Lemma Learning in the Model Evolution Calculus

Peter Baumgartner
National ICT Australia (NICTA)
Peter.Baumgartner@nicta.com.au

Cesare Tinelli
Department of Computer Science
The University of Iowa
tinelli@cs.uiowa.edu

Alexander Fuchs
Department of Computer Science
The University of Iowa
fuchs@cs.uiowa.edu

March 14, 2006
(Last modified May 19, 2006)

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 three 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 within Darwin of the three learning procedures.

1 Introduction

The Model Evolution ($\mathcal{ME}$) Calculus [BT03a] is a proper lifting to first-order logic of the DPLL procedure, a backtracking search procedure for propositional satisfiability. Similarly to 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 [BFT06b]. 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 and prove correct three lemma generation procedures for $\mathcal{ME}$ with various 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 discuss some of these issues and then present initial experimental results on the performance of an implementation within *Darwin* of the learning procedures.

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 the non-optional rules only. All the learning methods presented in this paper can be extended with minor modifications to $\mathcal{ME}$ derivations that use the optional rules as well. The derivation rules are presented in [BT03a] and in more detail in [BT03b]. 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 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

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

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.

In terms of constraint solving and search, the Split rule is the only rule that creates choice points in the search space. The Assert and the Close rules are just constraint propagation operations that limit the amount of explored search space. 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*, i.e., 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 three methods for implementing this sort of *learning* process. The first two 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 contrast, the third method we describe, although similar in spirit to the first two, has only an operational justification.

1.1 Related Work

The paper [GP00] is about satisfiability checking of function-free first-order formulas. Instead of the widely used reduction to propositional satisfiability and using a SAT solver then, the authors propose to keep some of the original formulas (clauses) and extend a SAT solver to reason with them natively, on the first-order level, instead of grounding them right away. Their main result is that this idea may well pay off when problem instances become larger.

There is potential to connect their approach to ours by observing that these first-order formulas obviously need not come from the input formula—they could be lemma clauses learnt by the techniques we propose. From the perspective of the Model Evolution calculus, the mentioned result allows to speculate that even for ground derivations, the learning of non-ground lemma clauses, which is supported by our techniques, may pay off when properly implemented.

Regarding directly related work, to our knowledge there is little work in the literature on conflict-driven lemma learning in first-order theorem proving. The only approaches we are aware of have been formulated for the model elimination calculus. One of them is described in [AS92] and consists of the “caching” and “lemmaizing” techniques. 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 sugoal, 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, caching corresponds to learning of *unit* clauses, and it works only for Horn clause sets.[2] Closer to our approach is the lemmaizing technique, which allows to generate and use new clauses as the derivation proceeds. Lemma clauses are generated on branch closure and by taking the so-called A-literals of a branch into consideration. The basic motivation for doing so is, like in our approach, to represent a sub-proof by a single clause. Unsurprisingly, the lemmas are consequences of the input clauses, potentially there are a lot of them, and their use may and should be subject to (arbitrary) heuristics. As far as we can tell from [AS92] (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 [Lov78]). The other related approach is described in [LS01]. Their approach generalizes the caching technique of [AS92]. By caching the solutions in a more context-dependent ("local") way, it works for non-Horn clauses, too. In particular the variant of *failure caching*, which can be used to conclude that a subgoal does not have a solution, turned out to be very useful in practice.

Another source for related work comes from Explanation-Based Learning (EBL), a set of deductive learning techniques used in Artificial Intelligence. Generally speaking, EBL 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 is presented in [SE94]. In theorem proving terms, its basis consists essentially of the language of definite logic programs and the calculus of SLD-resolution. 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 rationale is to derive clauses that are of high *utility*, that is, help to find shorter proofs of similar theorems without broadening the search space too much. Deriving such clauses can be explained using resolution terminology ([SE94] use their own language). It basically amounts to *partially* replaying parts of the given SLD proof and further modifing the obtained clause in a sound way. For instance, one of the heuristics prescribes to exhaustively apply resolution steps with binary clauses (implications with one body literal). Another heuristics amounts to factoring, while still another one is based on removing redundant subgoals. Apart from the specific heuristics, the learning procedures we present here follow a similar process. Structurally, they are SLD-derivations producing lemma clauses, and have a role comparable to the derivations of [SE94].

