Lemma Learning in the Model Evolution Calculus

Peter Baumgartner[1], Alexander Fuchs[2], and Cesare Tinelli[2]

[1] National ICT Australia (NICTA)
Peter.Baumgartner@nicta.com.au
[2] The University of Iowa, USA
{fuchs, tinelli}@cs.uiowa.edu

Abstract. The Model Evolution ($\mathcal{ME}$) Calculus is a proper lifting to first-order logic of the DPLL procedure, a backtracking search procedure for propositional satisfiability. Like DPLL, the ME calculus is based on the idea of incrementally building a model of the input formula by alternating constraint propagation steps with non-deterministic decision steps. One of the major conceptual improvements over basic DPLL is *lemma learning*, a mechanism for generating new formulae that prevent later in the search combinations of decision steps guaranteed to lead to failure. We introduce two lemma generation methods for $\mathcal{ME}$ proof procedures, with various degrees of power, effectiveness in reducing search, and computational overhead. Even if formally correct, each of these methods presents complications that do not exist at the propositional level but need to be addressed for learning to be effective in practice for $\mathcal{ME}$. We discuss some of these issues and present initial experimental results on the performance of an implementation of the two learning procedures within our $\mathcal{ME}$ prover *Darwin*.

1 Introduction

The Model Evolution ($\mathcal{ME}$) Calculus [5] is a proper lifting to first-order logic of the DPLL procedure, a backtracking search procedure for propositional satisfiability. Like DPLL, the calculus is based on the idea of incrementally building a model of the input formula by alternating constraint propagation steps with non-deterministic decision steps. Two of the major conceptual improvements over basic DPLL developed over the years are *backjumping*, a form of intelligent backtracking of wrong decision steps, and *lemma learning*, a mechanism for generating new formulae that prevent later in the search combinations of decision steps guaranteed to lead to failure.

Adapting backjumping techniques from the DPLL world to $\mathcal{ME}$ implementations is relatively straightforward and does lead to performance improvements, as our past experience with *Darwin*, our $\mathcal{ME}$-based theorem prover, has shown [2]. In contrast, adding learning capabilities is not immediate, first because one needs to lift properly to the first-order level both the notion of lemma and the lemma generation process itself, and second because any first-order lemma generation process adds a significant computation overhead that can offset the potential advantages of learning.

In this paper, we introduce two lemma learning procedures for $\mathcal{ME}$ with different degrees of power, effectiveness in reducing search, and computational overhead. Even if formally correct, each of these procedures presents issues and complications that do not arise at the propositional level and need to be addressed for learning to be effective for $\mathcal{ME}$. We mention some of these issues and then present initial experimental results on the performance of an implementation of the learning procedures within *Darwin*.

The $\mathcal{ME}$ calculus is a sequent-style calculus consisting of three basic derivation rules: Split, Assert and Close, and three more optional rules. To simplify the exposition we will consider here a restriction of the calculus to only the non-optional rules. The learning methods presented in this paper extend with minor modifications to $\mathcal{ME}$ derivations that use the optional rules as well. The derivation rules are presented in [5] and in more detail in [6]. We do not present them directly here because in this paper we focus on *proof procedures* for $\mathcal{ME}$, which are better described in terms of abstract transition systems (see Section 2). It suffices to say that Split, with two possible conclusions instead of one, is the only non-deterministic rule of the calculus, and that the calculus is proof-confluent, i.e., the rules may be applied in any order, subject to fairness conditions, without endangering completeness. Derivations in $\mathcal{ME}$ are defined as sequences of *derivation trees*, trees whose nodes are pairs of the form $\Lambda \vdash \Phi$ where Λ is a literal set and Φ a clause set. A derivation for a clause set Φ_0 starts with a single-node derivation tree containing the clause set Φ_0 and grows the tree by applying one of the rules to one of the leaves, adding to that leaf the rule's conclusions as children.

A proof procedure for $\mathcal{ME}$ in effect grows the initial derivation tree in a depth-first manner, backtracking on *closed branches*, i.e., failed branches whose leaf results from an application of Close.[1] The procedure determines that the initial clause set Φ_0 is unsatisfiable after it has determined that all possible branches are closed. Conversely, it finds a model of Φ_0 if it reaches a node that does not contain the empty clause and to which no derivation rule applies.

Like in all backtracking procedures, performance of a proof procedure for $\mathcal{ME}$ can be improved in principle by analyzing the sequence of non-deterministic choices (i.e, Split decisions) that have led to a *conflict*, a closed branch. The analysis determines which of the choices were really relevant to the conflict and saves this information so that the same choices, or *similar* choices that can also lead to a conflict, are avoided later in the search. In the next section, we present two methods for implementing this sort of *learning* process. The methods follow the footprints of popular learning methods from the DPLL literature: conflict analysis is performed by means of a guided resolution derivation that synthesizes a new clause, a *lemma*, containing the reasons for the conflict; then learning is achieved simply by adding the lemma to the clause set and using it like any other clause in constraint propagation steps during the rest of the derivation. These methods can be given a logical justification by seeing them just as another derivation rule that adds to the clause set selected logical consequences of the set. In our experiments we also tried, as a sanity check, a third and much simpler learning method based on purely propositional techniques. While this method has low overhead it is also a lot less general than the other two and did not fare well experimentally. Because of this we do not discuss it here and instead refer the reader to [3] for more details.

Related work. To our knowledge there is little work in the literature on conflict-driven lemma learning in first-order theorem proving. One of them is described in [1] and consists of the "caching" and "lemmaizing" techniques for the model elimination calculus.

[1] More precisely, the proof procedure performs a sort of iterative-deepening search, to avoid getting stuck in infinite branches.

