset$, so our rules will avoid adding such tautologies. We will assume clauses are in "normal" form where the literals in C are applied to different variables, so it is up to the substitutions to create equalities between variables.

Literals can also be constrained, and we use the notation $\ell\Theta$ for the literal ℓ whose instances are determined by the non-empty Θ. If Θ is a singleton set, we may just write the instance of the literal directly. So for example $p(a)$ is shorthand for $p(x)\{a\}$.

Example 1 (Constrained clauses). The set of (unit) clauses: $p(a,b), p(b,c), p(c,d)$ can be represented as the set-constrained clause

$$p(x_1, x_2) \cdot \{[x_1 \mapsto a, x_2 \mapsto b], [x_1 \mapsto b, x_2 \mapsto c], [x_1 \mapsto c, x_2 \mapsto d]\},$$

or simply: $p(x_1, x_2) \cdot \{(a,b), (b,c), (c,d)\}$

A context Γ is a sequence of constrained literals and decision markers ($\diamond$). For instance, the sequence $\ell_1\Theta_1, \diamond, \ell_2\Theta_2, \ell_3\Theta_3, \ldots, \ell_k\Theta_k$ is a context. We allow concatenating contexts, so $\Gamma, \Gamma', \ell\Theta, \diamond, \ell'\Theta', \Gamma''$ is a context that starts with a sequence of constrained literals in Γ, continues with another sequence Γ', contains $\ell\Theta$, a decision marker, then $\ell'\Theta'$, and ends with Γ''.

Operations from relational algebra will be useful in manipulating substitution sets. See also [17], [16] and [18]. We summarize the operations we will be using below:

Selection $\sigma_{\varphi(\boldsymbol{x})}\Theta$ is shorthand for $\{\theta \in \Theta \mid \varphi(\theta(\boldsymbol{x}))\}$.

Projection $\pi_{\boldsymbol{x}}\Theta$ is shorthand for the set of substitutions obtained from Θ by removing domain elements other than $\boldsymbol{x}$. For example, $\pi_x\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} = \{[x \mapsto a]\}$.

Co Projection $\hat{\pi}_{\boldsymbol{x}}\Theta$ is shorthand for the set of substitutions obtained from Θ by removing $\boldsymbol{x}$. So $\hat{\pi}_x\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} = \{[y \mapsto b], [y \mapsto c]\}$.

Join $\Theta \bowtie \Theta'$ is the natural join of two relations. If Θ uses the variables $\boldsymbol{x}$ and $\boldsymbol{y}$, and Θ' uses variables $\boldsymbol{x}$ and $\boldsymbol{z}$, where $\boldsymbol{y}$ and $\boldsymbol{z}$ are disjoint, then $\Theta \bowtie \Theta'$ uses $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$ and is equal to $\{\theta \mid \hat{\pi}_{\boldsymbol{z}}(\theta) \in \Theta, \hat{\pi}_{\boldsymbol{y}}(\theta) \in \Theta'\}$. For example, $\{[x \mapsto a, y \mapsto b], [x \mapsto a, y \mapsto c]\} \bowtie \{[y \mapsto b, z \mapsto b], [y \mapsto b, z \mapsto a]\} = \{[x \mapsto a, y \mapsto b, z \mapsto b], [x \mapsto a, y \mapsto b, z \mapsto a]\}$.

Renaming $\delta_{\boldsymbol{x} \to \boldsymbol{y}}\Theta$ is the relation obtained from Θ by renaming the variables $\boldsymbol{x}$ to $\boldsymbol{y}$. We here assume that $\boldsymbol{y}$ is not used in Θ already.

Restriction $\Theta \wr \theta$ restricts the set Θ to the substitution θ. It is shorthand for a sequence of selection and co-projections. For example, $\Theta \wr [x \mapsto a] = \hat{\pi}_x \sigma_{x=a}\Theta$, and $\Theta \wr [x \mapsto y, y \mapsto y] = \hat{\pi}_x \sigma_{x=y}\Theta$. More generally, $\Theta \wr \theta$ is $\hat{\pi}_{\boldsymbol{x}} \sigma_{\boldsymbol{x}=\theta(\boldsymbol{x})}\Theta$ where $\boldsymbol{x}$ is the subset of the domain of θ where θ is not idempotent. Thus, if $\boldsymbol{x} = \{x_1, \ldots, x_n\}$, then $\theta(x_1) \neq x_1, \ldots, \theta(x_n) \neq x_n$.

Set operations $\Theta \cup \Theta'$ creates the union of Θ and Θ', both sets of n-ary tuples (sets of instances with n variables in the domain). Subtraction $\Theta \setminus \Theta'$ is the set $\{\theta \in \Theta \mid \theta \notin \Theta'\}$. The complement $\overline{\Theta}$ is $\Sigma^n \setminus \Theta$.

It is important to point out here that we distinguish between $\emptyset$ and $\{[]\}$. While $\emptyset$ represents the empty set, $\{[]\}$ represents a substitution set with the empty domain. We illustrate the difference on the following example: let $\Theta_1 = \{[x \mapsto a, y \mapsto b]\}$ and $\Theta_2 = \{[y \mapsto c]\}$. Their join evaluates to $\Theta_1 \bowtie \Theta_2 = \emptyset$ while $\hat{\pi}_y \Theta_2 = \{[]\}$. In the Example 3 the difference between $\emptyset$ and $\{[]\}$ plays an important role in proving unsatisfiability of a formula.

Notice, that we can compute the image of a most general unifier of two substitutions θ and θ', by taking the natural join of their substitution set equivalents (the join is the empty set if the most general unifier does not exist). That is, if σ is the most general unifier of the two substitutions θ and θ', then $\mathit{instancesOf}(\sigma \circ \theta) = \mathit{instancesOf}(\theta) \bowtie \mathit{instancesOf}(\theta')$. In more detail, let us recall the definitions for composing substitutions and computing most general unifiers of subistutions. Let $\theta_1 = [x_1 \mapsto t_1, ..., x_n \mapsto t_n]$ and let $\theta_2 = [y_1 \mapsto s_1, ..., y_m \mapsto s_m]$ be two substitutions. Their composition $\theta_2 \circ \theta_1$ is a substitution obtained by applying θ_2 to t_i and adding entries in the domain of θ_2 that are not in the domain of θ_1. As an example, consider $\theta_1 = [x_1 \mapsto a, x_2 \mapsto y_2]$ and $\theta_2 = [x_1 \mapsto b, y_1 \mapsto z, y_2 \mapsto b]$. Then, $\theta_2 \circ \theta_1 = [x_1 \mapsto a, x_2 \mapsto b, y_1 \mapsto z, y_2 \mapsto b]$. Having two substitutions θ_1 and θ_2, a unifier of θ_1 and θ_2 is a substitution σ such that $\sigma \circ \theta_1 = \sigma \circ \theta_2$. In order to define the most general unifier (mgu), we introduce the ordering on the set of substitutions: $\theta_1 \preceq \theta_2$ iff $\mathit{instancesOf}(\theta_2) \subseteq \mathit{instancesOf}(\theta_1)$ (or equivalently, if there is a substitution σ, such that $\sigma \circ \theta_1 = \theta_2$). The most general unifier for substitutions θ_1 and θ_2 is a unifier σ such that there is no unifier σ' such that $\sigma' \prec \sigma$. Consider substitutions $\theta_1 = [x \mapsto z, y \mapsto b]$ and $\theta_2 = [x \mapsto a, y \mapsto b]$. We calculate the mgu as $\text{mgu}(\theta_1, \theta_2) = [z \mapsto a]$.