[2]The underlying reason is that for non-Horn input clause sets in general the so-called reduction inference rule is needed, which may introduce dependencies between sugoals that would have to be taken care of in the lemma generation process.

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, which is based on generating new clauses and adding them to the current clause set for use in later branches of the search tree. Following an approach first introduced in [NOT05] 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 relation $\Longrightarrow$ over these states, the *transition relation*, 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.

2.1 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-transitive closure 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 $S_0 \Longrightarrow_{\mathrm{R}} S_1 \Longrightarrow_{\mathrm{R}} S_2 \Longrightarrow \ldots$ When convenient, we will see an ordered context simply as a set of literals—ignoring both the annotations and multiple occurences of its elements. 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 formal framework described so far would be sufficiently specified to define proof procedures for propositional logic. However, for the first-order proof procedures we need additional technical preliminaries. They involve working with non-ground contexts and non-ground input clause sets.

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 and

y 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, and write $s \simeq t$, iff there is a p-preserving renaming ρ such that $s\rho = t$. 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 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 will start with a *pseudo-literal* of the form $\neg v$, the role of which will become clear later. In examples will usually not write $\neg v$ explicitly. 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. A context Λ is contradictory if one of its literals is contradictory with Λ. Each non-contradictory context starting with the pseudo-literal $\neg v$ defines a Herbrand interpretation I^{Λ}.[3] In a state of the form $\Lambda \vdash \Phi$, the interpretation I^{Λ} is meant to be 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 2.1 (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

1. σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
2. for all $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,
3. 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 $i, j = m+1, \ldots, n$, $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$. □

Note that each context unifier has a unique remainder. 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.

[3] 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. See [BT03a, BT03b] for details.

2.2 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. 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.

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

$$\text{where } (*) = \begin{cases} \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \text{ (cf. Def. 2.1)} \\ \text{with at least two remainder literals,} \\ L\sigma \text{ is a remainder literal, and} \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\mathrm{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

We call the literal $L\sigma$ above a *decision literal* of the context unifier σ and the clause $C \vee L$. This rule corresponds to an application of the left-hand side of the Split rule in $\mathcal{ME}$ (with the additional restriction that the context unifier must have at least two remainder literals). 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.

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

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

We call the literal $L\sigma$ in the rule above the *propagated literal* of the context unifier σ and the clause $C \vee L$. This rule corresponds to applying the left-hand side of Split in $\mathcal{ME}$ with a context unifier with a unit remainder, and ignoring the right-hand side.[4] Propagate also models the effect of applying the Assert rule of $\mathcal{ME}$ when L is a positive literal. For simplicity, we ignore here the case in which L is negative. That case somewhat complicates the Propagate's definition and is needed neither for the proof procedure's completeness nor for describing the results of this work.

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

This rule corresponds to the application of the Close rule in $\mathcal{ME}$, followed by a right-hand side Split application higher up in the closed branch. 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

[4] Ignoring the right-hand side in this case is justified in $\mathcal{ME}$ derivations because it produces a sequent to which the Close rule is immediately applicable.

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 [NOT05]. Again, we present this one here mostly for simplicity.

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

This rule corresponds to the application of the Close rule in $\mathcal{ME}$ to the last unexplored branch of the derivation tree, with all other branches being already closed.

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

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. Furthermore, for all (finite) derivations ending with an irreducible state of form $\Lambda \vdash \Phi$, Λ determines a Herbrand model of Φ.

2.3 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 2.2 Let Φ be a clause set containing, among others, the clauses:

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

Table 1 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.[5] 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 $\neg B(u)$.

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

[5] From the instance alone it should be easy to see which context unifier was used in the last four rule applications.

