The Model Evolution Calculus

Peter Baumgartner[1] and Cesare Tinelli[2]

[1] Institut für Informatik, Universität Koblenz-Landau, peter@uni-koblenz.de
[2] Department of Computer Science, The University of Iowa, tinelli@cs.uiowa.edu

Abstract. The DPLL procedure is the basis of some of the most successful propositional satisfiability solvers to date. Although originally devised as a proof-procedure for first-order logic, it has been used almost exclusively for propositional logic so far because of its highly inefficient treatment of quantifiers, based on instantiation into ground formulas. The recent FDPLL calculus by Baumgartner was the first successful attempt to lift the procedure to the first-order level without resorting to ground instantiations. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores other aspects of the procedure that, although not necessary for completeness, are crucial for its effectiveness in practice. In this paper, we present a new calculus loosely based on FDPLL that lifts these aspects as well. In addition to being a more faithful lifting of the DPLL procedure, the new calculus contains a more systematic treatment of *universal literals*, one of FDPLL's optimizations, and so has the potential of leading to much faster implementations.

1 Introduction

In propositional satisfiability the DPLL procedure, named after its authors: Davis, Putnam, Logemann, and Loveland [8, 7], is the dominant method for building (complete) SAT solvers. Its popularity is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and of clauses.

Interestingly, the DPLL procedure was actually devised in origin as a proof-procedure for first-order logic. Its treatment of quantifiers is highly inefficient, however, because it is based on enumerating all possible ground instances of an input formula's clause form, and checking the propositional satisfiability of each of these ground instances one at a time. Given the great success of DPLL-based SAT solvers today, two natural research questions arise. One is whether the DPLL procedure can be properly lifted to the first-order level—in the sense first-order resolution lifts propositional resolution, say. The other is whether those powerful search heuristics that make DPLL so effective at the propositional level can be successfully adapted to the first-order case. We answer the first of these two questions affirmatively in this paper, providing a complete lifting of the DPLL procedure to first-order clausal logic by means of a new sequent-style calculus, the *Model Evolution* calculus, or $\mathcal{ME}$ for short. We believe that the $\mathcal{ME}$ calculus can be used to answer the second question affirmatively as well, although that will be the subject of our future work.

The recent FDPLL calculus by Baumgartner [3] was the first successful attempt to lift the DPLL procedure to the first-order level without resorting to ground instantiations. FDPLL lifts the core of the DPLL procedure, the splitting rule, but ignores another major aspect, *unit propagation* [18], that although not necessary for its completeness is absolutely crucial to its effectiveness in practice. The calculus described in this paper lifts this aspect as well. While the $\mathcal{ME}$ calculus borrows many fundamental ideas from FDPLL and generalizes it, it is not an extension of FDPLL proper but of DPLL [16], a simple propositional calculus modeling the main features of the DPLL procedure.

A very useful feature of the DPLL procedure and, by extension, of the DPLL calculus is its ability to provide a (Herbrand) model of the input formula whenever that formula is satisfiable. The procedure generates this model incrementally as it goes. The $\mathcal{ME}$ calculus can be seen as lifting this model generation process to the first-order level. Its goal is to construct a Herbrand model of a given set Φ of clauses, if any such model exists. To do that, during a derivation the calculus maintains a *context* Λ, a finite set of (possibly non-ground) literals. The context Λ is a finite—and compact—representation of a Herbrand interpretation I_Λ serving as a candidate model for Φ. The induced interpretation I_Λ might not be a model of Φ because it does not satisfy some clauses in Φ. The purpose of the main rules of the calculus is to detect this situation and either *repair* I_Λ, by modifying Λ, so that it becomes a model of Φ, or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number of simplification rules whose purpose is, again like in DPLL, to simplify the clause set and, as a consequence, to speed up the computation.

The $\mathcal{ME}$ calculus starts with a default candidate model, one that satisfies no positive literals, and "evolves it" as needed until it becomes an actual model of the input clause set Φ, or until it is clear that Φ has no models at all. An important by-product of this evolution process is that all terminating derivations of a satisfiable clause set Φ produce a context whose induced interpretation is indeed a model of Φ. This makes the calculus well suited for all applications in which it is important to also provide counter-examples to invalid statements, as opposed to simply proving their invalidity.

The calculus is refutationally sound and complete: an input clause set Φ is unsatisfiable if and only if the calculus (finitely) fails to find a model for Φ. While the calculus is obviously non-terminating for arbitrary input sets, it is terminating for the class of ground clauses (of course), and for the class of clauses resulting from the translation of conjunctions of Bernays-Schönfinkel formulas into clause form.

As mentioned, the $\mathcal{ME}$ calculus is already a significant improvement over FDPLL because it is a more faithful lifting of the DPLL procedure, having additional rules for simplifying the current clause set and the current context. Another advantage over FPDLL is that it contains a more systematic and general treatment of *universal literals*, one of FDPLL's optimizations. As we will see, adding universal literals to a context imposes strong restrictions on future modifications of that context, with the consequence of greatly reducing the non-determinism of the calculus.

This paper is organized as follows. After some formal preliminaries, given below, we describe in Section 2 the DPLL calculus, a declarative version of the DPLL procedure. We then define and discuss the Model Evolution calculus in Section 3 as a first-order extension of DPLL. We sketch a proof of correctness for the calculus in Section 4.[1] Then we show in Section 5 how the calculus compares to other calculi in related work, and we conclude the paper with directions for further research.

Formal Preliminaries. Most of the notions and notation we use in this paper are the standard ones in the field. We report here only notable differences and additions.

We will use two disjoint, infinite sets of variables: a set X of *universal* variables, which we will refer to just as variables, and another set V, which we will always refer to as *parameters*. We will use u,v to denote elements of V and x,y to denote elements of X. We fix a signature Σ throughout the paper and denote by Σ^{sko} the expansion of Σ obtained by adding to Σ an infinite number of fresh (Skolem) constants. Unless otherwise specified, when we say term we will mean Σ^{sko}-term. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables and by $\mathcal{P}ar(t)$ the set of t's parameters. A term t is *ground* iff $\mathcal{V}ar(t) = \mathcal{P}ar(t) = \emptyset$. All of the above is extended to literals and clauses in the obvious way.