If two clauses $C \vee \ell(\boldsymbol{x})$ and $C' \vee \neg\ell(\boldsymbol{x})$, have substitution sets Θ and Θ' respectively, we can compute the resolvent $(C \vee C') \cdot \hat{\pi}_x(\Theta \bowtie \Theta')$. We quietly assumed that the variables in C and C' were disjoint and renamed apart from $\boldsymbol{x}$. If they are not disjoint, they can easily be made disjoint. We illustrate the technique on the following example: let $(p(x) \vee q(x)) \cdot \{[x \mapsto a], [x \mapsto b]\}$ and $(\overline{p}(x) \vee r(x)) \cdot \{[x \mapsto b], [x \mapsto c]\}$ be two clauses and let y and z be two fresh variables. Then, the given clauses are equivalent to $(p(x) \vee q(y)) \cdot \{[x \mapsto a], [x \mapsto b]\} \bowtie \{[y \mapsto x]\}$ and $(\overline{p}(x) \vee r(z)) \cdot \{[x \mapsto b], [x \mapsto c]\} \bowtie \{[z \mapsto x]\}$. After evaluating join operations, we obtain clauses $(p(x) \vee q(y)) \cdot \{(a, a), (b, b)\}$ and $(\overline{p}(x) \vee r(z)) \cdot \{(b, b), (c, c)\}$. Their resolvent is $(q(y) \vee r(z)) \cdot \{(b, b)\}$ which corresponds to the above rule.

In the rest of the paper, we assume that variables are by default disjoint, or use appropriate renaming silently instead of cluttering the notation.

2.2 Inference rules

We will adapt the proof calculus for DPLL($\mathcal{SX}$) from an exposition of DPLL($\mathcal{T}$) as an abstract transition system [12, 10]. States of the transition system are of the form $\Gamma \parallel F$, where Γ is a context, and the set F is a collection of constrained clauses. Constrained literals are furthermore annotated with optional *explanations.* In this presentation we will use one kind of explanation of the form:

$\ell^{C \cdot \Theta}\Theta'$ All Θ' instances of ℓ are implied by propagation from $C \cdot \Theta$

We maintain the following invariant, which is crucial for conflict resolution:

Invariant 1. *For every derived context of the form $\Gamma, \ell^{C \cdot \Theta}\Theta', \Gamma'$ it is the case that $C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x}))$ and there are assignments $\overline{\ell}_i\Theta_i \in \Gamma$, such that $\Theta' \subseteq \pi_{\boldsymbol{x}}(\Theta \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_1}\Theta_1 \bowtie \ldots \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_k}\Theta_k)$ (written $\Theta' \subseteq \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k)$ without the explicit renamings). Furthermore, Θ' is non-empty.*

Example 2 (Establishing the invariant). We show how to establish the invariant on the following example. Consider a clause $C_0 = p(x) \vee q(x) \vee r(x)$ and a context Γ that contains literals $\overline{p}(a)$ and $\overline{q}(a)$. Standard unit propagation rule infers that $r(a)$ has to be added to the context Γ. We formulate that in the previously introduced notation. First, we rename variables and obtain a substitution-set constrained clause $C_1 = C \cdot \Theta = p(x_1) \vee q(x_2) \vee r(x_3) \cdot \{[x_1 \mapsto x_3, x_2 \mapsto x_3, x_3 \mapsto x_3]\}$ and then, using a unit propagation rule, we add the literal $r(x_3)^{C \cdot \Theta}\Theta'$ to a context Γ (and $\Theta' = \{[x_3 \mapsto a]\}$). In order to establish the invariant, we define substitution sets $\Theta_1 = \{[x_1 \mapsto a]\}$ and $\Theta_2 = \{[x_2 \mapsto a]\}$: they are satisfying that $\overline{p}(x_1)\Theta_1 \in \Gamma$ and $\overline{q}(x_2)\Theta_2 \in \Gamma$. To verify that Θ_1 and Θ_2 are good choices for substitution sets, first we calculate $\Theta \bowtie \Theta_1 \bowtie \Theta_2 = \{[x_1 \mapsto a, x_2 \mapsto a, x_3 \mapsto a]\}$. Projecting x_3 in this substitution-set results with $\pi_{x_3}(\Theta \bowtie \Theta_1 \bowtie \Theta_2) = \{[x_3 \mapsto a]\} = \Theta'$, i.e. the invariant is established.

We take advantage of this invariant to introduce a notion of *premises*$(\ell^{C \cdot \Theta}\Theta')$ that extracts $\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k$ (or, to spell it out for the last time: $\Theta \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_1}\Theta_1 \bowtie \ldots \bowtie \delta_{\boldsymbol{x} \to \boldsymbol{x}_k}\Theta_k$). For a clause $C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x}))$ and a context Γ, one can think about *premises*$(\ell^{C \cdot \Theta}\Theta')$ as a set of tuples that make literals $\ell_1, \ldots, \ell_k$ false and $\ell(\boldsymbol{x})$ true in Γ.

During conflict resolution, we also use states of the form

$\Gamma \parallel F \parallel C \cdot \Theta, \Theta_r$

where $C \cdot \Theta$ is a conflict clause, and Θ_r is the set of instantiations that falsify C; it is used to guide conflict resolution.

We split the presentation of DPLL($\mathcal{SX}$) into two parts, consisting of the search inference rules, given in Fig. 1, and the conflict resolution rules, given in Fig. 2. Search proceeds similarly to propositional DPLL: It starts with the initial state $\parallel F$ where the context is empty. The result is either unsat indicating that F is unsatisfiable, or a state $\Gamma \parallel F$ such that every clause in F is satisfied by Γ. A sequence of Decide steps are used to guess an assignment of truth values to the literals in the clauses F. The side-conditions for UnitPropagate and Conflict are similar: they check that there is a non-empty join of the clause's substitutions with substitutions associated with the complemented literals. There is a conflict when all of the literals in a clause has complementary assignments, otherwise, if

$$\textsf{Decide}\ \dfrac{\ell \in F \quad \text{If } \overline{\ell}\Theta' \in \Gamma \text{ or } \ell\Theta' \in \Gamma, \text{ then } \Theta \bowtie \Theta' = \emptyset}{\Gamma \parallel F \implies \Gamma, \diamond, \ell\Theta \parallel F}$$

$$\textsf{Conflict}\ \dfrac{C = (\ell_1 \vee \ldots \vee \ell_k),\ \overline{\ell}_i\Theta_i \in \Gamma, \quad \Theta_r = \Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k \neq \emptyset}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma \parallel F, C \cdot \Theta \parallel C \cdot \Theta, \Theta_r}$$

$$\textsf{UnitPropagate}\ \dfrac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})),\ \overline{\ell}_i\Theta_i \in \Gamma, i = 1, .., k \\ \Theta' = \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k) \setminus \bigcup\{\Theta_\ell \mid \ell\Theta_\ell \in \Gamma\} \neq \emptyset \\ \Theta' \bowtie \bigcup\{\Theta_{\overline{\ell}} \mid \overline{\ell}\Theta_{\overline{\ell}} \in \Gamma\} = \emptyset \end{array}}{\Gamma \parallel F, C \cdot \Theta \implies \Gamma, \ell^{C \cdot \Theta} \cdot \Theta' \parallel F, C \cdot \Theta}$$

Fig. 1. Search inference rules

all but one literal has a complementary assignment, we may apply unit propagation to the remaining literal. The last side condition on UnitPropagate prevents it from being applied if there is a conflict. Semantically, a non-empty join implies that there are instances of the literal assignments that contradict (all but one of) the clause's literals.