Context Literal	**Derivation Rule**	**Clause Instance**
$\ldots$	$\ldots$	$\ldots$
$A(t(x))$	Propagate	instance $A(t(x)) \vee \cdots$ of some clause in Φ where $t(x)$ is a term in the variable x.
$\ldots$	$\ldots$	$\ldots$
$B(u)^{\mathsf{d}}$	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)

Table 1: A trace of a derivation in the system B.

$n \geq 0$—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 2.2, and in perfect analogy to the Abstract DPPL framework of Nieuwenhuis *et al.* [NOT05], 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 $\Phi \models C$

In this very general formulation, learning is simply the addition, via an application of Learn, of an entailed clause to the clause set. While in principle one could learn any entailed clause, Learn is meant to be used to add only clauses that are more likely to cause further propagations and correspondingly reduce the number of needed decisions. The intended use of 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 first two methods we present below lift this regression to the first-order case, although with different degrees of generality. The first method is strictly subsumed by the second. 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 third and last method is less general. In our experiments we used it mostly as a sanity check against the other two methods because of its much lower overhead.

2.4 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 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

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

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

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 proved in the next two results, 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.

To start, every regression of the initial annotated lemma with GRegress generates a logical consequence of the clause set.

Lemma 2.3 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} C \mid S$, then the following holds.*

1. *For every $M \mapsto N \in S$, M is a (ground) instance of $\overline{N}$.*

2. *The (ground) clause C is a consequence of Φ.*

Proof. Suppose a regression derivation $A_0 \Longrightarrow^*_{\mathsf{gr}} C \mid S$ of length $l \geq 0$ as given. We directly prove the claim by induction on l.

$l = 0$*)* 1. By construction of A_0, $M = L\underline{\sigma_0}$ for some literal L and N is the propagated literal L^{σ_0}. By definition of context unifier we have that $L\sigma_0 = \overline{N}\sigma_0$. So M is a ground instance of $\overline{N}$.

2. Immediately by construction, as $C_0\underline{\sigma_0}$ is a ground instance of the closing clause.

$l > 0$*)* 1. For the mappings of S added by the application of the rule, the proof is analogous to the base case. For the others, the claims holds by induction.

2. Using the notation as introduced in the GRegress rule above, we prove first that GRegress preserves consequenceship.

With M being a (ground) instance of $\overline{L\sigma}$, as obtained by the induction hypothesis and 1, the most general unifier μ is in fact a matcher such that $\overline{L\sigma}\mu = M$. Thus, the clause $D \vee C\sigma\underline{\mu}$ is a (ground) instance of the resolution resolvent $D \vee C\sigma$ of the parent clause $D \vee M$, which is ground, and the parent clause $L\sigma \vee C\sigma$, where the mgu used is μ.

Now, the (ground) clause $D \vee M$ is a consequence of Φ by induction assumption and $L\sigma \vee C\sigma$ is an instance of a clause in Φ by construction. With the soundness of resolution it follows that $D \vee C\sigma\underline{\mu}$ is a consequence of Φ. ☐

Proposition 2.4 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{C} \mid S$,[6] the clause C obtained from $\underline{C}$ by replacing each constant of $\underline{C}$ not in Φ by a fresh variable is a consequence of Φ_0.*

[6]Here and below, we write $\underline{C}$ as a suggestive notation to denote a ground clause standing in a certain relation with another clause C.

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

Figure 1: Grounded regression of $\neg D(u)\vee\neg E(u)$.