A substitution ρ is a *renaming on* $W \subseteq (V \cup X)$ iff its restriction to W is a bijection of W onto itself; ρ is simply a *renaming* if it is a renaming on $V \cup X$. A substitution σ is *p-preserving* (short for parameter preserving) if it is a renaming on V. If s and t are two terms, we write $s \gtrsim t$, iff there is a substitution σ such that $s\sigma = t$. We say that s is *a variant of* t, and write $s \approx t$, iff $s \gtrsim t$ and $t \gtrsim s$ or, equivalently, iff there is a renaming ρ such that $s\rho = t$. We write $s \gnsim t$ if $s \gtrsim t$ but $s \not\approx t$ We write $s \geq t$ and say that t is *a p-instance of* s iff there is a p-preserving substitution σ such that $s\sigma = t$. We say that s is *a p-variant of* t, and write $s \simeq t$, iff $s \geq t$ and $t \geq s$; equivalently, iff there is a p-preserving renaming ρ such that $s\rho = t$.We write $s \gneq t$ if $s \geq t$ but $s \not\simeq t$. Again, all of the above is extended from terms to literals in the obvious way.

We denote literals by the letters K, L. We denote by $\overline{L}$ the complement of a literal L, and by L^{sko} the result of replacing each variable of L by a fresh Skolem constant in $\Sigma^{\mathrm{sko}} \setminus \Sigma$. We denote clauses by the letters C and D, and the empty clause by $\Box$. We will write $L \vee C$ to denote a clause obtained as the disjunction of a (possibly empty) clause C and a literal L. When convenient, we will treat a clause as the set of its literals. A *(Herbrand) interpretation* I is a set of ground Σ^{sko}-literals that contains either L or $\overline{L}$, but not both, for every ground Σ^{sko}-literal L. Satisfiability/validity of literals and clauses in a Herbrand interpretation is defined as usual.

2 The DPLL Calculus

The DPLL procedure [7] can be used to decide the satisfiability of finite sets of ground (or propositional) clauses. Following [16], the essence of the procedure can be captured by a sequent-style calculus, the DPLL calculus, consisting of the derivation rules given below. The calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Λ, the *context* of the sequent, is a finite set of ground literals and Φ is a finite (multi)set of ground clauses.

[1] A complete and detailed proof can be found in the long version of this paper [4].

$$\text{Split}\ \frac{\Lambda \vdash \Phi,\, L \vee C}{\Lambda, L \vdash \Phi,\, L \vee C \qquad \Lambda, \overline{L} \vdash \Phi,\, L \vee C}\ \text{if} \begin{cases} C \neq \Box, \\ L \notin \Lambda,\ \overline{L} \notin \Lambda \end{cases}$$

$$\text{Assert}\ \frac{\Lambda \quad \vdash \Phi,\, L}{\Lambda, L \vdash \Phi,\, L}\ \text{if} \begin{cases} L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases} \qquad \text{Subsume}\ \frac{\Lambda,\, L \vdash \Phi,\, L \vee C}{\Lambda,\, L \vdash \Phi}$$

$$\text{Empty}\ \frac{\Lambda \vdash \Phi,\, \Box}{\Lambda \vdash \Box}\ \text{if}\ \Phi \neq \emptyset \qquad \text{Resolve}\ \frac{\Lambda,\, \overline{L} \vdash \Phi,\, L \vee C}{\Lambda,\, \overline{L} \vdash \Phi,\, C}$$

The intended goal of the calculus is to derive a sequent of the form $\Lambda \vdash \emptyset$ from an initial sequent $\emptyset \vdash \Phi_0$, where Φ_0 is a clause set to be checked for satisfiability. If that is possible, then Φ_0 is satisfiable; otherwise, Φ_0 is unsatisfiable. Informally, the purpose of the context Λ is to store incrementally a set of *asserted literals*, i.e., a set of literals in Φ_0 that must or can be true for Φ_0 to be satisfiable. When $\Lambda \vdash \emptyset$ is derivable from $\emptyset \vdash \Phi_0$, the context Λ determines a (Herbrand) model of Φ_0: one that satisfies an atom p in Φ_0 iff p occurs positively in Λ.

The context is grown by the Assert and the Split rules. The Split rule corresponds to the decomposition in smaller subproblems of the DPLL procedure. The unit propagation process of the DPLL procedure (see, e.g., [18]) is modeled by the Assert, Resolve and Subsume rules. The Resolve rule can be used to remove from a clause all literals whose complement has been asserted, whereas the Subsume rule can be used to remove from a clause set all clauses containing an asserted literal.

The DPLL calculus is easily proven sound, complete and terminating. It can be shown [16] that the calculus maintains its completeness even if the Split rule is constrained to split on positive literals only. Another change that does not alter the calculus in any fundamental way is the replacement of the Empty rule by the more powerful rule

$$\text{Close}\ \frac{\Lambda \vdash \Phi,\, L_1 \vee \cdots \vee L_n}{\Lambda \vdash \Box}\ \text{if} \begin{cases} \Phi \neq \emptyset \text{ or } n > 0, \\ \overline{L_1}, \ldots, \overline{L_n} \in \Lambda \end{cases}$$

which reduces to the Empty rule given earlier when $n = 0$ and, in turn, can be simulated by n applications of Resolve followed by one application of Empty. The Model Evolution calculus, described next, is a lifting of the version of DPLL that applies the positive-literal restriction on Split and uses Close in place of Empty.

3 The Model Evolution Calculus

The Model Evolution calculus works with sequents of the form $\Lambda \vdash \Phi$, similarly to DPLL. This time, however, Λ is a finite set of literals possibly with variables or with parameters, called again a context, and Φ is a set of clauses possibly with variables. The defining feature of the calculus, modeled after FDPLL, is the way DPLL contexts are extended to the first-order case, and the rôle they play in driving the derivation and the model generation process.

Definition 3.1 (Context). *A* context *is a set of the form $\{\neg v\} \cup S$ where $v \in V$ and S is a finite set of literals each of which is parameter-free or variable-free.*

Where L is a literal and Λ a context, we write $L \in_{\simeq} \Lambda$ if L is a p-variant of a literal in Λ, and write $L \in_{\geq} \Lambda$ if L is a p-instance of a literal in Λ.

Definition 3.2 (Contradictory). *A literal L is* contradictory with *a context Λ iff $L\sigma = \overline{K}\sigma$ for some $K \in_{\simeq} \Lambda$ and some p-preserving substitution σ. A context Λ is* contradictory *iff it contains a literal that is contradictory with Λ.*

Example 3.3. Let $\Lambda := \{\neg v,\ p(x_1,y_1),\ \neg q(v_1)\}$. Then $\neg p(h(x),u)$, $\neg p(v,u)$, and $q(y)$ are all contradictory with Λ. However, $q(f(v))$ and $r(x)$, say, are not. (Recall that x, x_1, y_1 are variables while v, v_1, u are parameters.)