Caching means to store solutions to subgoals (which are single literals) in the proof search. The idea is to look up a solution (a substitution) that solves the current subgoal, based on the solution of a previously computed solution of a compatible subgoal. This idea of replacing search by lookup is thus conceptually related to lemma learning as we consider it here. However, as far as we can tell from [1] (and other publications), the use of lemmas there seems having been restricted to *unit* lemmas, perhaps for pragmatic reasons, although the mechanism has been defined more generally (already in [12]). A more general caching mechanism for unit clauses has been described in [11].

A recent paper [8] describes the Geometric Resolution calculus, which includes a lemma learning mechanism that is closely related to our lifted method (cf. Section 2.2). A major difference is that lemmas learned there are used only to close branches, but not to derive new information such as implied unit clauses. Unfortunately, [8] does not contain an experimental analysis describing the impact of their learning technique.

Further related work comes from Explanation-Based Learning (EBL), which allows the learning of logical descriptions of a concept from the description of a single concept instance and a preexisting knowledge base. A comprehensive and powerful EBL framework based on the language of definite logic programs and SLD-resolution is presented in [15]. As depicted there, EBL is essentially the process of deriving from a given SLD proof a (definite) clause representing parts of the proof or even generalizations thereof. The goal is to derive clauses that are of high *utility*, that is, that help find shorter proofs of similar theorems without broadening the search space too much. The learning procedures we present here follows a similar process. Structurally, they are SLD-derivations producing lemma clauses, and have a role comparable to the derivations of [15].

2 An Abstract Proof Procedure $\mathcal{ME}$

Being a *calculus*, $\mathcal{ME}$ abstracts away many control aspects of a proof search. As a consequence, one cannot formalize in it stateful operational improvements such as learning. Following an approach first introduced in [14] for the DPLL procedure, one can however formalize general classes of proof procedures for $\mathcal{ME}$ in a way that makes it easy to model and analyze operational features like backtracking and learning.

An $\mathcal{ME}$ proof procedure can be described abstractly as a transition system over *states* of the form $\bot$, a distinguished fail state, or the form $\Lambda \vdash \Phi$ where Φ is a clause set and Λ is an *(ordered) context*, that is, a sequence of *annotated literals*, literals with an annotation that marks each of them as a *decision* or a *propagated* literal. We model generic $\mathcal{ME}$ proof procedures by means of a set of states of the kind above together with a binary *transition relation* over these states defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. A proof procedure is then a *transition system*, a set of transition rules defined over some given set of states. In the following, we first introduce a basic transition system for $\mathcal{ME}$ and then extend it with learning capabilities.

Formal Preliminaries. If $\Longrightarrow$ is a transition relation between states we write, as usual, $S \Longrightarrow S'$ instead of $(S,S') \in \Longrightarrow$. We denote by $\Longrightarrow^*$ the reflexive-transitivclosure of $\Longrightarrow$. Given a transition system R, we denote by $\Longrightarrow_\mathrm{R}$ the transition relation defined by R. We call any sequence of transitions of the form $S_0 \Longrightarrow_{\mathrm{R}} S_1,\ S_1 \Longrightarrow_{\mathrm{R}} S_2,\ \ldots$ a *derivation in R*, and denote it by $S_0 \Longrightarrow_{\mathrm{R}} S_1 \Longrightarrow_{\mathrm{R}} S_2 \Longrightarrow \ldots$

The concatenation of two ordered contexts will be denoted by simple juxtaposition. When we want to stress that a context literal L is annotated as a decision literal we will write it as L^{d}. With an ordered context of the form $\Lambda_0 L_1 \Lambda_1 \cdots L_n \Lambda_n$ where $L_1, \ldots L_n$ are all the decision literals of the context, we say that the literals in Λ_0 are at *decision level* 0, and those in $L_i \Lambda_i$ are at decision level i, for all $i = 1, \ldots, n$.

The $\mathcal{ME}$ calculus uses 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 and v to denote elements of V and x, y and z to denote elements of X. 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 $\mathcal{V}ar(t) = \mathcal{P}ar(t) = \emptyset$. A substitution ρ is a *renaming on $W \subseteq (V \cup X)$* iff its restriction to W is a bijection of W onto itself. 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 say that s is *a p-variant of t* iff there is a p-preserving renaming ρ such that $s\rho = t$. We write $s \geq t$ iff there is a p-preserving substitution σ such that $s\sigma = t$. We write t^{sko} to denote the term obtained from t by replacing each variable in t by a fresh Skolem constant. All of the above is extended from terms to literals in the obvious way.

Every (ordered) context the proof procedure works with starts with a *pseudo-literal* of the form $\neg v$ (which, intuitively, stands for all negative ground literals). Where L is a literal and Λ a context, we will write $L \in_{\simeq} \Lambda$ if L is a p-variant of a literal in Λ. 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. (We write $\overline{K}$ to denote the complement of the literal K.) A context Λ is contradictory if one of its literals is contradictory with Λ.

Each non-contradictory context containing $\neg v$ determines a Herbrand interpretation I^{Λ} over the input signature extended by a countable set of Skolem constants. We refer the reader to [5,6] for the formal definition of I^{Λ}. Here it should suffice to say that the difference between (universal) variables and parameters in $\mathcal{ME}$ lies mainly in the definition of this Herbrand interpretation. Roughly, a literal with a parameter, like $A(u)$, in a context assigns true to all of its ground instances that are not also an instance of a more specific literal, like $\neg A(f(u))$, with opposite sign. In contrast, a literal with a variable, like $A(x)$, assigns true to all of its ground instances, with no exceptions.

In a state of the form $\Lambda \vdash \Phi$, the interpretation I^{Λ} is a *candidate* model for Φ. The purpose of the proof procedure is to recognize whether the candidate model is in fact a model of Φ or whether it possibly falsifies a clause of Φ. The latter situation is detectable syntactically through the computation of *context unifiers*.