Proof. By Lemma 2.3 and the Free Constants Theorem of first order logic. ☐

From a practical viewpoint, an important invariant 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 2.5 *If $A_0 \Longrightarrow^*_{\mathrm{gr}} A$ and A has the form $D\vee M \mid S, M\mapsto N$, then the* GRegress *rule applies to A.*

Proof. It is enough to show that M is an Assert literal in $\mathbf{D}$ and M is an instance of $\overline{N}$. The latter holds by Lemma 2.3, the former is easily provable again by induction on the length of regression derivations. ☐

Example 2.6 Figure 1 shows a possible regression of the conflicting clause $\neg D(x)\vee\neg E(x)$ in the derivation of Example 2.2. This clause is conflicting because of the context 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:

$$\begin{aligned} A_0 &= (\neg D(x)\vee\neg E(x))\underline{\sigma_0} \mid \{(\neg D(x))\underline{\sigma_0}\mapsto(\neg D(x))^{\sigma_0},\ (\neg E(x))\underline{\sigma_0}\mapsto(\neg E(x))^{\sigma_0}\} \\ &= \neg D(a)\vee\neg E(a) \mid \{\neg D(a)\mapsto D(u),\ \neg E(a)\mapsto E(u)\}\ . \end{aligned}$$

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 $(L\vee C)\sigma$ identified in the precondition of GRegress. The introduced fresh Skolem constants are a,b and c. Stopping the regression with the last resolvent in the derivation gives, after abstracting away the Skolem constants, the lemma $\neg A(t(z))\vee\neg B(x)$.[7] ☐

[7] The fact that the literals in this lemma are variable disjoint is not typical of the regression process. It is just a (nice) feature of this particular example.

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 $C = L_1 \vee \cdots \vee L_m$ 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 $\{K_1,\dots,K_m\}$.

Lemma 2.7 *If $A_0 \Longrightarrow^*_{\mathsf{gr}} \underline{E} \mid S$ and the clause E is obtained from $\underline{E}$ by replacing each constant of $\underline{E}$ not in Φ by a fresh variable, then E is conflicting in any context that contains E^Λ.*

Proof. See the appendix. □

Proposition 2.8 *Any lemma C produced from* **D** *by the regression method in this section is conflicting in any context that contains C^Λ.*

Proof. Follows immediately from Lemma 2.7 □

Proposition 2.8 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.

Example 2.9 Looking back at the lemmas generated in Example 2.6, it is easy to see that the lemma $\neg C(x,y) \vee \neg A(t(z)) \vee \neg C(x,t(z))$ becomes conflicting in the derivation of Table 1 as soon as $C(u,y)$ is added to the context. In contrast, the more regressed lemma $\neg A(t(z)) \vee \neg B(x)$ 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.

Example 2.10 Consider an extension of the derivation in Table 1, where the context has been undone enough that now its last literal is $A(t(x))$. By applying Propagate to the lemma $\neg A(t(z)) \vee \neg B(x)$ it is possible to add $\neg B(x)$ to the context, thus preventing the addition of $B(u)$ as a decision literal (because $B(u)$ is contradictory with $\neg B(x)$) and avoiding the conflict with clause (4). With the less regressed lemma $\neg A(t(z)) \vee \neg C(x,t(z))$ it is still possible to add $\neg B(x)$, but with two applications of Propagate—to the lemma and then to clause (1). $\square$

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, in addition to that it does have 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 $\neg A(t(z)) \vee \neg B(x)$ in our running example will become conflicting once the context contains *any* instance of $A(t(z))$ and $B(x)$, 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 Table 1 shows that the lemma $\neg A(t(z)) \vee \neg B(x)$ 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 $B(x)$, also any *generalization* of $A(t(z))$. So a better possible lemma is $\neg A(z) \vee \neg B(x)$. 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.

2.5 The Lifted Method

Consider again the derivation $\mathbf{D}$ 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. Starting with the annotated lemma

$$C'_0 \mid S'_0 \quad = \quad C_0\underline{\sigma_0} \mid \{L\underline{\sigma_0} \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\},$$

one can build a regression of the form

$$C'_0 \mid S'_0 \Longrightarrow_{\mathsf{gr}} C'_1 \mid T'_1 \Longrightarrow_{\mathsf{gr}} \ldots \Longrightarrow_{\mathsf{gr}} C'_n \mid T'_n \,.$$

We have seen that this regression determines a linear resolution derivation,whose derivation tree is depicted in Figure 2(a), where C'_0 and D'_i are instances of clauses in Φ, and C'_{i+1} is a resolvent of C'_i and D'_i for all $i = 0,\ldots,n-1$. Using basic results about resolution and unification, this derivation can be lifted to one of the form shown in Figure 2(b) where C_0 and each D_i are the clauses in Φ that C'_0 and D'_i are instances of, and each C_{i+1} is a resolvent of C'_i and D'_i and a generalization of C'_{i+1}.