We will work only with non-contradictory contexts. Thanks to the next two notions, such contexts can be used as finite denotations of (certain) Herbrand interpretations.

Definition 3.4 (Most Specific Generalization). *Let L be a literal and Λ a context. A literal K is* a most specific generalization (msg) of L in Λ *iff $K \gtrsim L$ and there is no $K' \in \Lambda$ such that $K \gtrnsim K' \gtrsim L$.*

Definition 3.5 (Productivity). *Let L be a literal, C a clause, and Λ a context. A literal K* produces L in Λ *iff (i) K is an msg of L in Λ, and (ii) there is no $K' \in_{\geq} \Lambda$ such that $K \gtrnsim \overline{K'} \gtrsim L$. The context Λ* produces L *iff it contains a literal K that produces L in Λ.*

Example 3.6. Let $\Lambda := \{\neg v,\ p(v_1,g(u_1)),\ \neg p(v_1,g(v_1)),\ q(h(u),v),\ \neg q(u,g(v))\}$. The literals $\neg p(v,u)$, $p(v,g(u))$, $p(x,g(a))$, $\neg p(a,g(a))$ are all produced by Λ, specifically by $\neg v$, $p(v_1,g(u_1))$, $p(v_1,g(u_1))$, $\neg p(v_1,g(v_1))$, respectively. On the other hand, the literals $p(v,u)$, $\neg p(v,g(u))$, $\neg p(x,g(a))$, $p(a,g(a))$ are not produced by Λ. Note that, however, both $q(h(u),g(v))$ and $\neg q(h(u),g(v))$ are produced by Λ.

A consequence of the presence of the pseudo-literal $\neg v$ in every context Λ is that Λ produces L or $\overline{L}$ for every literal L. We can use this fact to associate to Λ a unique Herbrand interpretation.

Definition 3.7 (Induced interpretation). *Let Λ be a non-contradictory context. The* interpretation induced by Λ, *denoted by I_Λ, is the Herbrand interpretation that satisfies a positive ground literal L iff L is produced by Λ.*

Note that since it is possible for a context Λ to produce both a ground literal L and its complement $\overline{L}$, the above definition is asymmetric, because I_Λ always chooses to satisfy L over $\overline{L}$ if L is positive. Contrapositively, this means that if I_Λ satisfies a ground literal L and L is positive, then L and possibly also $\overline{L}$ are produced by Λ. If on the other hand L is negative, then L but not $\overline{L}$ is produced by Λ.

It should be clear now that the purpose of the pseudo-literal $\neg v$ in a context Λ is to provide a *default* truth-value to those ground literals whose value is not determined by the rest of the context. In fact, consider a ground literal L such that neither L nor $\overline{L}$ is produced by $\Lambda \setminus \{\neg v\}$. If L is positive, then it is false in I_Λ because it is not produced by Λ at all. If L is negative, then it is true in I_Λ because it is produced by $\neg v$.

For a given sequent $\Lambda \vdash \Phi$ the interpretation induced by the context Λ may falsify a clause of Φ. This situation is detectable through the computation of *context unifiers*.

Definition 3.8 (Context Unifier). *Let Λ be a context and $C = L_1 \vee \cdots \vee L_m \vee \cdots \vee L_n$ a parameter-free clause, where $0 \le m \le n$. A substitution σ is* a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ *if there are fresh $K_1, \ldots, K_n \in_\simeq \Lambda$ such that*

1. *σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,*
2. *$(\mathcal{P}ar(K_i))\sigma \subseteq V$ for $i = 1, \ldots, m$ and $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$ for $i = m+1, \ldots, n$.*

We say, in addition, that σ is productive *if K_i produces $\overline{L_i}\sigma$ in Λ for all $i = 1, \ldots, n$. We say that σ is* admissible (for Split) *if for all distinct $i, j = m+1, \ldots, n$, $L_i\sigma$ is parameter- or variable-free and $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.*

Note that each context unifier has a unique remainder. If σ is a context unifier with remainder D we call each literal of D *a remainder literal* of σ.

Example 3.9. Let $\Lambda := \{\neg v,\ p(v_1, u_1),\ \neg p(x_1, g(x_1)),\ q(v_2, g(v_2))\}$ and let $C_1 := r(x) \vee \neg p(x,y)$. The substitutions $\sigma_1 := \{v \mapsto r(x),\ v_1 \mapsto x,\ u_1 \mapsto y\}$ and $\sigma_2 := \{v \mapsto r(v_1),\ x \mapsto v_1,\ u_1 \mapsto y\}$ are both context unifiers of C_1 against Λ with respective remainders $r(x) \vee \neg p(x,y)$ and $r(v_1) \vee \neg p(v_1, y)$. While they are both productive, neither of them is admissible; the former because its remainder literals are not variable-disjoint, the latter because its remainder contains both variables and parameters.

By contrast, the substitution $\sigma_3 := \{v \mapsto r(v_1),\ x \mapsto v_1,\ y \mapsto u_1\}$ is a context unifier of C_1 against Λ, with remainder $r(v_1) \vee \neg p(v_1, u_1)$, that is both productive and admissible.

Consider now the clause $C_2 = \neg p(x,y) \vee \neg q(x,y)$. The substitution $\sigma_4 := \{v_1 \mapsto v_2,\ u_1 \mapsto g(u_2),\ x \mapsto v_2,\ y \mapsto g(u_2)\}$ is a context unifier of C_2 against Λ with remainder $\neg p(v_2, g(v_2))$. It is admissible, but it is not productive because the literal $p(v_1, u_1)$ of Λ chosen to unify with $\overline{\neg p(x,y)}$ does not produce $\overline{\neg p(x,y)}\sigma_4 = p(v_2, g(v_2))$.

Admissible context unifiers are fundamental in the Model Evolution calculus. With a context Λ and a clause C, the existence of an admissible context unifier of C against Λ is a sign that the induced interpretation I_Λ might not be a model of C. The discovery of an admissible context unifier σ of C against the current context Λ prompts the calculus to add a literal of $C\sigma$ to Λ, with the goal of making C valid in the new I_Λ. This literal is chosen only among the remainder literals of σ; non-remainder literals can be ignored with no loss of completeness. Note that while the existence of an admissible context unifier σ of C against Λ is necessary for the unsatisfiability of C in I_Λ, it is not sufficient unless σ is also productive. For completeness then, the calculus needs to consider only remainder literals of admissible unifiers that are also productive. [2]

Productivity issues aside, note that although context unifiers for a given clause C and context Λ are easily computable (they are just simultaneous most general unifiers), they are not unique and may not be admissible. Nevertheless, the calculus does not need to search for admissible context unifiers: any context unifier can be composed with a renaming substitution, determined deterministically, such that the resulting context unifier is admissible. The completeness of the calculus is not affected by computing admissible contexts unifiers this way. [3]

[2] But see [4] for a discussion on the usefulness of considering non-productive context unifiers as well.