Definition 1 (Context Unifier). *Let Λ be a context and $C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$ a parameter-free clause, where $0 \leq m \leq n$. A substitution σ is* a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ *iff there are fresh p-variants $K_1, \ldots, K_n \in_{\simeq} \Lambda$ 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 $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$, (iii) for all $i = m+1, \ldots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$. A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is* admissible (for Split) *iff for all distinct $i, j = m+1, \ldots, n$, $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.*

If σ is a context unifier with remainder D of a clause C against a context Λ, we call each literal of D *a remainder literal* of σ. We say that C is *conflicting (in Λ because of σ)* if σ has an empty remainder.

For space constraints we must refer the reader again to [5,6] for the rationale behind context unifiers and how parameters arise in $\mathcal{ME}$ derivations. Intuitively, the existence of a context unifier σ for a clause C indicates that $C\sigma$ is possibly falsified by the current I^Λ. If σ has a remainder literal $L\sigma$, adding $L\sigma$ to the context makes progress towards making I^Λ eventually satisfy $C\sigma$. If σ has no remainder literals, the problem is not repairable and backtracking is instead needed.

2.1 A Basic Proof Procedure for $\mathcal{ME}$

A basic proof procedure for $\mathcal{ME}$ is the transition system B defined by the rules Decide, Propagate, Backjump and Fail below. Since the transition system B is at a lower level of abstraction, its rules do not correspond one-to-one to the derivation rules of $\mathcal{ME}$. Roughly speaking, Decide implements Split, Propagate implements Assert, while Backjump and Fail implement Close. The relevant derivations in this system are those that start with a state of the form $\{\neg v\} \vdash \Phi$, where Φ is the clause set whose unsatisfiability one is interested in.

$$\mathsf{Decide:}\ \Lambda \vdash \Phi, C \vee L \implies \Lambda\,(L\sigma)^{\mathsf{d}} \vdash \Phi, C \vee L \ \text{ if } (*)$$

$$\text{where } (*) = \begin{cases} \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \text{ (cf. Def. 1)} \\ \text{with at least two remainder literals,} \\ L\sigma \text{ is a remainder literal, and} \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

We call the literal $L\sigma$ above a *decision literal* of the context unifier σ and the clause $C \vee L$. Decide makes the non-deterministic decision of adding the literal $L\sigma$ to the context. It is the only rule that adds a literal as a decision literal.

$$\mathsf{Propagate:}\ \Lambda \vdash \Phi, C \vee L \implies \Lambda, L\sigma \vdash \Phi, C \vee L \ \text{ if } (*)$$

$$\text{where } (*) = \begin{cases} \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with a single remainder literal } L\sigma, \\ L\sigma \text{ is not contradictory with } \Lambda, \text{ and} \\ \text{there is no } K \in \Lambda \text{ such that } K \geq L\sigma \end{cases}$$

We call the literal $L\sigma$ in the rule above the *propagated literal* of the context unifier σ and the clause $C \vee L$.

$$\mathsf{Backjump:}\ \Lambda L^{\mathsf{d}} \Lambda' \vdash \Phi, C \implies \Lambda \overline{L}^{\text{sko}} \vdash \Phi, C \ \text{ if } \begin{cases} C \text{ is conflicting in} \\ \Lambda L^{\mathsf{d}} \text{ but not in } \Lambda \end{cases}$$

Backjump models both chronological and non-chronological backtracking by allowing, but not requiring, that the undone decision literal L be the most recent one. Note that L's complement is added as a propagated literal, after all (and only) the variables of L have been Skolemized, which is needed for soundness. More general versions of

Backjump are conceivable, for instance along the lines of the backjump rule of Abstract DPLL [14]. Again, we present this one here mostly for simplicity.

$$\textsf{Fail:}\ \Lambda \vdash \Phi, C \implies \bot \ \text{ if } \begin{cases} C \text{ is conflicting in } \Lambda, \\ \Lambda \text{ contains no decision literals} \end{cases}$$

Fail ends a derivation once all possible decisions have generated a conflict.

$$\textsf{Restart:}\ \Lambda \vdash \Phi \implies \{\neg v\} \vdash \Phi$$

Restart is used to generate fair derivations that explore the search space in an iterative-deepening fashion.

Although it is beyond the scope of this paper, one can show that there are (deterministic) rule application strategies for this transition system that are refutationally sound and complete, that is, that reduce a state of the form $\{\neg v\} \vdash \Phi$ to the state $\bot$ if and only if Φ is unsatisfiable.

2.2 Adding Learning to $\mathcal{ME}$ Proof Procedures

To illustrate the potential usefulness of learning techniques for a transition system like the system B defined in the previous subsection, it is useful to look first at an example of a derivation in B.

Example 1. Let Φ be a clause set containing, among others, the clauses:

$$(1)\ \neg B(x) \vee C(x,y) \quad (2)\ \neg A(x) \vee \neg C(y,x) \vee D(y) \quad (3)\ \neg C(x,y) \vee E(x) \quad (4)\ \neg D(x) \vee \neg E(x).$$

The table below provides a trace of a possible derivation of Φ. The first column shows the literal added to the context by the current derivation step, the second column specifies the rule used in that step, and the third indicates which instance of a clause in Φ was used by the rule. A row with ellipses stands for zero or more intermediate steps. Note that Backjump *replaces* the whole subsequence $B(u)^{\mathsf{d}}\, C(u,y)\, D(u)\, E(u)$ of the current context with $\neg B(u)$.