Conceptually, 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

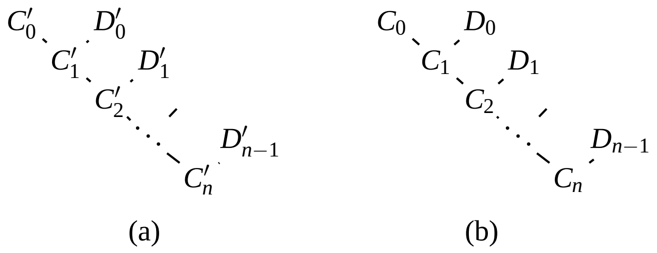

Figure 2: Grounded regression derivation and its lifting.

directly, without building the grounded derivation first. We do this by starting with the annotated lemma $C_0 \mid S_0 = C_0 \mid \{L \mapsto L^{\sigma_0} \mid L \in C_0 \text{ and } L^{\sigma_0} \text{ is a propagated literal}\}$, and regressing that lemma with the following lifted version of GRegress:

$$\textsf{Regress:} \quad D \vee M \mid S, M \mapsto L\sigma \implies_{\mathsf{r}} (D \vee C_{\mathrm{v}})\mu \mid S, T \quad \text{if } (*)$$

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

Proposition 2.11 *For every grounded regression*

$$C_0' \mid S_0' \implies_{\mathsf{gr}} C_1' \mid T_1' \implies_{\mathsf{gr}} \ldots \implies_{\mathsf{gr}} C_n' \mid T_n' \,,$$

there is a lifted regression

$$C_0 \mid S_0 \implies_{\mathsf{r}} C_1 \mid T_1 \implies_{\mathsf{r}} \ldots \implies_{\mathsf{r}} C_n \mid T_n \,,$$

such that $C_i \gtrsim C_i'$ and $\Phi \models C_i$ for all $i = 0, \ldots, n$.

Proof. The grounded regression can be written as a linear resolution derivation from ground instances of clauses from Φ. Using standard lifting arguments (see [CL73]) this derivation can be lifted to a derivation using the clauses from Φ instead of their instances, which, in turn, can be written as the lifted regression as stated. This proves $C_i \gtrsim C_i'$.

Regarding $\Phi \models C_i$, observe that $C_0 \in \Phi$ and, according to the just said, C_i is a resolution resolvent of C_{i-1} and some clause from Φ, for all $i = 1, \ldots, n$. Then $\Phi \models C_i$ follows by the soundness of the resolution inference rule. □

As in the grounded case then, we can use any regressed clause C as a lemma. In contrast, this time there are no constants to abstract, as the regression process resolves only input clauses of C. Again, the resulting clause is a logical consequence of Φ.

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

Figure 3: Lifted regression of $\neg D(x)\vee\neg E(x)$.

Example 2.12 Figure 3 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 2.2. 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 $L_\mathrm{v}\vee C_\mathrm{v}$ in the precondition of Regress. The lemma learned in this case is $\neg A(z)\vee\neg B(x)$. $\square$

2.6 The Propositional Method

Recall that each propagated literal L in a context is the result of a unification of a clause in the clause set with some previous context literals $K_1,\dots,K_n$. This sort of dependency of L on $K_1,\dots,K_n$ defines a *dependency graph* over context literals (whose roots are the context's decision literals) that can be used for conflict analysis. In fact, if C is a conflicting clause in a context Λ because of some context unifier σ, starting from the context literals used by σ and tracing the dependency graph backwards, one can precisely determine the set $\{L_1,\dots,L_k\}$ of decision literals that are ultimately responsible for the conflict. Then one could simply remember this set of decisions and make sure that they are not repeated again. The way we do this is to abstract each L_i by a unique (modulo p-renaming) propositional variable P_i, add the propositional clause $\overline{P_1}\vee\cdots\vee\overline{P_k}$ to the clause set, and from then on add P_i to the context each time L_i is added. Then the clause will become conflicting every time $L_1,\dots,L_k$ occur together again in a context. By applying Propagate to these propositional clauses, one can even avoid the conflict by not adding L_i again as a decision literal if $\overline{P_i}$ is present in the context.