[3] Again, see [4] for more details.

Parameters vs. Variables. Before moving to describe the rules of the Model Evolution calculus, it is important to clarify the respective rôles that parameters and variables play in the calculus. Each derivation in the calculus starts with a sequent of the form $\neg v \vdash \Phi_0$, where Φ_0 contain only standard clauses, i.e. clauses with no parameters—but possibly with variables. Similarly, all sequents generated during a derivation contain standard clauses only. Variables then can appear both in clauses sets and in contexts. Parameters instead can appear only in contexts. The rôle of variables within a clause is the usual one. In contrast, the rôle of variables and parameters within a context is to constrain, in different ways, how a candidate model can be repaired.

Now, given a sequent $\Lambda \vdash \Phi$, the interpretation I_Λ needs repairing only if it falsifies a clause C in Φ. In that case there is an admissible context unifier σ of C against Λ. If every instance of C falsified by I_Λ is also an instance of $C\sigma$, to make C valid in I_Λ it is enough to modify Λ so that I_Λ satisfies $C\sigma$. One way to do that is to pick from $C\sigma$ a literal $L\sigma$ that is not contradictory with Λ, and *assert* it by adding it to Λ. The idea is to make the unit clause $L\sigma$ valid in I_Λ, which then makes $C\sigma$ valid as well. Now recall that, since σ is admissible, the added literal will not contain both parameters and variables.

If $L\sigma$ is a parameter-free literal, a *universal literal* in FDPLL's terminology, the assertion of $L\sigma$ cannot be retracted. No repairs that involve making instances of $L\sigma$ false in the induced interpretation will be allowed from that point on.

If $L\sigma$ is variable-free instead, the assertion of $L\sigma$ expresses simply the conjecture that there is a model of $C\sigma$ satisfying all ground instances of $L\sigma$. This conjecture can be partially revised later if evidence against it is found. This might happen if the calculus later adds to the current context Λ' a literal $\overline{L}\sigma'$, for some context unifier σ', and $C\sigma'$ is an instance of $C\sigma$. After the addition, the induced interpretation satisfies only the instances of $L\sigma$ that are not an instance of $\overline{L}\sigma'$. At that point, $C\sigma$ may not be valid anymore because its instance $C\sigma'$ may now be falsified. If so, the calculus will detect it and will try to make $C\sigma'$ valid (thereby restoring the validity of $C\sigma$) by looking in $C\sigma'$ for a literal other than $L\sigma'$ that can be added to the context, as explained earlier for $L\sigma$.

Derivation Rules. The Model Evolution calculus lifts the DPLL calculus to the first-order level by providing a first-order version of DPLL's rules—Split with the positive literal restriction, Assert, Subsume, Resolve and Close—and adding one new simplification rule, Compact, specific to the first-order case.

In [4] we show that, except for the presence of the pseudo-literal $\neg v$ in its contexts, the Model Evolution calculus reduces precisely to DPLL when the input clause set is ground. We refer the interested reader to [4] for more details on why this is the case. In brief, the reason is that in the ground case Compact never applies, and the other rules reduce exactly to their namesake in DPLL.

A definition and a brief explanation of $\mathcal{ME}$'s rules follows next.

$$\text{Assert} \quad \frac{\Lambda \quad \vdash \Phi,\, L}{\Lambda, L \vdash \Phi,\, L} \quad \text{if } K \geq L \text{ for no } K \in \Lambda, \text{ and } L \text{ is not contradictory with } \Lambda$$

As in DPLL, the Assert rule is extremely useful in reducing the non-determinism of the calculus. Every candidate model of a clause set $\Phi \cup \{L\}$ *must* make L valid in order to become a model of $\Phi \cup \{L\}$. The Assert rule achieves just that by adding L to the context. Note that since L is parameter-free, its addition to the context is not retractable.

Also note that the rule does not apply if the (permanent) validity of L has been already established. This is the case when Λ contains a—necessarily parameter-free—literal K such that $K \geq L$. The rule does not apply also if L is contradictory with Λ. In that case, however, the candidate model is unrepairable. The Close rule will detect that.

$$\textsf{Split}\ \frac{\Lambda \vdash \Phi, C \vee L}{\Lambda, L\sigma \vdash \Phi, C \vee L \qquad \Lambda, (\overline{L\sigma})^{\mathrm{sko}} \vdash \Phi, C \vee L}\ \text{if}\ \begin{cases} C \neq \Box, \\ \sigma \text{ is an admissible context} \\ \text{unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\mathrm{sko}} \text{ is} \\ \text{contradictory with } \Lambda \end{cases}$$

As in DPLL, Split is the only (*don't-know*) non-deterministic rule of the calculus. Split is the rule that discovers when the current candidate model falsifies one of the clauses in the current clause set. It does so by computing a context unifier σ with non-empty remainder for a clause with at least two literals. Once it finds σ, it attempts to repair the model by selecting a remainder literal $L\sigma$ and adding either $L\sigma$ or its complement to the context. Adding the complement of $L\sigma$ in alternative to $L\sigma$ is necessary for soundness, as the current clause set may have no models that satisfy $L\sigma$. Of course, the addition of $L\sigma$'s complement to the context will not make the selected clause $C \vee L$ valid in the new candidate model. But it will make sure that no context unifier σ' of $C \vee L$ has $L\sigma'$ in its remainder, forcing the calculus to select other literals to make $C \vee L$ valid. Note that Split does not quite add the complement of $L\sigma$: when $L\sigma$ is parameter-free it adds a Skolemized version of $\overline{L\sigma}$.[4] This is in accordance to our treatment of parameter-free literals in contexts as universal sentences.

$$\textsf{Subsume}\ \frac{\Lambda, K \vdash \Phi, L \vee C}{\Lambda, K \vdash \Phi}\ \text{if}\ K \geq L.$$

The purpose of Subsume is the same as in DPLL: to get rid of clauses that are valid in the current candidate model—and are guaranteed to stay so. These are exactly those clauses containing a p-instance L of a—necessarily parameter-free—literal K in the current context. Although Subsume is not needed for completeness, it is useful in practice because it reduces the size of the current clause set.

$$\textsf{Resolve}\ \frac{\Lambda, \overline{K} \vdash \Phi, L \vee C}{\Lambda, \overline{K} \vdash \Phi, C}\ \text{if}\ K \geq L.$$

The Close rule is in essence the dual of Subsume, and like Subsume it is not needed for completeness. Its main goal is to generate unit clauses, which can then be added to the context by Assert as parameter-free literals.

$$\textsf{Compact}\ \frac{\Lambda, K, L \vdash \Phi}{\Lambda, K \quad \vdash \Phi}\ \text{if}\ K \geq L$$

[4] When $L\sigma$ is variable-free the Skolemization step is vacuous.