Context Literal	**Derivation Rule**	**Clause Instance**
...	...	...
$A(t(x))$	Propagate	instance $A(t(x)) \vee \cdots$ of some clause in Φ where $t(x)$ is a term with a single variable x.
...	...	...
$B(u)^{\mathsf{d}}$	Decide	instance $B(u) \vee \cdots$ of some clause in Φ
$C(u,y)$	Propagate	instance $\neg B(u) \vee C(u,y)$ of (1)
$D(u)$	Propagate	instance $\neg A(t(x)) \vee \neg C(u,t(x)) \vee D(u)$ of (2)
$E(u)$	Propagate	instance $\neg C(u,y) \vee E(u)$ of (3)
$\neg B(u)$	Backjump	instance $\neg D(u) \vee \neg E(u)$ of (4)

It is clear by inspection of the trace that any intermediate decisions made between the additions of $A(t(x))$ and $B(u)$ are irrelevant in making clause (4) conflicting at the point of the Backjump application. The fact that (4) is conflicting depends only on the decisions that lead to the propagation of $A(t(x))$—say, some decision literals $S_1,\ldots,S_n$ with $n \geq 0$—and the decision to add $B(u)$. This means that the decision literals $S_1,\ldots,S_n,B(u)$ will eventually produce a conflict (i.e., make some clause conflicting) in any context that contains them. The basic goal of this work is to define efficient conflict analysis procedures that can come to this conclusion automatically and store it into the system in such a way that Backjump is applicable, possibly with few propagation steps, whenever the current context happens to contain again the literals $S_1,\ldots,S_n,B(u)$. Even better would be the possibility to avoid altogether the addition of $B(u)$ as a decision literal in any context containing $S_1,\ldots,S_n$, and instead add the literal $\neg B(u)$ as a propagated literal. We discuss how to do these in the rest of the paper. □

Within the abstract framework of Section 2.1, and in perfect analogy to the Abstract DPPL framework of Nieuwenhuis *et al.* [14], learning can be modeled very simply and generally by the addition of the following two rules to the transition system B:

Learn: $\Lambda \vdash \Phi \implies \Lambda \vdash \Phi, C$ if $\Phi \models C$

Forget: $\Lambda \vdash \Phi, C \implies \Lambda \vdash \Phi$ if $\Phi \models C$

Note that adding entailed clauses to the clause set is superfluous for completeness. The Learn rule then is meant to be used only to add clauses that are more likely to cause further propagations and correspondingly reduce the number of needed decisions. The intended use of the Forget rule is to control the growth of the clause set, by removing entailed clauses that cause little propagation.

Because of the potentially high overhead involved in generating lemmas and propagating them in practice, we focus in this work on only the kind of *conflict-driven* learning that has proven to be very effective in DPLL-based solvers. In the following we discuss two methods for doing that. Both of them are directly based on a lemma generation technique common in DPLL implementations. This technique can be described proof-theoretically as a linear resolution derivation whose initial central clause is a conflicting clause in the DPLL computation, and whose side clauses are clauses used in unit propagation steps. In terms of the abstract framework above, the linear resolution derivation proceeds as follows. The central clause $C \vee \overline{L}$ is resolved with a clause $L \vee D$ in the clause set only if L was added to the current context by a Propagate step with clause $L \vee D$. Since the net effect of each resolution step is to replace $\overline{L}$ in $C \vee \overline{L}$ by L's "causes" D, we can also see this resolution derivation as a *regression* process.

Both of the methods we present below lift this regression to the first-order case, although with different degrees of generality. The first method produces lemmas that are strictly subsumed by the lemmas produced by the second method. We present it here because it is practically interesting in its own right, and because it can be used to greatly simplify the presentation of the second method.

The Grounded Method. Let $\mathbf{D} = (\{\neg v\} \vdash \Phi_0 \Longrightarrow_{\mathrm{L}} \ldots \Longrightarrow_{\mathrm{L}} \Lambda \vdash \Phi)$ be a derivation in the transition system L where Λ contains at least one decision literal and Φ contains a clause C_0 conflicting in Λ. We describe a process for generating from $\mathbf{D}$ a *lemma*, a clause logically entailed by Φ, which can be *learned* in the derivation by an application of Learn to the state $\Lambda \vdash \Phi$.

We describe the lemma generation process itself as a transition system, this time applied to *annotated clauses*, pairs of the form $C \mid S$ where C is a clause and S is finite mapping $\{L \mapsto M, \ldots\}$ from literals in C to context literals of $\mathbf{D}$. A transition invariant for $C \mid S$ will be that C consists of negated ground instances of context literals, while S specifies for each literal L of C the context literal M of which $\overline{L}$ is an instance, provided that M is a propagated literal. The mapping $L \mapsto M$ will be used to *regress* L, that is, to resolve it with M in the clause used in $\mathbf{D}$ to add M to the context.

The initial annotated clause A_0 will be built from the conflicting clause of $\mathbf{D}$ and will be regressed by applying to it the GRegress rule, defined below, one or more times. In the definition of A_0 and of GRegress we use the following notational conventions: if σ is a substitution and C a clause or a literal, $C\underline{\sigma}$ denotes the expression obtained by replacing each variable or parameter of $C\sigma$ by a fresh Skolem constant (one per variable or parameter); if σ is a context unifier of a clause $L_1 \vee \cdots \vee L_n$ against some context, we denote by L_i^σ the context literal paired with L_i by σ.

Assume that C_0 is conflicting in Λ because of some context unifier σ_0. Then A_0 is defined as the annotated lemma

$$A_0 = C_0\underline{\sigma_0} \mid \{L\underline{\sigma_0} \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\}$$

consisting of a fresh grounding of $C_0\sigma_0$ by Skolem constants (hence the name "grounded method") and a mapping of each literal of $C_0\underline{\sigma_0}$ to its pairable literal in Λ if that literal is a propagated literal. The regression rule is

$$\text{GRegress:}\ \ D \vee M \mid S, M \mapsto L\sigma \ \Longrightarrow_{\mathsf{gr}}\ D \vee C\sigma\underline{\mu} \mid S, T \ \ \text{if } (*)$$

$$\text{where } (*) = \begin{cases} L\sigma \text{ is the propagated literal of some context unifier } \sigma \text{ and clause } L \vee C, \\ \mu \text{ is a most general unifier of } M \text{ and } \overline{L\sigma}, \\ T = \{N\sigma\underline{\mu} \mapsto N^\sigma \mid N \in C \text{ and } N^\sigma \text{ is a propagated literal}\} \end{cases}$$

Note that the mapping is used by GRegress to guide the regression so that no search is needed. The regression process simply repeatedly applies the rule GRegress an arbitrary number of times starting from A_0 and returns the last clause. While this clause is ground by construction, it can be generalized to a non-ground clause C by replacing each of its Skolem constants by a distinct variable. As stated in the next result, this generalized clause is a logical consequence of the current clause set Φ in the derivation, and so can be learned with an application of the Learn rule.[2]

Proposition 1. *If* $A_0 \Longrightarrow_{\mathsf{gr}}^* C' \mid S$*, the clause* C *obtained from* C' *by replacing each constant of* C' *not in* Φ *by a fresh variable is a consequence of* Φ_0.