The appeal of this method is that it is relatively cheap to generate and process this sort of lemmas. The downside is that these lemmas are less general than those computed with the previous methods, as they just cache *one* specific set of conflicting decisions.

3 Implementation

We implemented the three learning methods described in the previous section, in our $\mathcal{ME}$ theorem prover *Darwin*.[8] We briefly discuss this implementation and comment on a few details for improving performance.

[8] `http://goedel.cs.uiowa.edu/Darwin/`.

3.1 Lemma Generation

Since we merely take the relevant decision literals for a conflict as the pseudo-lemma in the propositional abstraction, the lemma generation methods of interest are grounded and lifted regression. For both cases we employ memoization to avoid regressing the same context literal more than once.

In the case of the grounded regression, memoization is done implicitly. Recall that here each literal to regress corresponds to a (negated) ground instance of a propagated context literal, which in turn depends itself on previous context literals in the context. It is easy to see that these dependencies between context literals determine a directed graph, called a *conflict graph* in the SAT literature, whose roots are the context literals associated to the conflict clause and whose leaves are decision literals. In the regression process, the literals in the current clause are regressed in an order that corresponds to a breadth-first exploration of the associated conflict graph. Among the literals at the same depth level in the graph, instances of more recent context literals are regressed first. This makes sure that all instances of the same context literal are regressed in a row. Now, by simply keeping the literals to regress in a set, each literal is automatically regressed only once.

The process is not as simple for the lifted method, as it in general involves unification operations, as opposed to just matching operations in the grounded case. More precisely, the regression process is implemented by maintaining three data structures, a set of all the literals in the central clause that we want to regress, a set of *regressed* literals (or *regression set*), literals that will not be regresses further according to some stop criterion, and set of unification constraints. If a literal chosen from the first set is not to be regressed, for example because it is paired with a decision literal, then it is simply moved to the second set. Otherwise, it is replaced by the literals in the corresponding side clause, and the unifier of the corresponding resolution step is added to the set of unification constraints. The regression stops when the first set is empty. At that point, the unification constraints are solved, and the resulting unifier is applied to the set of regressed literals, thus producing the lemma clause.

In this process memoization is achieved by doing the regression depth-first, based on the order of the context literals to regress. For each regressed literal its regressed literals and contraints are stored. Whenever the same literal is to be regressed again, this information is reused by creating a copy using fresh variables.[9] As an optimization and similar to the grounded case, a context literal is regressed only once if it is an instance of a ground clause literal.

[9] As described in [BFT06b] this does not require the creation of new terms, but merely replacing integer offsets.

3.2 Regression Depth

In analogy to the common procedure in SAT solvers, only literals propagated after the most recent decision literal responsible for the conflict are regressed. But in contrast this is not necessarily the most recent decision level. This is because, as allowed by the Backjump rule, *Darwin* only backtracks up to and including the most recent decision literal responsible for the conflict. This has the effect that a negated decision literal does not necessarily depend on the current decision level, and therefore a closure might not depend on the most recent decision literal. In contrast, the favored backjumping method in propositional solvers backtracks up to but excluding the *second* most recent responsible decision literal. Thus, propositional backjumping backtracks farther and is in a sense more eager. Experimental results have shown that this more eager form of backjumping is not beneficial in *Darwin*, as the right split does in general not prevent the jumped over decision literals and the subsequent propagations from being reasserted. Most of the times eager backjumping does not change the search space in a beneficial way, but instead introduces additional overhead.