The Compact rule as well is unnecessary for completeness but useful in practice. To understand the rule's rationale it is important to know that, the way the calculus is defined, Compact's precondition holds only if K is a parameter-free literal. As discussed in a previous section, parameter-free context literals stand for all their instances, with no exception. This means that when a parameter-free literal K is added to a context, all literals in the context that are an instance of K become superfluous. The purpose of Compact is to eliminate these superfluous literals.

$$\mathsf{Close}\quad \frac{\Lambda \vdash \Phi, C}{\Lambda \vdash \Box} \quad \text{if} \begin{cases} \Phi \neq \emptyset \text{ or } C \neq \Box, \\ C \text{ has a context unifier against } \Lambda \text{ with an empty remainder} \end{cases}$$

The idea behind Close is that when its precondition holds there is no way to repair the current candidate model to make it satisfy C. The replacement of the current close set by the empty clause signals that the calculus has given up on that candidate model. Note that, because of Resolve, it is possible for the calculus to generate a sequent containing an empty clause among other clauses. The Close rule recognizes such sequents and applies to them as well. To see that it is enough to observe that, for any context Λ, the empty substitution is a context unifier of $\Box$ against Λ with an empty remainder.

A Derivation Example. We show a simple example of a derivation of an unsatisfiable clause set. We sketch the construction of only one branch of the final derivation tree, as the other branches are constructed similarly. Consider the following initial sequent (where we use the usual mathematical notation for greater clarity):

$$\neg v \vdash \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x),\ |x| \geq 0,\ 0 \geq -|x|, \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x),\ \neg(|c| \geq c) \vee \neg(|c| \geq -|c|) \end{array}$$

Each unit clause in the clause set can be moved into the context by means of Assert, and then removed from the set by means of Subsume. This results in the sequent:

$$\neg v,\ |x| \geq 0,\ 0 \geq -|x| \vdash \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x), \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x), \\ \neg(|c| \geq c) \vee \neg(|c| \geq -|c|) \end{array}$$

By considering the fresh p-variants $|x_1| \geq 0, 0 \geq -|x_2|, \neg v_1$ of context literals, the substitution $\sigma = \{x \mapsto |x_1|,\ y \mapsto 0,\ z \mapsto -|x_2|,\ v_1 \mapsto |x_1| \geq -|x_2|\}$ is an admissible context unifier of $\neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z)$ with residue $|x_1| \geq -|x_2|$. Since neither $|x_1| \geq -|x_2|$ nor its complement is contradictory with the context, we can add it the context by one application of Split. With $|x_1| \geq -|x_2|$ in the context, we can then simplify $\neg(|c| \geq c) \vee \neg(|c| \geq -|c|)$ to $\neg(|c| \geq c)$ with Resolve, and then move $\neg(|c| \geq c)$ to the context by means of Assert and Subsume, obtaining

$$\begin{array}{l} \neg v,\ |x| \geq 0,\ 0 \geq -|x| \\ |x_1| \geq -|x_2|,\ \neg(|c| \geq c) \end{array} \vdash \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x), \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x), \end{array}$$

Using $\neg v$ and $\neg(|c| \geq c)$ we can apply Split to $\neg(x \geq 0) \vee (|x| \geq x)$ with remainder $\neg(|c| \geq 0)$, and then to $\neg(0 \geq x) \vee (|x| \geq x)$ with remainder $\neg(0 \geq |c|)$, and obtain:

$$\ldots,\ \neg(|c| \geq 0),\ \neg(0 \geq |c|) \vdash \ldots,\ (x \geq 0) \vee (0 \geq x),\ \ldots$$

At this point we can apply the Close rule because, with the context literals $\neg(|c| \geq 0)$ and $\neg(0 \geq |c|)$, the substitution $\sigma = \{x \mapsto |c|\}$ is a context unifier of $(x \geq 0) \vee (0 \geq x)$ with an empty remainder.

4 Correctness of the Calculus

In this section, we sketch a proof of soundness and completeness for the Model Evolution calculus. For that first we need a proper notion of derivation.

Derivations in the Model Evolution calculus are defined in terms of *derivation trees*, where each node corresponds to a particular application of a derivation rule, and each of the node's children corresponds to one of the conclusions of the rule. More precisely, a derivation tree in the $\mathcal{ME}$ calculus is a labeled tree inductively defined as follows.

A one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set. A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$ *is derived from* $\mathbf{T}$. We say that a derivation tree $\mathbf{T}$ is a *derivation tree of a clause set* Φ iff its root node tree is labeled with $\neg v \vdash \Phi$.

We say that a branch in a derivation tree is *closed* if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$; otherwise, the branch is *open*. A derivation tree is *closed* if each of its branches is closed, and it is *open* otherwise. We say that a derivation tree (of a clause set Φ) is a *refutation tree (of* Φ*)* iff it is closed.

In the rest of the paper, the letter κ will denote an ordinal smaller than or equal to the first infinite ordinal.

Definition 4.1 (Derivation). *A* derivation (in $\mathcal{ME}$) *is a possibly infinite sequence of derivation trees* $(\mathbf{T}_i)_{i<\kappa}$*, such that for all i with* $0 < i < \kappa$, $\mathbf{T}_i$ *is derived from* $\mathbf{T}_{i-1}$.

We say that a derivation $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ is a *derivation of a clause set* Φ iff $\mathbf{T}_0$ is a one-node tree with label $\{\neg v\} \vdash \Phi$. We say that $\mathcal{D}$ is a *refutation of* Φ iff $\mathcal{D}$ is finite and ends with a refutation tree of Φ.

We show in the next sections that for all sets Φ_0 of Σ-clauses with no parameters, Φ_0 is unsatisfiable iff Φ_0 has a refutation in the calculus.

Soundness. To prove the calculus sound we use the fact that its derivation rules preserve a particular notion of satisfiability which we call *a-satisfiability*, after [3].