An important invariant in practice is that one can continue regressing the initial clause until it contains only decision literals. This result, expressed in the next proposition, gives one great latitude in terms of how far to push the regression. In our implementation, to reduce the regression overhead, and following a common practice in DPLL solvers, we regress only propagated literals belonging to the last decision level of Λ.

Proposition 2. *If* $A_0 \Longrightarrow_{\mathsf{gr}}^* A$ *and* A *has the form* $D \vee M \mid S, M \mapsto N$*, then the* GRegress *rule applies to* A.

Example 2. Figure 1 shows a possible regression of the conflicting clause $\neg D(x) \vee \neg E(x)$ in the derivation of Example 1. This clause is conflicting because of the con-

[2] We refer the reader to a longer version of this paper [3] for all the proofs of the results below.

$$\cfrac{\cfrac{\cfrac{\cfrac{\neg D(a)\lor\neg\mathbf{E}(\mathbf{a}) \qquad \neg C(u,y)\lor E(u)}{\neg\mathbf{D}(\mathbf{a})\lor\neg C(a,b) \qquad \neg A(t(x))\lor\neg C(u,t(x))\lor D(u)}}{\neg\mathbf{C}(\mathbf{a},\mathbf{b})\lor\neg A(t(c))\lor\neg C(a,t(c)) \qquad \neg B(u)\lor C(u,y)}}{\neg A(t(c))\lor\neg\mathbf{C}(\mathbf{a},\mathbf{t}(\mathbf{c}))\lor\neg B(a) \qquad \neg B(u)\lor C(u,y)}}{\neg A(t(c))\lor\neg B(a)}$$

Fig. 1. Grounded regression of $\neg D(u)\lor\neg E(u)$

text unifier $\sigma_0 = \{x \mapsto u\}$, pairing the clause literals $\neg D(x)$ and $\neg E(x)$ respectively with the context literals $D(u)$ and $E(u)$. So we start with the initial annotated clause: $A_0 = (\neg D(x)\lor\neg E(x))\underline{\sigma_0} \mid \{(\neg D(x))\underline{\sigma_0} \mapsto (\neg D(x))^{\sigma_0},\ (\neg E(x))\underline{\sigma_0} \mapsto (\neg E(x))^{\sigma_0}\} = \neg D(a)\lor\neg E(a) \mid \{\neg D(a) \mapsto D(u),\ \neg E(a) \mapsto E(u)\}$. To ease the notation burden, we represent the regression in the more readable form of a linear resolution tree, where at each step the central clause is the regressed clause, the literal in bold font is the regressed literal, and the side clause is the clause $(L\lor C)\sigma$ identified in the precondition of GRegress. The introduced Skolem constants are a, b and c. Stopping the regression with the resolvent $\neg A(t(c))\lor\neg B(a)$ gives, after replacing the Skolem constants by fresh variables, the lemma $\neg A(t(z_c))\lor\neg B(z_a)$. (Similarly for the previous resolvents.) □

To judge the effectiveness of lemmas learned with this process in reducing the explored search space we also need to argue that they let the system later recognize more quickly, or possibly avoid altogether, the set of decisions responsible for the conflict in **D**. This is not obvious within the $\mathcal{ME}$ calculus because of the role played by parameters in the definition of a conflicting clause. (Recall that a clause is conflicting because of some context unifier σ iff it moves parameters only to parameters in the context literals associated with the clause.) To show that lemmas can have the intended consequences, we start by observing that, by construction, every literal L_i in a lemma $C = L_1\lor\cdots\lor L_m$ generated with the process above is a negated instance of some context literal K_i in Λ. Let us write C^Λ to denote the set $\{K_1,\ldots,K_m\}$.

Proposition 3. *Any lemma C produced from* **D** *by the regression method in this section is conflicting in any context that contains C^Λ.*

Proposition 3 implies, as we wanted, that having had the lemma C in the clause set from the beginning could have led to the discovery of a conflict sooner, that is, with less propagation work and possibly also less decisions than in **D**. Moreover, the more regressed the lemma, the sooner the conflict would have been discovered. For instance, looking back at the lemmas generated in Example 2, it is easy to see that the lemma $\neg C(z_a,z_b)\lor\neg A(t(z_c))\lor\neg C(z_a,t(z_c))$ becomes conflicting in the derivation of Example 1 as soon as $C(u,y)$ is added to the context. In contrast, the more regressed lemma $\neg A(t(z_c))\lor\neg B(z_a)$ becomes conflicting as soon as the decision $B(u)$ is made. Since a lemma generated from **D** is typically conflicting once a *subset* of the decisions in Λ are taken, learning it in the state $\Lambda \vdash \Phi, C_0$ will help recognize more quickly these wrong decisions later in extensions of **D** that undo parts of Λ by backjumping. In fact, if the lemma is regressed enough, one can do even better and completely avoid the conflict later on if one uses a derivation strategy that prefers applications of Propagate to applications of Decide.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg D(x) \lor \neg \mathbf{E}(\mathbf{x}) \qquad \neg C(x_1,y_1) \lor E(x_1)}{\neg \mathbf{D}(\mathbf{x}) \lor \neg C(x,y_1)} \qquad \neg A(x_2) \lor \neg C(y_2,x_2) \lor D(y_2)}{\neg \mathbf{C}(\mathbf{x},\mathbf{y_1}) \lor \neg A(x_2) \lor \neg C(x,x_2)} \qquad \neg B(x_3) \lor C(x_3,y_3)}{\neg A(x_2) \lor \neg \mathbf{C}(\mathbf{x},\mathbf{x_2}) \lor \neg B(x)} \qquad \neg B(x_4) \lor C(x_4,y_4)}{\neg A(x_2) \lor \neg B(x)}$$