An important optimization for propositional solvers is not to stop the regression at the most recent responsible decision literal, but already at a unique implication point (UIP). It is unclear how to lift this idea to the first-order level, though, as in general there may be several, distinct instances of a propagated literal used in a closure. The naive approach of treating all instances of the same context literal as a potential UIP did not turn out to be efficient in practice. Furthermore, while the UIP can be found automatically in the grounded regression, namely when the regression set contains only instances of exactly one context literal, this is not possible using the depth-first approach of the lifted regression. Here, either the regression needs to be done breadth-first, but then memoization cannot be used, or the UIP must be computed before the actual regression is performed.

As a side note we point out that, for the same reason as above, unlike for the propositional case lemmas can not be used in general to make the explicit addition of the negated decision literal unnecessary after backjumping. For example, a possible derivation part based on the clauses $P(a,b) \lor Q(a,x)$, $\neg P(a,b) \lor \neg Q(a,c)$, $P(x,b) \lor \neg Q(x,x)$, is Decide of $Q(a,x)$, Propagate of $\neg P(a,b)$, and Fail. This yields the grounded and lifted lemma $\neg Q(a,d) \lor \neg Q(a,a)$. While this lemma does become conflicting if Decide of $Q(a,x)$ is applied again, it does not prevent that application of Decide. To do that, the lemma would need to be unit. Due to their increased generality, this tends to happen more often with lifted than with grounded lemmas, making Propagate sometimes less efficient with lifted lemmas.

3.3 Simplification

The lemma computed in the grounded and lifted regression is simplified before usage. Note that a context literal asserted in the root decision level, that is before any decision literal is added to the context, is never regressed according to the description above. But, as it is implied by the clause set, its grounded regression does in essence correspond to a unit resolution step. As a consequence, root assert context literals do not need to be and are not added to the lemma. For the lifted regression the case is more complicated, as it is only directly applicable if the context literal is an instance of a unit clause. As in addition the contraint has to be computed as usual, which in some cases leads to a significant overhead due to too many constraints, root asserts are not handled special here.

Simplifying the lemma is particularly important in the lifted case because it is not unusual for the lemma regression process to produce very long lemmas, with several instances or variants of the same literal. As condensing is too expensive, we employ a simpler method which produces good results in practice with a linear number of unification tasks. If all instances of the same context literal in a lemma have a common instance, they are replaced by their most general common instance, and the corresponding unifier is applied to the remainder of the lemma. Then, if still several variants of a literal occur, they are condensed into one literal, and the renaming is again applied to the remainder of the lemma. Finally, duplicates of literals are removed (as clauses are treated as sets of literals).

Unfortunately, this method sometimes simplifies the lemma in an unwanted way, making it in effect useless. For example, if in a context the literals $\neg A(a)$, $\neg B(a)$, $\neg B(b)$, and $\neg C(b)$ lead to a conflict, then the learnt grounded lemma might be $A(a) \vee B(a) \vee B(b) \vee C(b)$, and the more general lifted lemma might be $A(x) \vee B(x) \vee B(y) \vee C(y)$. Now, its simplification, $A(x) \vee B(x) \vee C(x)$, can not be used to prevent the recreation of the conflicting context, and in fact not even to close on it.

3.4 Application

In principle, during a derivation of the proof procedure lemmas can be used like any other clause as far as the rules Decide, Propagate, and Fail are concerned. As a lemma's purpose is to prune the search space, applying Decide to lemmas does not seem like a sensible choice, as confirmed by our experimental results. Using lemmas for Fail applications and for selected applications of Propagate turned out to be the most efficient usage.

Furthermore, to reduce the context unifier computation overhead, potential propagations for a new lemma are not computed in retrospect, but only when adding a literal to the context unifies with the lemma. Therefore, when after an application of Backjump a lemma is learnt, it does not propagate the negated decision literal $\overline{L}^{\mathrm{sko}}$. This happens only, as an optimization, if the lemma is unit, which might in fact propagate a strictly p-preservingly more general literal than $\overline{L}^{\mathrm{sko}}$. In conjunction with the above described shortening of grounded lemmas with root context literals, this case occurs more often in the grounded than the lifted case, making the grounded lemma sometimes more effective than the corresponding lifted one.