Let us fix a constant a from the signature $\Sigma^{\text{sko}} \setminus \Sigma$. Given a literal L, we denote by L^a the result or replacing every parameter of L by a. Similarly, given a context Λ, we denote by Λ^a the set of *unit clauses* obtained from Λ by removing the pseudo-literal $\neg v$ and replacing each literal L of Λ with the unit clause L^a. We say that a sequent $\Lambda \vdash \Phi$ is *a-(un)satisfiable* iff the clause set $\Lambda^a \cup \Phi$ is (un)satisfiable in the standard sense—that is, has no (Herbrand) model.

Lemma 4.2. *For each rule of the* $\mathcal{ME}$ *calculus, if the premise of the rule is a-satisfiable, then one of its conclusions is a-satisfiable as well.*

Proposition 4.3 (Soundness). *For all sets* Φ_0 *of parameter-free* Σ*-clauses, if* Φ_0 *has a refutation tree* $\mathbf{T}$*, then* Φ_0 *is unsatisfiable.*

Fairness. As customary, we prove the completeness of the calculus with respect to *fair derivations*. The specific notion of fairness that we adopt is defined formally as follows. For that, it will be convenient to describe a tree $\mathbf{T}$ as the pair $(\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is the set of the nodes of $\mathbf{T}$ and $\mathbf{E}$ is the set of the edges of $\mathbf{T}$. Each derivation $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa} = ((\mathbf{N}_i, \mathbf{E}_i))_{i<\kappa}$ in $\mathcal{ME}$ determines a *limit tree* $\mathbf{T} := (\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$. It is easy to show that a limit tree of a derivation $\mathcal{D}$ is indeed a tree. But note that it will not be a derivation tree unless $\mathcal{D}$ is finite.

Definition 4.4 (Persistency). *Let* $\mathbf{T}$ *be the limit tree of some derivation, and let* $\mathbf{B} = (N_i)_{i<\kappa}$ *be a branch in* $\mathbf{T}$ *with* κ *nodes. Let* $\Lambda_i \vdash \Phi_i$ *be the sequent labeling node* N_i*, for all* $i < \kappa$*. We define the* persistent context literals *of* $\mathbf{B}$ *as* $\Lambda_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i \le j < \kappa} \Lambda_j$*, and the* persistent clauses *of* $\mathbf{B}$ *as* $\Phi_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i \le j < \kappa} \Phi_j$*.*

Although, strictly speaking, $\Lambda_{\mathbf{B}}$ is not a context because it may be infinite, for the purpose of the completeness proof we treat it as one. This is possible because all relevant definitions (Definitions 3.1 to 3.8) can be applied without change to $\Lambda_{\mathbf{B}}$ as well.

Fair derivations in the $\mathcal{ME}$ calculus are defined in terms of *exhausted branches*.

Definition 4.5 (Exhausted branch). *Let* $\mathbf{T}$ *be a limit tree, and let* $\mathbf{B} = (N_i)_{i<\kappa}$ *be a branch in* $\mathbf{T}$ *with* κ *nodes. For all* $i < \kappa$*, let* $\Lambda_i \vdash \Phi_i$ *be the sequent labeling node* N_i*. The branch* $\mathbf{B}$ *is* exhausted *iff for all* $i < \kappa$ *all of the following hold:*

(i) *For all* $C \in \Phi_{\mathbf{B}}$*, if* Split *is applicable to* $\Lambda_i \vdash \Phi_i$ *with selected clause* C *and productive context unifier* σ*, then there is a remainder literal* L *of* σ *and a* $j \ge i$ *with* $j < \kappa$ *such that* Λ_j *produces* L *but does not produce* $\overline{L}$*.*

(ii) *For all unit clauses* $L \in \Phi_{\mathbf{B}}$*, if* Assert *is applicable to* $\Lambda_i \vdash \Phi_i$ *with selected unit clause* L*, then there is a* $j \ge i$ *with* $j < \kappa$ *such that* $L \in_{\ge} \Lambda_j$*.*

(iii) *For all* $C \in \Phi_{\mathbf{B}}$*,* Close *is not applicable to* $\Lambda_i \vdash \Phi_i$ *with selected clause* C*.*

(iv) $\Phi_i \neq \{\Box\}$*.*

Definition 4.6 (Fairness). *A limit tree of a derivation is* fair *iff it is a refutation tree or it has an exhausted branch. A derivation is* fair *iff its limit tree is fair.*

We point out that fair derivations in the sense above exist and are computable for any set of (parameter-free) Σ-clauses. A proof of this fact can be given by adapting a technique used in [3] to show the computability of fair derivations in FDPLL.

Completeness. For the rest of this section, let Φ be a set of parameter-free Σ-clauses and assume that $\mathcal{D}$ is a fair derivation of Φ that is not a refutation. Observe that $\mathcal{D}$'s limit tree must have at least one exhausted branch. We denote this branch by $\mathbf{B} = (N_i)_{i<\kappa}$. Then, by $\Lambda_i \vdash \Phi_i$, we will always mean the sequent labeling the node N_i in $\mathbf{B}$, for all $i < \kappa$. (As a consequence, we will also have that $\Lambda_0 = \{\neg v\}$ and $\Phi_0 = \Phi$.)

The following proposition is the main result for proving the calculus complete.

Proposition 4.7. *If* $\Box \notin \Phi_{\mathbf{B}}$*, then* $I_{\Lambda_{\mathbf{B}}}$ *is a model of* $\Phi_{\mathbf{B}}$*.*

Proof. (Sketch) Suppose ad absurdum that $\Phi_{\mathbf{B}}$ does not contain the empty clause, but $I_{\Lambda_{\mathbf{B}}}$ is not a model of $\Phi_{\mathbf{B}}$. This means that there is a ground instance $C\gamma$ of a clause $C = L_1 \vee \cdots \vee L_n$ with $n \ge 1$ from $\Phi_{\mathbf{B}}$ that is not satisfied by $I_{\Lambda_{\mathbf{B}}}$. This is to say that the literals $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$ are all satisfied by $I_{\Lambda_{\mathbf{B}}}$. It can be shown that $\Lambda_{\mathbf{B}}$ produces $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$ then.

We distinguish two complementary cases, depending on whether $n = 1$ or $n > 1$, and show that they both lead to a contradiction.

($n = 1$) C consists of the single literal L_1. Given that $\Lambda_{\mathbf{B}}$ produces $\overline{L_1}\gamma$, it can be shown that there is a $K \in \Phi_{\mathbf{B}}$ and i such that for all $j \geq i$ with $j < \kappa$, $K \in \Lambda_j$ and K produces $\overline{L_1}\gamma$ in Λ_j. Since L_1 is a (unit) clause from $\Phi_{\mathbf{B}}$, there is an i' such that $L_1 \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality assume that $i \geq i'$. By Definition 4.5-(iii), Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L_1. Since $L_1 \in \Phi_i$, this entails that all context unifiers of L_1 against Λ_i have a non-empty remainder. With the result above about K it can be shown that Assert is applicable to $\Lambda_i \vdash \Phi_i$ with selected unit clause L_1.