$$\textsf{Resolve}\ \dfrac{\begin{array}{c} \delta_{\boldsymbol{y} \to \boldsymbol{x}}\pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell = \emptyset \ \text{ for every } \ell(\boldsymbol{y}) \in C',\ C_\ell = (C(\boldsymbol{y}) \vee \overline{\ell}(\boldsymbol{x})) \\ \Theta'_r = \hat{\pi}_{\boldsymbol{x}}(\Theta_r \bowtie \Theta_\ell \bowtie \mathit{premises}(\overline{\ell}\Theta_\ell)) \neq \emptyset, \quad \Theta'' = \hat{\pi}_{\boldsymbol{x}}(\Theta \bowtie \Theta') \end{array}}{\Gamma, \overline{\ell}^{C_\ell \cdot \Theta'}\Theta_\ell \parallel F \parallel (C'(\boldsymbol{z}) \vee \ell(\boldsymbol{x})) \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel (C(\boldsymbol{y}) \vee C'(\boldsymbol{z})) \cdot \Theta'', \Theta'_r}$$

$$\textsf{Skip}\ \dfrac{\delta_{\boldsymbol{y} \to \boldsymbol{x}}\pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell = \emptyset \ \text{ for every } \ell(\boldsymbol{y}) \in C}{\Gamma, \overline{\ell}^{C_\ell \cdot \Theta'}\Theta_\ell \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel C \cdot \Theta, \Theta_r}$$

$$\textsf{Factoring}\ \dfrac{\Theta'_r = \hat{\pi}_{\boldsymbol{z}}\sigma_{\boldsymbol{y}=\boldsymbol{z}}\Theta_r \neq \emptyset, \quad \Theta' = \hat{\pi}_{\boldsymbol{z}}\sigma_{\boldsymbol{y}=\boldsymbol{z}}\Theta}{\Gamma \parallel F \parallel (C(\boldsymbol{x}) \vee \ell(\boldsymbol{y}) \vee \ell(\boldsymbol{z})) \cdot \Theta, \Theta_r \implies \Gamma \parallel F \parallel (C(\boldsymbol{x}) \vee \ell(\boldsymbol{y})) \cdot \Theta', \Theta'_r}$$

$$\textsf{Learn}\ \dfrac{C \cdot \Theta \notin F}{\Gamma \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma \parallel F, C \cdot \Theta \parallel C \cdot \Theta, \Theta_r}$$

$$\textsf{Unsat}\ \Gamma \parallel F \parallel \Box \cdot \Theta, \Theta_r \implies \textsf{unsat} \qquad \textbf{if } \Theta \neq \emptyset$$

$$\textsf{Backjump}\ \dfrac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})),\ \overline{\ell}_i\Theta_i \in \Gamma_1 \\ \Theta' = \pi_{\boldsymbol{x}}(\Theta \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k) \setminus \bigcup\{\Theta_\ell \mid \ell\Theta_\ell \in \Gamma_1\} \neq \emptyset \end{array}}{\Gamma_1, \diamond, \Gamma_2 \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma_1, \ell^{C \cdot \Theta}\Theta' \parallel F}$$

$$\textsf{Refine}\ \Gamma, \diamond, \ell\Theta_1, \Gamma' \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma, \diamond, \ell\Theta'_1 \parallel F \qquad \textbf{if } \emptyset \neq \Theta'_1 \subset \Theta_1$$

Fig. 2. Conflict resolution rules

Conflict resolution rules, shown in Fig. 2, produce resolution proof steps based on a clause identified in a conflict. The Resolve rule unfolds literals from conflict clauses that were produced by unit propagation and the rule Skip bypasses propagations that were not used in the conflict. The preconditions of Resolve only apply if there is a single literal that is implied by the top of the context, if that is not the case, we can use factoring. Unlike propositional DPLL, it is not always possible to apply factoring on repeated literals in a conflict clause. We therefore include a Factoring rule to explicitly handle factoring when it applies. Any clause derived by resolution or factoring can be learned using Learn, and added to the clauses in F. The inference system produces the result unsat if conflict resolution results in the empty clause. There are two ways to transition from conflict resolution to search mode. Back-jumping applies when all but one literal in the conflict clause is assigned below the current decision level. In this case the rule Backjump adds the uniquely implied literal to the logical context Γ and resumes search mode. As factoring does not necessarily always apply, we need another rule, called Refine, for resuming search. The Refine rule allows refining the set of substitutions applied to a decision literal. The side condition to Refine only requires that Θ_1 be a non-empty, non-singleton set. In some cases we can use the conflict clause to guide refinement: If C contains two occurrences $\overline{\ell}(\boldsymbol{x}_1)$ and $\overline{\ell}(\boldsymbol{x}_2)$, where $\pi_{\boldsymbol{x}_1}(\Theta_r)$ and $\pi_{\boldsymbol{x}_2}(\Theta_r)$ are disjoint but are subsets of Θ_1, then use one of the projections as Θ_1'.

To illustrate use of the inference rules we derive a proof that a set containing $p(a)$ and $\overline{p}(a)$ is unsatisfiable.

Example 3 (Unsatisfiable set of clauses). Let $C_1\Theta_1 = p(x) \cdot \{a\}$ and $C_2\Theta_2 = \overline{p}(x) \cdot \{a\}$ and let $F = C_1, C_2$. Then the proof for unsatisfiability of F can be derived as follows:

$$\begin{aligned}
&\quad \| F \\
\Longrightarrow\ & \mathsf{UnitPropagate} \\
&\quad p(x)^{p(x)\{a\}}\{a\} \,\|\, F \\
\Longrightarrow\ & \mathsf{Conflict} \\
&\quad p(x)^{p(x)\{a\}}\{a\} \,\|\, F \,\|\, \overline{p}(x)\{a\}, \{a\} \\
\Longrightarrow\ & \mathsf{Resolve} \\
&\quad p(x)^{p(x)\{a\}}\{a\} \,\|\, F \,\|\, \Box \cdot \{[]\}, \{[]\} \\
\Longrightarrow\ & \mathsf{unsat}
\end{aligned}$$

The Resolve step is enabled because we distinguish $\emptyset$ between $\{[]\}$. Here we see that the difference between $\emptyset$ and $\{[]\}$ is not artificially introduced: $\{[]\}$ can be seen as a binding for $\Box$.

The next example is more complex and illustrates in particular the use of factoring and splitting:

Example 4 (Factoring). Assume we have the clauses:

$$F : \begin{cases} C_1 : \overline{p}(x) \ \vee\ q(y) \ \vee\ \overline{r}(z) \cdot \{(a,a,a)\}, \\ C_2 : \overline{p}(x) \ \vee\ s(y) \ \vee\ \overline{t}(z) \cdot \{(b,b,b)\}, \\ C_3 : \overline{q}(x) \ \vee\ \overline{s}(y) \cdot \{(a,b)\} \end{cases}$$

A possible derivation may start with the empty assignment and take the shape shown in Fig 3. We end up with a conflict clause with two occurrences

$\parallel F$
$\Longrightarrow$ Decide
$\diamond, r(x)\{a,b,c\} \parallel F$
$\Longrightarrow$ Decide
$\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\} \parallel F$
$\Longrightarrow$ Decide
$\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\} \parallel F$
$\Longrightarrow$ UnitPropagate using C_1
$\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\}, q(x)\{a\} \parallel F$
$\Longrightarrow$ UnitPropagate using C_2
$\underbrace{\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{a,b\}, q(x)\{a\}, s(x)\{b\}}_{\Gamma} \parallel F$
$\Longrightarrow$ Conflict using C_3
$\Gamma \parallel F \parallel \overline{q}(x) \;\vee\; \overline{s}(y) \cdot \{(a,b)\}$
$\Longrightarrow$ Resolve using C_2
$\Gamma \parallel F \parallel \overline{p}(x) \vee \overline{q}(y) \vee \overline{t}(z) \cdot \{(b,a,b)\}$
$\Longrightarrow$ Resolve using C_1
$\Gamma \parallel F \parallel \overline{p}(x) \vee \overline{p}(x') \vee \overline{r}(y) \vee \overline{t}(z) \cdot \{(b,a,a,b)\}$

Fig. 3. Example derivation steps

of $\overline{p}$. Factoring does not apply, because the bindings for x and x' are different. Instead, we can choose one of the bindings as the new assignment for p. For example, we could take the subset $\{b\}$, in which case the new stack is:

$\Longrightarrow$ Refine
$\diamond, r(x)\{a,b,c\}, \diamond, t(x)\{b,c\}, \diamond, p(x)\{b\} \parallel F$

2.3 Soundness, Completeness and Complexity

It can be verified by inspecting each rule that all clauses added by conflict resolution are resolvents of the original set of clauses F. Thus,

Theorem 1 (Soundness). *DPLL($\mathcal{SX}$) is sound.*

The use of premises and the auxiliary substitution Θ_r have been delicately formulated to ensure that conflict resolution is finite and stuck-free, that is, Resolve and Factoring always produce a conflict clause that admits either Backjump or Refine.