Fig. 2. Lifted regression of $\neg D(x) \lor \neg E(x)$

Example 3. Consider an extension of the derivation in Example 1 where the context has been undone enough that now its last literal is $A(t(x))$. By applying Propagate to the lemma $\neg A(t(z_c)) \lor \neg B(z_a)$ it is possible to add $\neg B(z_a)$ to the context, thus preventing the addition of $B(u)$ as a decision literal (because $B(u)$ is contradictory with $\neg B(z_a)$) and avoiding the conflict with clause (4). With the less regressed lemma $\neg C(z_a,z_b) \lor \neg A(t(z_c)) \lor \neg C(z_a,t(z_c))$ it is still possible to add $\neg B(z_a)$, but with two applications of Propagate—to the lemma and then to clause (1). □

So far, what we have described mirrors what happens with propositional clause sets in DPLL SAT solvers. What is remarkable about learning at the $\mathcal{ME}$ level, besides producing the same nice effects obtained in DPLL, is that its lemmas are not just caching compactly the reasons for a specific conflict. For being a *first-order* formula, a lemma in $\mathcal{ME}$ represents an *infinite* class of conflicts of the same form. For instance, the lemma $\neg A(t(z_c)) \lor \neg B(z_a)$ in our running example will become conflicting once the context contains *any* instance of $A(t(z_c))$ and $B(z_a)$, not just the original $A(t(x))$ and $B(u)$.

Our lemma generation process then does learning in a more proper sense of the word, as it can generalize over a single instance of a conflict, and later recognize *unseen* instances in the same class, and so lead to additional pruning of the search space.

A slightly more careful look at the derivation in Example 1 shows that the lemma $\neg A(t(z_c)) \lor \neg B(z_a)$ is actually not as general as it could be. The reason is that a conflict arises also in contexts that contain, in addition to any instance of $B(z_a)$, also any *generalization* of $A(t(z_c))$. So a better possible lemma is $\neg A(z) \lor \neg B(z_a)$. We can produce generalized lemmas like the above by lifting the regression process similarly as in Explanation-Based Learning (cf. Section 1). We describe this lifted process next.

The Lifted Method. Consider again the derivation **D** from the previous subsection, whose last state $\Lambda \vdash \Phi$ contains a clause C_0 that is conflicting in Λ because of some context unifier σ_0. Using basic results about resolution and unification, this derivation can be lifted to the first-order level. The lifted derivation can be built simply by following the steps of the grounded derivation, but this time using the original clauses in Φ for the initial central clause and the side clauses. In practice of course, the lifted derivation can be built directly, without building the grounded derivation first. As in the grounded case, we can use any regressed clause C as a lemma but with the difference that we do not need to abstract away Skolem constants because the regression process resolves only input clauses of C. Again, the resulting clause is a logical consequence of Φ.

More details, including a soundness proof can be found in the long version [3]. Here we will only present the main idea by means of an example.

Example 4. Figure 2 shows the lifting of the grounded regression in Figure 1 for the conflicting clause $\neg D(x) \vee \neg E(x)$ in the derivation of Example 1. This time, we start with the initial annotated clause: $(\neg D(x) \vee \neg E(x)) \mid \{\neg D(x) \mapsto D(u),\ \neg E(x) \mapsto E(u)\}$. As before, we represent the regression as a linear resolution tree, where this time at each step the central clause is the regressed clause, the literal in bold font is the regressed literal, and the side clause corresponds to the clause $L \vee C$ in the precondition of the lifted version of GRegress. The lemma learned in this case is $\neg A(x_2) \vee \neg B(x)$. □

3 Experimental Evaluation

A detailed discussion on implementing the various methods can be found in [3], where we describe the regression processes more concretely. We also discuss some memoization techniques used to reduce the regression cost, condensing techniques to limit the size of lemmas, and a simple lemma forgetting policy. Here we focus on our initial experimental results.

First problem set. We first evaluated the effectiveness of lemma learning with version 1.2 of *Darwin* over version 3.1.1 of the TPTP problem library. Since *Darwin* can handle only clause logic, and has no dedicated inference rules for equality, we considered only clausal problems without equality, both satisfiable and unsatisfiable ones. Furthermore, as *Darwin* never applies the Decide rule in Horn problems [10], and thus also never backtracks, we further restricted the selection to non-Horn problems only. All tests were run on Xeon 2.4Ghz machines with 1GB of RAM. The imposed limit on the prover were 300s of CPU time and 512MB of RAM.

The first 4 rows of Table 1 summarize the results for various configurations of *Darwin*, namely, not using lemmas and using lemmas with the grounded and lifted regression methods. The first significant observation is that all configurations solve almost exactly the same number of problems, which is somewhat disappointing. The situation is similar even with an increased timeout of one hour per problem. A sampling of the derivation traces of the unsolved problems, however, reveals that they contain only a handful of Backjump steps, suggesting that the system spends most of the time in propagation steps and supporting operations such as the computation of context unifiers.