According to Definition 4.5-(ii) then, there is a $j \geq i$ with $j < \kappa$ and an $L \in \Lambda_j$ such that $L \geq L_1$. It is not difficult to see that with L_1 being parameter-free, L must be parameter-free as well. From this and the fact that $L \geq L_1\gamma$ we can then show that Λ_j produces $L_1\gamma$ but not $\overline{L_1}\gamma$, which contradicts the assertion above that for some literal K and all $j \geq i$, $K \in \Lambda_j$ produces $\overline{L_1}\gamma$.

($n > 1$) By a suitable lifting lemma, there exist fresh p-variants $K_1, \ldots, K_n \in_{\simeq} \Lambda_{\mathbf{B}}$ and a substitution σ such that (i) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$, (ii) for all $k = 1, \ldots, n$, $L_k \gtrsim L_k\sigma \gtrsim L_k\gamma$, and (iii) for all $k = 1, \ldots, n$, K_k produces $\overline{L_k}\sigma$ in $\Lambda_{\mathbf{B}}$. By Definition 3.8, σ is a productive context unifier of C against $\Lambda_{\mathbf{B}}$ with some remainder D. It is not difficult to prove that then an admissible context unifier of C against $\Lambda_{\mathbf{B}}$ can be obtained as $\sigma' = \sigma\rho$, for some renaming ρ. Let $k \in \{1, \ldots, n\}$ and observe that a literal K produces a literal L in a context Λ iff K produces a variant of L in Λ. From the fact that K_k produces $\overline{L_k}\sigma$ in $\Lambda_{\mathbf{B}}$, we have that K_k produces $\overline{L_k}\sigma'$ in $\Lambda_{\mathbf{B}}$ as well. Similarly to the case $n = 1$, it can be proven that there is an i such that for all $k = 1, \ldots, n$ and all $j \geq i$ with $j < \kappa$, $K_k \in \Lambda_j$ and K_k produces $\overline{L_k}\sigma'$. By assumption, C is a clause of $\Phi_{\mathbf{B}}$. Hence, there is a i' such that $C \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality suppose that $i \geq i'$. By Definition 4.5-(iii), Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. Hence, all context unifiers of C against Λ_i must have a non-empty remainder. By the above, Λ_i produces $\overline{L_k}\sigma'$ for $k = 1, \ldots, n$, and so, in particular, Λ_i produces all remainder literals of σ'. It can be shown then, that Split is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and productive context unifier σ'. By Definition 4.5-(i), there is then a remainder literal $L\sigma'$ of σ' and a $j \geq i$ such that Λ_j produces $L\sigma'$ but not $\overline{L}\sigma'$. This contradicts the conclusion above that for all $k = 1, \ldots, n$, $K_k \in \Lambda_j$ produces $\overline{L_k}\sigma'$ in Λ_j. $\square$

The completeness of the calculus is a consequence of Proposition 4.7. We state it here in contrapositive form to underline the model computation abilities of $\mathcal{ME}$.

Theorem 4.8 (Completeness). *Let $\mathcal{D}$ be a fair derivation of Φ with limit tree $\mathbf{T}$. If $\mathbf{T}$ is not a refutation tree, then Φ is satisfiable; more precisely, for every exhausted branch $\mathbf{B}$ of $\mathbf{T}$, $I_{\Lambda_{\mathbf{B}}}$ is a model of Φ.*

When the branch $\mathbf{B}$ in Theorem 4.8 is finite, $\Lambda_{\mathbf{B}}$ coincides with the context Λ_n, say, in $\mathbf{B}$'s leaf. From a model computation perspective, this is a very important fact because it means that a model of the original clause set—or rather, a finite representation of it, Λ_n—is readily available at the end of the derivation; it does not have to be computed from the branch, as in other model generation calculi.

The calculus is proof confluent [5]: any derivation of an unsatisfiable clause set extends to a refutation. In fact, because of the strong completeness result in Theorem 4.8, the calculus satisfies an even stronger property, as illustrated by the corollary below. In practical terms, the corollary implies that as long as a derivation strategy guarantees fairness, the order of application of the rules of the calculus is irrelevant for proving an input clause set unsatisfiable, giving to the $\mathcal{ME}$ calculus the same kind of flexibility enjoyed by the DPLL calculus at the propositional level.

Corollary 4.9 (Proof Convergence). *Let Φ be a a parameter-free clause set over the signature Σ. If Φ is unsatisfiable, then every fair derivation of Φ is a refutation.*

5 Conclusions

In this paper we have introduced the Model Evolution ($\mathcal{ME}$) calculus. The $\mathcal{ME}$ calculus extends the propositional part of the DPLL procedure to first-order clause logic by supplying unification-based, first-order versions of DPLL's inference rules. Compared to its most immediate predecessor, FDPLL [3], $\mathcal{ME}$ is a more faithful lifting of DPLL, as it also includes first-order versions of the unit propagation rules, which are not present in FDPLL. Two additional improvements of $\mathcal{ME}$ over FDPLL, both leading to a smaller search space, are the absence of a rule like FDPLL's Commit, and the ability of Split to generate universal literals more often than in FDPLL.

Related work. Besides the FDPLL calculus, $\mathcal{ME}$ is related to the family of *instance-based methods*.[5] Proof search in instance-based methods relies on maintaining a set of instances of input clauses and analyzing it for satisfiability until completion. $\mathcal{ME}$ is *not* an instance-based method in this sense, as clause instances are used only temporarily within the Split inference rule and can be forgotten after the split has been carried out.

The contemporary stream of research on instance-based methods was initiated with the Hyperlinking calculus [11], whose current successor is Ordered Semantic Hyperlinking (OSHL) [14]. OSHL has many interesting features not present in $\mathcal{ME}$. In the intersection with $\mathcal{ME}$, OSHL can be described as a calculus that grows a set of ground clauses, based on the detection of input clause instances falsified by a current interpretation and a repair operation roughly comparable to $\mathcal{ME}$, however at the ground level.