Lemma 1 (Preserving Invariant). *Each rule of DPLL($\mathcal{SX}$) preserves Invariant 1.*

Proof. Literals annotated with explanations can only be added to a context using the Backjump or UnitPropagate rules. Since a rule can only be applied only if its preconditions are enabled, whenever a literal annotated with explanations is added to a context, preconditions for Backjump or UnitPropagate hold. They directly ensure that the invariant is preserved.

Theorem 2 (Stuck-freeness). *For every derivation starting with rule* Conflict *there is a state $\Gamma \,\|\, F \,\|\, C \cdot \Theta, \Theta_r$, such that* Backjump *or* Refine *is enabled.*

Proof. Our goal is to show that the proof derivation cannot be stuck in a conflict mode. We aim to prove that if a derivation is in a conflict resolution mode, then there is a state where the preconditions for Backjump or Refine are enabled and a derivation can transition to a search mode. Note that Refine is always enabled when there is a non-singleton set attached to a decision literal, so for the rest of the proof we assume that associated to every decision literal has a singleton binding set. We prove the theorem by induction on the number of constrained literals annotated with explanations.

Let us assume that a context Γ contains no literals with explanations, i.e. all literals are decision literals. If a proof derivation starts with the Conflict rule and C is a conflicting clause, Backjump rule is enabled immediately in the next step because all its preconditions hold. It can easily be verified using the fact that C is a conflicting clause. If a clause C causes a conflict, then it satisfies certain properties (preconditions of Conflict), which directly imply preconditions for Backjump. However, this way we obtain one constrained literal with an explanation in a context Γ.

In induction step we consider the case when a context Γ contains n constrained literals with explanations and remaining literals in Γ are decision literals. Let $C_1\Theta$ be a conflicting clause and let Θ_r be a set of instantiations that falsify C_1. If $C_1\Theta$ is falsified only by decision literals from Γ, then we apply the same reasoning as above and conclude that Backjump is enabled. Let us now assume that a literal $\overline{\ell}^{C_l\Theta'}\Theta_l$ is used for falsifying C_1. Then, $C_1\Theta = (C'(\boldsymbol{z}) \vee \ell(\boldsymbol{x})) \cdot \Theta$ and $\Theta_r \bowtie \Theta_l \neq \emptyset$. Our goal is to show that the preconditions of the Resolve rules are enabled. Using Invariant 1 for a literal $\overline{\ell}^{C_l\Theta'}\Theta_l$, we obtain that $C_l\Theta' = (\ell_1 \vee \ldots \vee \ell_k \vee \overline{\ell}(\boldsymbol{x}))$ and there are assignments $\overline{\ell}_i\Theta_i \in \Gamma$ such that $\emptyset \neq \Theta_l \subseteq \pi_{\boldsymbol{x}}(\Theta' \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_k)$. Let us introduce the shorthand $C(\boldsymbol{y}) = \ell_1 \vee \ldots \vee \ell_k$.

One also needs to check whether there exists a literal $\ell(\boldsymbol{y}) \in C'$ such that $\delta_{\boldsymbol{y}\to\boldsymbol{x}}\pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell \neq \emptyset$. If such a literal exists, preconditions of the Factoring rule are enabled and using the Factoring rule we eliminate that literal from C'. We repeat the procedure until there are no more such literals, which ensures that $\delta_{\boldsymbol{y}\to\boldsymbol{x}}\pi_{\boldsymbol{y}}\Theta_r \bowtie \Theta_\ell = \emptyset$, for every $\ell(\boldsymbol{y}) \in C'$. During those transformations, only clause C' was changing; the context Γ stayed unchanged and we remained in the conflict mode.

After eliminating all redundant literals from C', it remains to check whether $\hat{\pi}_{\boldsymbol{x}}(\Theta_r \bowtie \Theta_\ell \bowtie \mathit{premises}(\overline{\ell}\Theta_\ell)) \neq \emptyset$. This holds as well, because $\Theta_r \bowtie \Theta_l \neq \emptyset$ and $\Theta_l \subseteq \pi_{\boldsymbol{x}}\mathit{premises}(\overline{\ell}\Theta_\ell)$ and $\boldsymbol{x}$ is the only common variable that occurs in Θ_r, Θ_ℓ and $\mathit{premises}(\overline{\ell}\Theta_\ell)$. Let $\Theta'_r = \hat{\pi}_{\boldsymbol{x}}(\Theta_r \bowtie \Theta_\ell \bowtie \mathit{premises}(\overline{\ell}\Theta_\ell))$. Note that if $C(\boldsymbol{y}) = C'(\boldsymbol{z}) = \Box$ then $\Theta'_r = \{[]\}$.

To summarize, if some constrained literal with annotated explanation falsifies the clause C_1, then in a finite number of steps all preconditions of the Resolve rule can become enabled. After applying it, we remain in the conflict mode.

However, the context Γ has changed and now it contains $n-1$ constrained literals annotated with explanations. On this new context, we apply induction hypothesis.

This concludes the proof for stuck-freeness of DPLL($\mathcal{SX}$).

The DPLL($\mathcal{SX}$) calculus can directly simulate propositional grounding and this way we obtain completeness:

Theorem 3 (Completeness). *DPLL($\mathcal{SX}$) is complete for EPR.*

The calculus admits the expected asymptotic complexity of EPR. Suppose the maximal arity of any relation is a, the number of constants is $n = |\Sigma|$, and the number of relations is m, set $K \leftarrow m \times (n^a)$, then:

Theorem 4 (Complexity). *The rules of DPLL($\mathcal{SX}$) terminate with at most $O(K \cdot 2^K)$ applications, and with maximal space usage $O(K^2)$.*

Proof. First note that a context can enumerate each literal assignment explicitly using at most $O(K)$ space, since each of the m literals should be evaluated at up to n^a instances. Literals that are tagged by explanations require up to additional $O(K)$ space, each.

For the number of rule applications, consider the ordering $\prec$ on contexts defined as the transitive closure of:

$$\Gamma, \ell'\Theta', \Gamma' \prec \Gamma, \diamond, \ell\Theta, \Gamma'' \tag{1}$$

$$\Gamma, \diamond, \ell\Theta', \Gamma' \prec \Gamma, \diamond, \ell\Theta, \Gamma'' \quad \text{when } \Theta' \subset \Theta \tag{2}$$

The two rules for $\prec$ correspond to the inference rules Backjump and Refine that generate contexts of decreased measure with respect to $\prec$. Furthermore, we may restrict our attention to contexts Γ where for every literal ℓ, such that $\Gamma = \Gamma', \ell\Theta, \Gamma''$ if $\exists \Theta' \,.\, \ell\Theta' \in \Gamma', \Gamma''$, then $\Theta' \bowtie \Theta = \emptyset$. There are at most $K!$ such contexts, but we claim the longest $\prec$ chain is at most $K \cdot 2^K$. First, if all sets are singletons, then the derivation is isomorphic to a system with K atoms, which requires at most 2^K applications of the rule (1). Derivations that use non-singleton sets embed directly into a derivation with singletons using more steps. Finally, rule (2) may be applied at most K times between each step that corresponds to a step of the first kind.

Note that the number of rule applications does not include the cost of manipulating substitution sets. This cost depends on the set representations. While the asymptotic time complexity is (of course) no better than what a brute force grounding provides, DPLL($\mathcal{SX}$) only really requires space for representing the logical context Γ. While the size of Γ may be in the order K, there is no requirement for increasing F from its original size. As we will see, the use of substitution sets may furthermore compress the size of Γ well below the bound of K. One may worry that in an implementation, the overhead of using substitution sets may be prohibitive compared to an approach based on substitutions alone. Section 3.3 describes a data-structure that compresses substitution sets when they can be represented as substitutions directly.

3 Refinements of DPLL($\mathcal{SX}$)

The calculus presented in Section 2 is a general framework for using substitution sets in the context of DPLL. We first discuss a refinement of the calculus that allows to apply unit propagation for several assignments simultaneously. Second, we examine data-structures and algorithms for representing, indexing and manipulating substitution sets efficiently during search.