The second observation is that for the solved problems the search space, measured in the number of Decide applications, is significantly pruned by all learning methods (with 18% to 58% less decisions), although this improvement is only marginally reflected in the run times. This too seems to be due to the fact that most derivations involve only a few applications of Backjump in the configuration without lemmas. Indeed, 652 of the 898 problems solved with the lifted technique require at most 2 backjumps. This implies that only a few lemmas can be learned, and thus their effect is limited and the run time of most problems remains unchanged. Based on these tests alone, it is not clear if the small number of backjumps is an artifact of the specific proof procedure implemented by *Darwin* or a characteristic of the problems in the TPTP library.

The rest of Table 1 shows the same statistics, but restricted to the problems solved by the no lemmas configuration using, respectively, at least 3, 20, and 100 applications of Backjump within the 300s time limit. There, the effect of the search space pruning is more pronounced and does translate into reduced run times. In particular, the speed

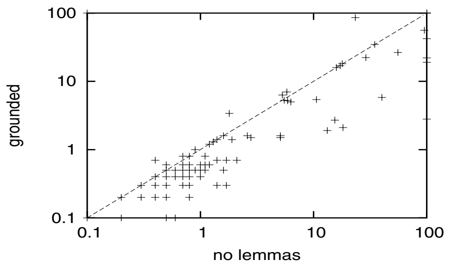
Table 1. Problems that respectively take at least 0, 3, 20, and 100 applications of Backjump without lemmas within 300s, where **Solved Problems** gives the number of problems solved by a configuration, while the remaining values are for the subsets of 894, 241, 106, 65 problems solved by *all* configurations. **Avg Time** (**Total Time**) gives the average (total) time needed for the problems solved by all configurations, **Speed up** shows the run time speed up factor of each configuration versus the one with no lemmas. **Failure**, **Propagate**, and **Decide** give the number of rule applications, with **Failure** including both Backjump and Fail applications.

Method	**Solved Probls**	**Avg Time**	**Total Time**	**Speed up**	**Failure Steps**	**Propag. Steps**	**Decide Steps**
no lemmas	896	2.7	2397.0	1.00	24991	597286	45074
grounded	895	2.4	2135.6	1.12	9476	391189	18935
lifted	898	2.4	2173.4	1.10	9796	399525	19367
no lemmas	244	3.0	713.9	1.00	24481	480046	40766
grounded	243	1.8	445.1	1.60	8966	273849	14627
lifted	246	2.0	493.7	1.45	9286	282600	15059
no lemmas	108	5.2	555.7	1.00	23553	435219	38079
grounded	108	2.2	228.5	2.43	8231	228437	12279
lifted	111	2.6	274.4	2.02	8535	238103	12688
no lemmas	66	5.0	323.9	1.00	21555	371145	34288
grounded	67	1.7	111.4	2.91	6973	183292	9879
lifted	70	2.3	151.4	2.14	7275	193097	10294

up of each lemma configuration with respect to the no lemmas one steadily increases with the difficulty of the problems, reaching a factor of almost 3 for the most difficult problems in the grounded case. Moreover, the lifted lemmas configuration always solves a few more problems than the no lemmas one.

Because of the way *Darwin*'s proof procedure is designed [3], in addition to pruning search space, lemmas may also cause changes to the order in which the search space is explored. Since experimental results for unsatisfiable problems are usually more stable with respect to different space exploration orders, it is instructive to separate the data in Table 1 between satisfiable and unsatisfiable problems. For lack of space, we must refer the reader to [3] for detailed tables with this breakdown. Here it suffices to say that the results for unsatisfiable problems show the same pattern as the aggregate results in Table 1. Those solved by all configurations and solved by the no lemmas one with at least 0, 3, 20, and 100 backjumps are respectively 561, 191, 89, and 61. For these unsatisfiable problems, the speed up factors for grounded lemmas in particular are respectively 1.07, 1.55, 3.74, and 4.19, which actually compares more favorably overall to the corresponding speed up factors in Table 1: resp., 1.12, 1.60, 2.43, and 2.91.

In Figure 3 we plot the individual run times of the no lemmas configuration against the lemma configurations for all problems solved by at least one configuration and generating at least 3 backjumps in the no lemma one. The scatter plots clearly show the positive effect of learning. For nearly all of the problems, the performance of the



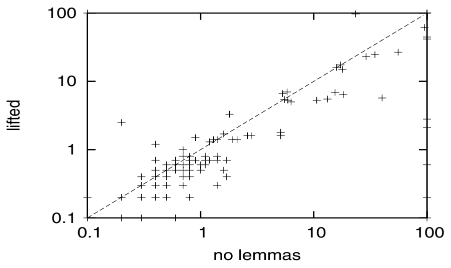


Fig. 3. Comparative performance, on a log-log scale, for different configurations for problems with at least 3 applications of Backjump. For readability, the cutoff is set at 100s instead of 300s, because in all cases less than a handful of problems are solved in the 100-300s range.

grounded lemmas configuration is better, often by a large margin, than the one with no lemmas. A similar situation occurs with lifted lemmas, although there are more problems for which the no lemmas configuration is faster.

Overall, our results indicate that lifted lemmas generate more Decide applications and have higher overhead than grounded lemmas. The larger number of decision steps of the lifted method versus the grounded one seems paradoxical at first sight, but can be explained by observing that lifted lemmas—in addition to avoiding or detecting early a larger number of conflicts—also cause the addition of more general propagated literals to a context, leading to a higher number of (possibly useless) context unifiers. Furthermore, due to the increased generality of lifted lemmas and the way they are condensed when they are too long, sometimes Propagate applies to a grounded lemma but not the corresponding lifted lemma, making the latter *less* effective at avoiding conflicts (see [3] for more details).

The higher overhead of the lifted method can be attributed to two main reasons. The first is of course the increased number of context unifiers to be considered for rule applications. The second is the intrinsically higher cost of the lifted method versus the grounded one, because of its use of unification—as opposed to matching—operations during regression, and its considerable post-processing work in removing multiple variants of the same literals from a lemma—something that occurs quite often.