Some instance-based calculi have been formulated within the (clausal) tableau framework. The initial work in this direction is Billon's disconnection method [6]. The calculus described in [2] relates to the disconnection method much like the hyper resolution calculus relates to the resolution calculus. The disconnection method has been picked up by Letz and Stenz for further improvements, which include a dedicated inference rule for deriving unit clauses [13]. Compared to $\mathcal{ME}$, tableau calculi branch on subformulas (or, the literals of a clause in the clausal logic case), as opposed to complementary literals like $\mathcal{ME}$ does. For the propositional case it is easy to see that branching on complementary literals is more general than branching on clauses. In fact, each branching on a clause with n literals can be simulated by n splits with complementary literals.

[5] The detailed discussion in [3] on FDPLL's related work extends to $\mathcal{ME}$ as well. The points made there will not be repeated here in detail. However, see the long version of the present paper [4] for an extended section on this related work.

Furthermore, some improvements like factoring (see [12]) are *automatically* realized by the branching on complementary literals approach. A systematic investigation on how this fact exactly carries over to the first-order case—i.e. $\mathcal{ME}$ vs. certain clausal tableau calculi—is left for future work.

Two variants of an instance-based method are described by Hooker *et al.* [10]. One of them, the “Primal Approach” seems to be very similar to the disconnection method (see above) although, unfortunately, the relation with this method is not made explicit in [10]. The other variant, the “Dual Approach”, differs from the former by the presence of *auxiliary* clauses of the form $K \to L$ generated during the proof search, where (K, L) is a connection of literals occurring in the current clause set. No simplification mechanisms have been described, like for instance those based on unit propagation rules. Finally, a rather abstract framework for instance-based calculi which also admits simplification techniques is described in [9].

A significant difference between the instance based methods we are aware of and the $\mathcal{ME}$ calculus is that the former maintain a growing set of instances of input *clauses*, while $\mathcal{ME}$ does maintain a growing set of instances of input *literals*: the current context. Since contexts grow more slowly than sets of clause instances, this may lead to an (at least) exponential advantage for $\mathcal{ME}$ regarding space consumption. As a drastic example, consider a clause C of the form $P_1(x_1) \vee \cdots \vee P_n(x_n)$ and assume a signature that includes m constants. There are clearly more than m^n different instances of C, and there seems to be no principled way to avoid including that many of them in the set of instances of input clauses (by nature of instance-based methods, clause subsumption cannot be used). In $\mathcal{ME}$ in contrast, since contexts never contain p-variants of the same literal, the number of instances of P_i-literals is at most $2n \cdot (m + 2)$.

Further Work. Our immediate goal is to implement the $\mathcal{ME}$ calculus and evaluate its potential in practice. In addition to that, various directions for further work are conceivable. We list some below, referring the interested reader to [4] for more details.

The inference rules of the $\mathcal{ME}$ calculus make sure that only literals that are parameter-free or variable-free are inserted into contexts. “Mixed” literals with parameters and variables presently occur in $\mathcal{ME}$ only temporarily, during the computation of branch unifiers. A possible improvement would involve admitting mixed literals in contexts, allowing then individual variables to be singled out as universal, as opposed to entire literals as it is now.

The $\mathcal{ME}$ calculus is proof convergent (cf. Corollary 4.9), and so the order of rule applications does not matter. This *don’t-care* nondeterminism can be exploited to have the calculus stepwise simulate certain other calculi such as, e.g., the propositional logic oriented OSHT calculus [17] or the Hyper Tableaux calculus in [2].

The most significant search heuristics for improving the performance of DPLL-based solvers are *learning* (i.e. the addition of dynamically generated lemmas to the input clause set) and *intelligent backtracking* of split choices. While the latter is straightforward to achieve within $\mathcal{ME}$, the former is not. In particular, a number of alternatives seem possible which need further theorical investigation and experimental evaluation.

As presented here, the calculus always starts with an interpretation that assigns false to all ground atoms. By simply replacing the pseudo-literal $\neg v$ by v, it is possible to have the calculus start instead with a complementary initial interpretation. The kind of semantic guidance achieved in OSHL [14] by means of a user-defined initial interpretation, is trivially achievable in $\mathcal{ME}$ when this interpretation is denotable by a context: one simply starts the derivation with that context. More work is needed to allow $\mathcal{ME}$ to start with arbitrary interpretations, in particular, ones that cannot be encoded into a (finite) context.

In many theorem proving applications, a proper treatment of equational theories or equality is mandatory. In principle, there seems to be nothing against a modern treatment of equality in $\mathcal{ME}$ by means of a superposition-style inference rule and of simplification rules based on rewriting [1].

References

1. L. Bachmair and H. Ganzinger. Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction. A Basis for Applications*, volume I. Kluwer, 1998.
2. P. Baumgartner. Hyper Tableaux—The Next Generation. In H. de Swaart, editor, *Proc. of TABLEAUX'98*, volume 1397 of *LNAI*, pages 60–76. Springer, 1998.
3. P. Baumgartner. FDPLL—A First-Order Davis-Putnam-Logeman-Loveland Procedure. In D. McAllester, editor, *Proc. of CADE-17*, volume 1831 of *LNAI*, Springer, 2000.
4. P. Baumgartner and C. Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, 2003.
5. W. Bibel. *Automated Theorem Proving*. Vieweg, 1982.
6. J.-P. Billon. The Disconnection Method. In P. Miglioli et al., editors, *Proc of TABLEAUX'96*, volume 1071 of *LNAI*, pages 110–126. Springer, 1996.
7. M Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
8. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
9. Harald Ganzinger and Konstantin Korovin. New directions in instance-based theorem proving. In *LICS - Logics in Computer Science*, 2003. To appear.
10. J. N. Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial Instantiation Methods for Inference in First Order Logic. *Journal of Automated Reasoning*, 28:371–396, 2002.
11. S.-J. Lee and D. Plaisted. Eliminating Duplicates with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.
12. R. Letz, K. Mayr, and C. Goller. Controlled Integrations of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning*, 13, 1994.
13. R. Letz and G. Stenz. Proof and Model Generation with Disconnection Tableaux. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of LPAR'01*, volume 2250 of *LNAI*. Springer, 2001.
14. D. A. Plaisted and Y. Zhu. Ordered Semantic Hyper Linking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.
15. G. Stenz. DCTP 1.2 - System Abstract. In U. Egly and C. G. Fermüller, editors, *Proc. of TABLEAUX'02*, volume 2381 of *LNAI*, pages 335–340. Springer, 2002.
16. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In G. Ianni and S. Flesca, editors, *Proc. of JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.
17. A. Yahya and D. A. Plaisted. Ordered Semantic Hyper-Tableaux. *Journal of Automated Reasoning*, 29(1):17–57, 2002.
18. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proc. of AI-MATH'96*, 1996.