3.1 Simultaneous propagation and FUIP-based conflict resolution

In general, literals are assigned substitutions at different levels in the search. Unit propagation and conflict detection can therefore potentially be identified based on several different instances of the same literals. For example, given the clause $(\overline{p}(x) \vee q(x))$ and the context $p(a), p(b)$, unit propagation may be applied on $p(a)$ to imply $q(a)$, but also on $p(b)$ to imply $q(b)$. We can factor such operations into simultaneous versions of the UnitPropagate and Conflict rules. The simultaneous version of the propagation and conflict rules take the form shown in Fig 4.

$$\textsf{S-UnitPropagate}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell(\boldsymbol{x})), \\ \mathit{\Theta}'_i = \cup\{\mathit{\Theta}_i \mid \overline{\ell}_i \mathit{\Theta}_i \in \mathit{\Gamma}\},\ \mathit{\Theta}'_\ell = \bigcup\{\mathit{\Theta}_\ell \mid \ell\mathit{\Theta}_\ell \in \mathit{\Gamma}\}, \\ \mathit{\Theta}' = \pi_{\boldsymbol{x}}(\mathit{\Theta} \bowtie \mathit{\Theta}'_1 \bowtie \ldots \bowtie \mathit{\Theta}'_k) \setminus \mathit{\Theta}'_\ell \neq \emptyset \\ \mathit{\Theta}' \bowtie \bigcup\{\mathit{\Theta}_{\overline{\ell}} \mid \overline{\ell}\mathit{\Theta}_{\overline{\ell}} \in \mathit{\Gamma}\} = \emptyset \end{array}}{\mathit{\Gamma} \parallel F, C \cdot \mathit{\Theta} \implies \mathit{\Gamma}, \ell^{C \cdot \mathit{\Theta}} \cdot \mathit{\Theta}' \parallel F, C \cdot \mathit{\Theta}}$$

$$\textsf{S-Conflict}\ \frac{\begin{array}{l} C = (\ell_1 \vee \ldots \vee \ell_k),\ \mathit{\Theta}'_i = \cup\{\mathit{\Theta}_i \mid \overline{\ell}_i \mathit{\Theta}_i \in \mathit{\Gamma}\}, \\ \mathit{\Theta}' = \mathit{\Theta} \bowtie \mathit{\Theta}'_1 \bowtie \ldots \bowtie \mathit{\Theta}'_k \neq \emptyset \end{array}}{\mathit{\Gamma} \parallel F, C \cdot \mathit{\Theta} \implies \mathit{\Gamma} \parallel F, C \cdot \mathit{\Theta} \parallel C \cdot \mathit{\Theta}, \mathit{\Theta}'}$$

Fig. 4. S-UnitPropagate and S-Conflict

Correctness of these simultaneous versions rely on the basic the property that $\bowtie$ distributes over unions:

$$(R \cup R') \bowtie Q \ = \ (R \bowtie Q) \cup (R' \bowtie Q) \ \text{ for every } R, R', Q \qquad (3)$$

Thus, every instance of S-UnitPropagate corresponds to a set of instances of UnitPropagate, and for every S-Conflict there is a selection of literals in $\mathit{\Gamma}$ that produces a Conflict. The rules suggest to maintain accumulated sets of substitutions per literal, and apply propagation and conflict detection rules once per literal, as opposed to once per literal occurrence in $\mathit{\Gamma}$. A trade-off is that we break invariant 1 when using these rules. Instead we have:

Invariant 2. *For every derived context of the form $\mathit{\Gamma}, \ell^{C \cdot \mathit{\Theta}}\mathit{\Theta}', \mathit{\Gamma}'$ where $C = (\ell_1 \vee \ldots \vee \ell_k \vee \ell)$, it is the case that $\emptyset \neq \mathit{\Theta}' \subseteq \pi_{\boldsymbol{x}}(\mathit{\Theta} \bowtie \mathit{\Theta}'_1 \bowtie \ldots \bowtie \mathit{\Theta}'_k)$ where $\mathit{\Theta}'_i = \bigcup\{\mathit{\Theta}_i \mid \overline{\ell}_i \mathit{\Theta}_i \in \mathit{\Gamma}\}$.*

Invariant (2) still suffices to establish the theorems from Section 2.3, even though it is weaker, as the set $(\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k)$ contains the joins of substitutions from individual literals in Γ. The function *premises* is still admissible, thanks to (3).

Succinctness Note the asymmetry between the use of simultaneous unit propagation and the conflict resolution strategy: while the simultaneous rules allow to use literal assignments from several levels at once, conflict resolution traces back the origins of the propagations that closed the branches. The net effect may be a conflict clause that is exponentially larger than the depth of the branch. As an illustration of this situation consider the clauses where p is an n-ary predicate:

$$\begin{array}{l} \neg p(0,\ldots,0) \;\wedge\; \mathit{shape}(\star) \;\wedge\; \mathit{shape}(\triangle) \\ \wedge_i\, [p(\boldsymbol{x},\star,0,..,0) \wedge p(\boldsymbol{x},\triangle,0,..,0) \;\rightarrow\; p(\boldsymbol{x},0,0..,0)] \text{ where } \boldsymbol{x} = x_0,\ldots,x_{i-1} \\ \wedge_{0\le j<n}\, \mathit{shape}(x_j) \;\rightarrow\; p(x_0,\ldots,x_{n-1}) \end{array} \tag{4}$$

Claim. The clauses are contradictory, and any resolution proof requires 2^n steps.

Justification. Backchaining from $p(0,\ldots,0)$, we observe that all possible derivations are of the form:

$$\begin{array}{rl} p(0,\ldots,0) & \leftarrow p(\star,0,\ldots,0), p(\triangle,0,\ldots,0) \\ & \leftarrow p(\star,\triangle,0,..,0), p(\star,\star,0,..,0), p(\triangle,\triangle,0,..,0), p(\triangle,\star,0,..,0) \\ & \leftarrow \ldots \\ & \leftarrow p(\star,\star,\ldots),\ldots,p(\triangle,\triangle,\ldots) \text{ all } 2^n \text{ combinations} \\ & \leftarrow \mathit{shape}(\star)\ldots\mathit{shape}(\triangle) \end{array}$$

Claim. DPLL($\mathcal{SX}$) with simultaneous unit propagation requires $O(n)$ steps to complete the derivation.

Justification. The two assertions $\mathit{shape}(\star)$ and $\mathit{shape}(\triangle)$ may be combined into $\mathit{shape}(x)\{\star,\triangle\}$ and then used to infer $p(\boldsymbol{x})\{\star,\triangle\} \times \ldots \times \{\star,\triangle\}$ in one propagation. Each consecutive propagation may be used to produce $p(\boldsymbol{x})\Theta$, where Θ contains a suffix with k consecutive 0's and the rest being all combinations of $\star$ and $\triangle$.

In this example, we did in fact not need to perform conflict resolution at all because the problem was purely Horn, and no decisions were required to derive the empty clause. But it is simple to modify such instances to non-Horn problems, and the general question remains how and whether to avoid an exponential cost of conflict resolution as measured by the number of propagation steps used to derive the conflict.

One crude approach for handling this situation is to abandon conflict resolution if the size of the conflict clause exceeds a threshold. When abandoning conflict resolution apply Refine, which is enabled as long as there is at least one decision literal on the stack whose substitution set is a non-singleton. If all decision literals use singletons, then Refine does not apply. In this case we have to use a different way of backtracking. We can flip the last decision literal under the context of the negation of all decision literals on the stack. The rule corresponding to this approach is U(nit)-Refine:

$$\textsf{U-Refine}\ \frac{\begin{array}{c}\boldsymbol{c} \in \Sigma^k \quad \diamond \notin \Gamma', \quad \ell_1\Theta_1, \ldots, \ell_m\Theta_m \text{ are the decision literals in } \Gamma \\ C' = \overline{\ell} \vee \overline{\ell}_1 \vee \ldots \vee \overline{\ell}_m, \Theta' = \{\boldsymbol{c}\} \bowtie \Theta_1 \bowtie \ldots \bowtie \Theta_m \text{ is a singleton}\end{array}}{\Gamma, \diamond, \ell\{\boldsymbol{c}\}, \Gamma' \parallel F \parallel C \cdot \Theta, \Theta_r \implies \Gamma, \overline{\ell}\{\boldsymbol{c}\}^{C' \cdot \Theta'} \parallel F}$$

But with succinct substitution sets it is sometimes possible to match the succinctness of unit propagation during conflict resolution. The approach we are going to present will use ground clauses during resolution. The ground clauses can have multiple occurrences of the same predicate symbol, but applied to different (ground) arguments. We then summarize the different arguments using a substitution set, so that the representation of the ground clause only requires each predicate symbol at most twice (positive and/or negated), but with a potentially succinct representation of the arguments.