Second problem set. Given that only a minority of the TPTP problems we could use in the first experiment cause a considerable amount of search and backtracking, and that, on the other hand, many decidable fragments of first-order logic are NP-hard, we considered a second problem set, stemming from an application of *Darwin* for finite model finding [4]. This application follows an approach similar to that of systems like Paradox [7]. To find a finite model of a given cardinality n, a clause set, with or without equality, is converted into an equisatisfiable Bernays-Schönfinkel problem (instead of a propositional problem as in Paradox) that includes the cardinality restriction. If *Darwin* proves the latter clause set unsatisfiable, it increases the value of n by 1 and restarts, repeating the process until it finds a model—and diverging if the original problem has no finite models. Since *Darwin* is a decision procedure for the Bernays-Schönfinkel class, starting with n above at 1, it is guaranteed to find a finite model of minimum size if one exists. In the configurations with learning, *Darwin* uses lemmas during each

Table 2. Satisfiable problems that transformed to a finite model representation respectively take at least 0, 100, and 1000 applications of Backjump without lemmas within 300s, where **Solved Problems** gives the number of problems solved by a configuration, while the remaining values are for the subsets of 647, 152, 47 problems solved by *all* configurations.

Method	**Solved Probls**	**Average Time**	**Total Time**	**Speed up**	**Failure Steps**	**Propagate Steps**	**Decide Steps**
no lemmas	657	5.6	3601.3	1.00	404237	16122392	628731
grounded	669	3.3	2106.3	1.71	74559	4014058	99865
lifted	657	4.7	3043.9	1.18	41579	1175468	68235
no lemmas	162	17.8	2708.6	1.00	398865	15911006	614572
grounded	174	7.9	1203.1	2.25	70525	3833986	87834
lifted	162	14.0	2126.2	1.27	38157	1023589	57070
no lemmas	52	36.2	1702.9	1.00	357663	14580056	555015
grounded	64	10.5	495.3	3.44	53486	3100339	64845
lifted	57	11.5	538.7	3.16	26154	678319	39873

iteration of the process and carries over to the next iteration those lemmas not depending on the cardinality restriction. Since a run over a problem with a model of minimum size n includes $n-1$ iterations over unsatisfiable clause sets, it is reasonable to consider together all the n iterations in the run when measuring the effect of learning.

Table 2 shows our results for (the BS translation of) all the 815 satisfiable problems of the TPTP library.[3] In general, solving a problem in *Darwin* with the process above requires significantly more applications of Backjump than for the set of experiments presented earlier. As a consequence, the grounded lemmas configuration performs significantly better than the no lemmas configuration, solving the same problems in about half the time, and also solving 12 new problems. The lifted configuration on the other hand performs only moderately better. Although the search space is drastically reduced (the number of decisions is reduced by an order of magnitude in all cases), the overhead of lemma simplification almost outweighs the positive effects of pruning. Restricting the analysis to harder problems shows that the speed up factor of grounded lemmas increases gradually to about 3.5.

This second set of results then confirms that learning has a significant positive effect in solving problems that require a lot of search and produce comparatively few unit propagations.

4 Conclusion and Further Work

We have presented two methods for implementing conflict-based learning in proof procedures for the Model Evolution calculus. The methods have different degrees of

[3] For an idea how we compare with other systems, Mace 4 [13] and Paradox 1.3, currently the fastest finite model finders available, respectively solve 553 and 714 of those problems, making *Darwin* second only to Paradox.

generality, implementation difficulty, and effectiveness in practice. Our initial experimental results indicate that for problems that are not trivially solvable by the *Darwin* implementation and do not cause too much constraint propagation all methods have a dramatic pruning effect on the search space. The grounded method, however, is the most effective at reducing the run time as well.

We plan to investigate the grounded and the lifted methods further, possibly adapting to our setting some of the heuristics developed in [15], in order to make learning more effective and reduce its computational overhead. We also plan to evaluate experimentally our learning methods with sets of problems not (yet) in the TPTP library.

References

1. O. L. Astrachan and M. E. Stickel. Caching and Lemmaizing in Model Elimination Theorem Provers. In D. Kapur, ed., *Proc. CADE-11*, LNAI 607. Springer, 1992.
2. P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.
3. P. Baumgartner, A. Fuchs, and C. Tinelli. Lemma Learning in the Model Evolution Calculus. Technical Report no. 06-04, Department of Computer Science, The University of Iowa, 2006. (Available at `http://www.cs.uiowa.edu/~tinelli/papers.html`.)
4. P. Baumgartner, A. Fuchs, C. Tinelli and H. de Nivelle, and Cesare Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. In W. Ahrendt, P. Baumgartner and H. de Nivelle, eds., *IJCAR'06 Workshop on Disproving*, 2006.
5. P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In Franz Baader, ed., *Proc. CADE-19*, LNAI 2741. Springer, 2003.
6. P. Baumgartner and C. Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Germany, 2003.
7. K. Claessen and N. Sörensson. New Techniques that Improve MACE-Style Finite Model Building. In P. Baumgartner and C. G. Fermüller, eds., *CADE-19 Workshop on Model Computation*, 2003.
8. H. de Nivelle and J. Meng. Geometric resolution: A Proof Procedure Based on Finite Model Search. In U. Furbach and N. Shankar, eds., *Proc. IJCAR*, *LNAI 4130*. Springer, 2006.
9. G. DeJong and R. J. Mooney. Explanation-Based Learning: An Alternative View. *Machine Learning*, 1(2):145–176, 1986.
10. A. Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau, 2004.
11. R. Letz and G. Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, Elsevier, 2001.
12. D. Loveland. *Automated Theorem Proving - A Logical Basis.* North Holland, 1978.
13. William McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003.
14. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In F. Baader and A. Voronkov, eds., *Proc. LPAR'04*, LNCS 3452, Springer, 2005.
15. A. Segre and C. Elkan. A High-Performance Explanation-Based Learning Algorithm. *Artificial Intelligence*, 69:1–50, 1994.