Suppose S-Conflict infers the conflict clause $C \cdot \Theta$ and set Θ_r. Let θ_0 be an arbitrary instantiation in Θ_r. Initialize the map Ψ from the set of signed predicate symbols to substitution sets as follows:

$$\Psi(\ell) \leftarrow \bigcup\{\pi_{\boldsymbol{x}_i}\theta_0 \mid \ell(\boldsymbol{x}_i) \in C\}, \quad \text{for } \ell \in \mathcal{L}. \tag{5}$$

Note that a clause C may have multiple occurrences of a predicate symbol with the same sign, but applied to different arguments. The definition ensures that if $\ell \in \mathcal{L}$ is a signed predicate symbol that does not occur in C, then $\Psi(\ell) = \emptyset$.

Example 5. Assume $C = p(x_1) \vee p(x_2) \vee q(x_3)$ and $\theta = (a, b, c)$, then $\Psi(p) = \{a, b\}, \Psi(\overline{p}) = \emptyset, \Psi(q) = \{c\}, \Psi(\overline{q}) = \emptyset$.

We can directly reconstruct a clause from Ψ by creating a disjunction of $\Sigma_{\ell \in \mathcal{L}}|\Psi(\ell)|$ literals and a substitution that is the product of all elements in the range of Ψ. This inverse mapping is called *clause_of*(Ψ). Sets in the range of Ψ may get large, but we can here rely on the same representation as used for substitution sets. We can now define a (first-unique implication point) resolution strategy that works using Ψ. We formulate the strategy separately from the already introduced rules, as we need to ensure that we can maintain the representation of the conflict clause using Ψ.

```
$resolve(\Gamma_1, \diamond, \Gamma_2, \Psi)$ = Backjump with $\Gamma_1 \ell^{C \cdot \Theta}\{\boldsymbol{c}\}$ if
        $\ell \in \mathbf{Dom}(\Psi),\ \boldsymbol{c} \in \Psi(\ell),\quad \Psi(\ell) \setminus \{\boldsymbol{c}\} \subseteq \bigcup\{\Theta' \mid \overline{\ell}\Theta' \in \Gamma_1\}$
        $\Psi(\ell') \subseteq \bigcup\{\Theta' \mid \overline{\ell'}\Theta' \in \Gamma_1\}$ for $\ell \neq \ell'$
        $C \cdot \Theta = clause\_of(\Psi)$
$resolve(\Gamma, \ell\Theta, \Psi) = resolve(\Gamma, \Psi)$ if $\Theta \cap \Psi(\ell) = \emptyset$
$resolve(\Gamma, \ell^{C \vee \ell \cdot \Theta}\Theta', \Psi) = resolve(\Gamma, \Psi)$, if $\Theta' \cap \Psi(\ell) = \{\boldsymbol{c}\}$, and where
    $\Psi(\ell) \leftarrow \Psi(\ell) \setminus \Theta'$
    for $\ell'(\boldsymbol{x}) \in C$: $\Psi(\ell') \leftarrow \Psi(\ell') \cup \pi_{\boldsymbol{x}}(premises(\ell^{C \vee \ell \cdot \Theta}\Theta'))$
$resolve(\Gamma, \diamond, \ell\Theta, \Psi)$ = Refine if other rules don't apply.
```

Besides the cost of performing the set operations, the strategy still suffers from the potential of generating an exponentially large implied learned clause $C \cdot \Theta'$ during backjumping. An implementation can choose to resort to applying Refine or U-Refine in these cases. We do not have experimental experience with the representation in terms of Ψ. Instead our prototype implementation uses Refine together with U-Refine, when the conflict resolution steps start generating conflict clauses with more than 20 (an arbitrary default) literals.

3.2 Selecting decision literals and substitution sets

Selecting literals and substitution sets blindly for Decide is possible, but not a practical heuristic. As in the Model-evolution calculus [3], we take advantage of the current assignment Γ to guide selection. Closure of substitution sets under complementation streamlines the task a bit for the case of DPLL($\mathcal{SX}$). First observe that Γ induces a default interpretation of the instances of every atom p by taking:

$$[\![p]\!] = \bigcup \{\Theta' \mid p\Theta' \in \Gamma\} \quad \text{and} \quad [\![\overline{p}]\!] = \overline{[\![p]\!]} \tag{6}$$

Note that we can assume that Γ is consistent, so $\bigcup\{\Theta' \mid \overline{p}\Theta' \in \Gamma\} \subseteq [\![\overline{p}]\!]$. Using the current assignment for the positive literals and the complement thereof for negative ones is an arbitrary choice in the context of DPLL($\mathcal{SX}$). One may fix a default interpretation differently for each atom. But note that this particular choice coincides with negation as failure for the case of Horn clauses.

We now say that ℓ_i is a candidate decision literal with instantiation Θ'_i if there is a clause $C \cdot \Theta$, such that $C = (\ell_1 \vee \ldots \vee \ell_k)$, $1 \leq i \leq k$, and:

$$\Theta'_i = (\Theta \bowtie [\![\overline{\ell}_1]\!] \bowtie \ldots \bowtie [\![\overline{\ell}_k]\!]) \setminus \bigcup\{\Theta' \mid \overline{\ell}_i\Theta' \in \Gamma\} \neq \emptyset \tag{7}$$

Our prototype uses a greedy approach for selecting decision literals and substitution sets: predicates with lower arity are preferred over predicates with higher arities. In particular, propositional atoms are used first and they are assigned using standard SAT heuristics. Predicates with non-zero arity that are not completely assigned are checked for condition (7) and we pick the first applicable candidate. The process either produces a decision literal, or determines that the current set of clauses are satisfiable in the default interpretation, as the following easy lemma summarizes:

Lemma 2. *If for a state $\Gamma \parallel F, (\ell_1 \vee \ldots \vee \ell_k) \cdot \Theta$ it is the case that neither* Unit-Propagate *or* Conflict *are enabled and*

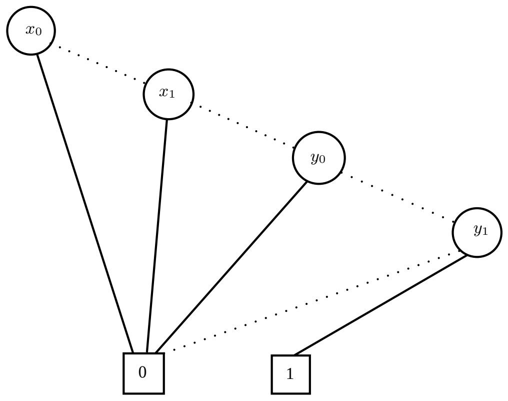
$$\Theta \bowtie [\![\overline{\ell}_1]\!] \bowtie \ldots \bowtie [\![\overline{\ell}_k]\!] \neq \emptyset \tag{8}$$

then there is some i, such that $1 \leq i \leq k$, that satisfies (7). Furthermore, the identified substitution Θ'_i is disjoint from any Θ', where $\ell_i\Theta' \in \Gamma$ or $\overline{\ell}_i\Theta' \in \Gamma$. Conversely, if the current state is closed under propagation and conflict and there is no clause that satisfies (8), then the default interpretation is a model for the set of clauses F.

Proof. If the current state is closed under Conflict and UnitPropagate, then for every clause $(\ell_1 \vee \ldots \vee \ell_k) \cdot \Theta$: $\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k = \emptyset$ for $\Theta'_i = \bigcup\{\Theta' \mid \overline{\ell}_i\Theta' \in \Gamma\}$. Suppose that (8) holds, then by $\bigcup\{\Theta' \mid \overline{p}\Theta' \in \Gamma\} \subseteq [\![\overline{p}]\!]$ and distributivity of $\bowtie$ over $\cup$ there is some i where (7) holds. The converse direction is immediate.

3.3 Hybrid substitution sets

Substitution sets can be represented as BDDs. Here we briefly outline how to do it. Let Σ be a signature. We encode each constant using $\log(|\Sigma|)$ bits. For example, let $\Sigma = \{a, b, c, d\}$. Then we can encode those constants as follows: $a = 00,\ b = 01,\ c = 10,\ d = 11$. To represent a variable we also use $\log(|\Sigma|)$ bits. To continue with the previous example, let $\mathcal{V} = \{x, y\}$ and let $x = x_0x_1,\ y = y_0y_1$. For a given substitution set Θ we construct a BDD corresponding to Θ. We do it as follows: nodes in BDD are labeled with variable bits. The edge labeled i emanating from the node labeled x expresses that the bit x has value i. Each path in the BDD encodes an assignment of variables. As an example, the path $x_0 0 x_1 1 y_0 1 y_1 0$ defines the assignment $x \mapsto b, y \mapsto c$. We use this encoding for the representation of substitution sets. Every path ends in 0-sink or 1-sink. If the path ends with the 1-sink, then the assignment that the path encodes belongs to the substitution set. Figure 5 illustrates the described technique on two examples.



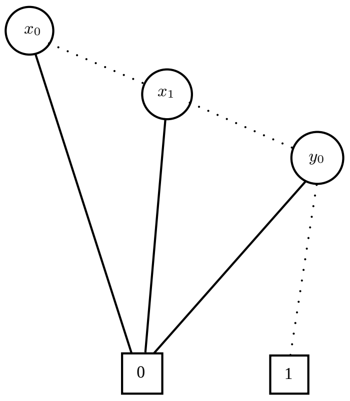


Fig. 5. The left BDD represents the substitution set $\Theta_1 = \{[x \mapsto a, y \mapsto b]\}$, while the right one represents $\Theta_2 = \{[x \mapsto a, y \mapsto a], [x \mapsto a, y \mapsto b]\}$. The dotted-line represents an edge labeled with 0 and the continuous line corresponds to an edge labeled with 1.

Representing all substitution sets directly as BDDs is not practical. In particular, computing $\Theta \bowtie \Theta'_1 \bowtie \ldots \bowtie \Theta'_k$ by directly applying the definitions of $\bowtie$ as conjunction and $\delta_{\rightarrow}$ as BDD renaming does not work in practice for clauses with several literals: simply building a BDD for Θ (where Θ is the substitution set associated with a clause) can be prohibitively expensive. We here investigate a representation of substitution sets called *hybrid substitution sets* that admit pre-compiling and factoring several of the operations used during constraint propagation. The format is furthermore amenable to a two-literal watch strategy for the propositional case.

Definition 1 (Hybrid substitution sets). *A hybrid substitution set is a pair (θ, Θ), where θ is a substitution, and Θ is a relation (substitution set). Furthermore, the domain of Θ consists of the variables where θ is idempotent. That is, $\mathbf{Dom}(\Theta) = \{x \in \mathbf{Dom}(\theta) \mid \theta(x) = x\}$. The substitution set associated with a hybrid substitution is the set of instances: $\Theta \bowtie \{\theta\}$.*

In one extreme, a standard substitution θ is equivalent to the hybrid substitution set $(\theta, \top)$. We are using the terminology *standard* substitution for a mapping from variables to constants or variables. Our prototype compiles clauses into such substitution sets. In the other extreme, every substitution set Θ can be represented as (id, Θ), where id is the identity substitution over the domain of Θ. Our prototype uses such substitution sets to constrain literals. We will henceforth take the liberty to abuse notation and treat substitutions θ and substitution sets Θ also as hybrid substitution sets.

Hybrid substitution sets are attractive because common operations are cheap (linear time) when the substitutions are proper. They also enjoy closure properties under the main relational algebraic operations that are used in conflict resolution.

Lemma 3. *Standard substitutions are closed under the operations: $\bowtie$, $\pi_{\boldsymbol{x}}$, $\hat{\pi}_{\boldsymbol{x}}$, $\sigma_{\boldsymbol{x}=\boldsymbol{y}}$, and $\delta_{\boldsymbol{x}\to\boldsymbol{y}}$, but not under union nor complementation.*

For example, given the alphabet $\Sigma = \{a, b, c\}$, the standard substitution $[x \mapsto a]$ has the complement $\{[x \mapsto b], [x \mapsto c]\}$, but this is a substitution set, and not a standard substitution.

Even if the hybrid substitution sets are not proper, the complexity of the common operations is reduced by using the substitution component when it is not the identity. For example, representing each variable in a BDD requires $\log(|\Sigma|)$ bits, and if the bits of two variables are spaced apart by k other bits, the operation that restricts a BDD equating the two variables may cause a size increase of up to 2^k. The problem can be partially addressed using static or dynamic variable reordering techniques, but variable orderings have to be managed carefully when variables are shared among several substitution sets.

Constraint propagation Consider a clause $C \cdot (\theta, \Theta) \in F$ and a substitution Θ'_i (associated with literal $\overline{\ell}_i(\boldsymbol{x})$ in Γ, where ℓ_i occurs C). The main operation during constraint propagation is computing $(\theta, \Theta) \bowtie \delta_{\boldsymbol{x}\to\boldsymbol{x}_i}\Theta'_i$, which is equivalent to

$$(\theta, \Theta \bowtie (\Theta'_i \wr r_i)) \quad \text{where} \quad r_i = [x \mapsto \theta(\delta_{\boldsymbol{x}\to\boldsymbol{x}_i} x) \mid x \in \boldsymbol{x}]. \qquad (9)$$

The equivalence suggests to pre-compute and store the substitution r_i, for every clause C and literal in C. Each renaming may be associated with several clauses; and we can generalize the two-literal watch heuristic for a clause C by using watch literals ℓ_i and ℓ_j from C as guards if the current assignments Θ'_i and Θ'_j satisfy $\Theta'_i \wr r_i = \Theta'_j \wr r_j = \emptyset$.

Resolution In general, when taking the join of two hybrid substitution sets we have the equivalence: $(\theta, \Theta) \bowtie (\theta', \Theta') = (m, m(\Theta) \bowtie m(\Theta'))$, where $m = mgu(\theta, \theta')$, if the most general unifier exists, otherwise the join is $(id, \bot)$. Resolution requires computing $\hat{\pi}_{\boldsymbol{x}}((\theta, \Theta) \bowtie (\theta', \Theta'))$ or in general $\hat{\pi}_{\boldsymbol{x}}(\delta_{y \to z}(\theta, \Theta) \bowtie \delta_{u \to v}(\theta', \Theta'))$ where $\boldsymbol{x}, y, z, u, v$ are suitable vectors of variables. Again, we can compose the re-namings first with the substitutions, compute the most general unifier $m = mgu(\delta_{y \to z}\theta, \delta_{u \to v}\theta')$, in such a way that if $m(y) = m(x)$, for $x \in \boldsymbol{x}, y \notin \boldsymbol{x}$, then $m(y)$ is a constant or maps to some variable also not in $\boldsymbol{x}$; and returning $(m \setminus \boldsymbol{x}, \exists \boldsymbol{x}(m(\Theta) \wedge m(\Theta')))$. It is common for BDD packages to supply a single operation for $\exists \boldsymbol{x}(\varphi \wedge \psi)$.

4 Implementation and evaluation

We implemented DPLL($\mathcal{SX}$) as a modification of the propositional SAT solver used in the SMT solver Z3. The implementation associates with each clause a hybrid substitution set and pre-compiles the set of substitutions r_i used in (9). This allows the BDD package, we use BuDDy[1], to cache results from repeated substitutions of the same BDDs (the corresponding operation is called `vec_compose` in BuDDy). BDD caching was more generally useful in obliterating special purpose memoization in the SAT solver. For instance, we attempted to memoize the default interpretations of clauses as they could potentially be re-used after back-tracking, but we found so far no benefits of this added memoization over relying on the BDD cache. BuDDy supports finite domains directly making it easier to map a problem with a set of constants $\Sigma = c_1, \ldots, c_k$ into a finite domain of size $2^{\lceil \log(k) \rceil}$. Rounding the domain size up to the nearest power of 2 does not change satisfiability of the problem, but has a significant impact on the performance of BDD operations. Unfortunately, we have not been able to get dynamic variable re-ordering to work with finite domains in BuDDy, so all our results are based on a fixed default variable order.

As expected, our prototype scales reasonably well on formula (4). It requires n propagations to solve an instance where p has arity n. With $n = 10$ takes $0.01s.$, $n = 20$ takes $0.2s.$, and $n = 200$ takes $18s.$ (and caches 1.5M BDD nodes, on a 32bit, 2GHz, 2GB, TS2500). Darwin [2] handles $n = 10$ in 0.4 seconds and 2049 propagations, while increasing n to 20 is already too overwhelming.

Example 6. Suppose p is an n-ary predicate, and that we have n unary predicates $a_0, \ldots, a_{n-1}$, then consider the (non-Horn) formula:

$$\begin{aligned} & \wedge_{0 \leq i < n} \forall \boldsymbol{x} \ . \ [p(\boldsymbol{x}) \rightarrow p(.., x_{i-1}, 1, x_{i+1}, ..)] \\ & \wedge \ p(0, \ldots, 0) \ \wedge \ \wedge_{0 \leq i < n}(a_i(0) \vee a_i(1)) \ \wedge \ \forall \boldsymbol{x} \ . \ [(\wedge_i a_i(x_i)) \rightarrow \neg p(\boldsymbol{x})] \end{aligned} \tag{10}$$

DPLL($\mathcal{SX}$) uses n simultaneous propagations to learn the assignment $p(\boldsymbol{x})\top$. In contrast, standard unit propagation requires 2^n steps. Since no splitting was required to learn this assignment, it can be used to eliminate p from conflict

[1] http://buddy.wiki.sourceforge.net

clauses during lemma learning. The resulting conflict clauses during backjumping are then $\forall \boldsymbol{x} \, . \, \bigvee_{0 \leq i < m} \neg a_i(x_i)$ for $m = n - 1, \ldots 1$. Accordingly, the prototype uses 0.06 seconds for $n = 30$, 0.9s for $n = 80$, and 26s. for $n = 200$, while even a very good instantiation based prover Darwin requires $O(2^{n-1})$ branches, which is reflected in the timings: for $n = 11, 12, 13, 14, 15, 16$, take $1, 4, 16, 60, 180, 477$ seconds respectively.

We also ran our prototype on the CASC-21 benchmarks from the EPS and EPT divisions. In the EPT division fails to prove `PUZ037-3.p`, with a timeout of 120 seconds, as the BDDs built during propagation blow upIt solves the other 49 problems, using less than 1 second for all but `SYN439-1.p`, which requires 894 conflicts and 9.8 seconds. In the EPS division our prototype solves 46 out of 50 problems within the given 120s. timeout.

5 Conclusions

Related work DPLL($\mathcal{SX}$) is a so called *instance*-based method [1] and it shares several features with instance-based implementations derived from DPLL, such as the Model Evolution Calculus ($\mathcal{ME}$) calculus [3], the iProver [9], and the earlier work on a primal-dual approach for satisfiability of EPR [8]. These methods are also decision procedures for EPR that go well beyond direct propositional grounding (as do resolution methods [7]). Lemma learning in $\mathcal{ME}$ [2] comprises of two rules GRegress and a non-ground lifting Regress. In a somewhat rough analogy to Regress, the resolution rules used in DPLL($\mathcal{SX}$) uses the set Θ_r to guide a more general lifting for the produced conflict clause. Connections between relational technology and theorem proving were made in [17], as well as [16]. The use of BDDs for compactly representing relations is wide-spread. Of high relevance to DPLL($\mathcal{SX}$) is the system BDDBDDB, which is a Datalog engine based on BDDs [18]. Semantics of negation in Datalog aside, DPLL($\mathcal{SX}$) essentially reduces to BDDBDDB for Horn problems. Simultaneous unit propagation is for instance implicit in the way clauses get compiled to predicate transformers, but on the other hand, apparatus for handling non-Horn problems is obviously absent from BDDBDDB.

Extensions A number of compelling extensions to DPLL($\mathcal{SX}$) remain to be investigated. For example, we may merge two clauses $C \cdot \Theta$ and $C \cdot \Theta'$ by taking the union of the substitution sets. The clause $C \cdot \Theta'$ could for instance be obtained by resolving binary clauses, so this feature could simulate iterative squaring known from symbolic model checking. Examples where iterative squaring pays of are provided in [14]. We currently handle equality in our prototype by supplying explicit equality axioms (reflexivity, symmetry, transitivity, and congruence) for the binary equality relation $\simeq$, but supporting equality as an intrinsic theory is possible and the benefits would be interesting to study. We have also to work out efficient ways of building in subsumption. Supporting other theories is also possible by propagating all instances from substitution sets, but it would be appealing to identify cases where an explicit enumeration of substitution sets can be avoided. We used reduced ordered BDDs in our evaluation of the calculus, but this is by no means the only possible representation. We may for instance delay forming canonical decision diagrams until it is required for evaluating (non-emptiness) queries (a technique used for Boolean Expression Diagrams). It would also be illustrative to investigate how DPLL($\mathcal{SX}$) applies to finite model finding and general first-order problems. Darwin(FM) already addressed using EPR for finite model finding, and as GEO [5] exemplifies, one can extend finite model finders to the general first-order setting.

Another avenue to pursue is relating our procedure with methods used for QBF. While there is a more or less direct embedding of QBF into EPR (obtained by Skolemization) the decision problem for QBF is *only* PSPACE complete, while the procedure we outlined requires up to exponential space.

Thanks to the referees for their very detailed and constructive feedback on the submitted version of this paper.

References

1. Peter Baumgartner. Logical engineering with instance-based methods. In Pfenning [15], pages 404–409.
2. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. In Miki Hermann and Andrei Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 572–586. Springer, 2006.
3. Peter Baumgartner and Cesare Tinelli. The model evolution calculus as a first-order DPLL method. *Artif. Intell.*, 172(4-5):591–632, 2008.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006.
6. Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded Model Checking with QBF. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 408–414. Springer, 2005.
7. Christian G. Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1791–1849. Elsevier and MIT Press, 2001.
8. G. Gallo and G. Rago. The satisfiability problem for the Schönfinkel-Bernays fragment: partial instantiation and hypergraph algorithms. Technical Report 4/94, Dip. Informatica, Universit'a di Pisa, 1994.
9. Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *LICS*, pages 55–64. IEEE Computer Society, 2003.
10. Sava Krstic and Amit Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In Boris Konev and Frank Wolter, editors, *FroCos*, volume 4720 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2007.
11. Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.

12. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
13. Juan Antonio Navarro Pérez and Andrei Voronkov. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. In Pfenning [15], pages 346–361.
14. Juan Antonio Navarro Pérez and Andrei Voronkov. Proof systems for effectively propositional logic. In Allesandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, 2008.
15. Frank Pfenning, editor. *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*. Springer, 2007.
16. Tanel Tammet and Vello Kadarpik. Combining an inference engine with database: A rule server. In Michael Schroeder and Gerd Wagner, editors, *RuleML*, volume 2876 of *Lecture Notes in Computer Science*, pages 136–149. Springer, 2003.
17. Andrei Voronkov. Merging relational database technology with constraint technology. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Ershov Memorial Conference*, volume 1181 of *Lecture Notes in Computer Science*, pages 409–419. Springer, 1996.
18. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.