In general, applications of Propagate are restricted in *Darwin* to those with *universal* propagated literals (i.e., containing no parameters). Adding non-universal propagated literals to the context is not only unnecessary for completeness but also counterproductive for efficiency because it substantially increases the number of context unifiers usable by Decide or Propagate. On the other hand, adding literals propagated by a lemma is useful to avoid conflicts, as we discussed earlier in the paper. In the current implementation, we strike a balance between these two conflicting needs by adding to the context a non-universal propagated literal only if the propagating clause has been learned as a lemma at least n times in the derivation—a crude but easily computed estimate of the lemma's usefulness in avoiding future conflicts. Experimentally, a value of $n = 3$ seems to give the best results.

3.5 Forget

At the moment we have implemented only a relatively unsophisticated scheme for forgetting lemmas, again inspired by similar schemes in the SAT literature. In this scheme, there exists an upper limit u and a lower limit l on how many lemmas are stored at any time. If a new lemma is learned after u has been reached, the *worst* lemmas are removed until there are only l lemmas left. The new lemma is then added to this smaller lemma set.

The value of a lemma is determined by a score, which is initially set to the worst score among the existing lemmas. Whenever the lemma is responsible for an application of Fail, i.e., the lemma is involved in the regression of a conflict, its score is incremented by 1. When the worst lemmas are removed, all scores are divided by 2. As an alternative, we also tried to decay the score periodically after a certain number of Backjump applications. This score is not currently used in the heuristics for choosing which lemma to propagate on, mostly because it is not trivial to integrate properly into the system's architecture. Unfortunately, these schemes did not lead to any improvement over not applying Forget at all.

4 Experimental Evaluation

In this section we present our initial experimental evaluation of the three learning methods presented above. We considered over two different problem sets.

4.1 First problem set

We first evaluated the effectiveness of lemma learning in *Darwin* over the TPTP problem library version 3.1.1. Since *Darwin* can handle only clause logic, and has no dedicated

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
propositional	895	2.8	2507.6	0.96	20056	570962	37102
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
propositional	243	3.4	821.1	0.87	19546	453577	32794
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
propositional	107	4.5	478.8	1.16	18703	392616	30209
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
propositional	66	4.5	289.7	1.12	17044	333648	27026
grounded	67	1.7	111.4	2.91	6973	183292	9879
lifted	70	2.3	151.4	2.14	7275	193097	10294

Table 2: 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 894 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.

inference rules for equality, we considered only clausal problems without equality. Furthermore, as *Darwin* never applies the Decide rule in Horn problems [Fuc04], 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 2 summarize the results for various configurations of *Darwin*, namely, not using lemmas and using lemmas with the propositional, 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. Indeed, 652 of the 898 solved problems 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, it is not clear if this an intrinsic property of the calculus, an artifact of the specific proof procedure implemented by *Darwin*, or a feature of the TPTP library.

For a more meaningful comparison, the rest of Table 2 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 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 [BFT06c], 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 2 between unsatisfiable and satisfiable problems. These data are provided respectively in Table 3 and Table 4.

The separated results for unsatisfiable and satisfiable problems show the same pattern as the aggregate results in Table 2. It is interesting to notice, however, that for the unsatisfiable problems solved by all configurations and solved by the no lemmas one with at least 0, 3, 20, and 100 backjumps the speed up factors for grounded lemmas are respectively 1.07, 1.55, 3.74, and 4.19. This actually compares more favorably overall to the corresponding speed up factors in Table 2: resp., 1.12, 1.60, 2.43, and 2.91.

Plotting the individual run times of the no lemmas configuration against the lemma configurations, and the grounded against the lifted lemmas configuration for all solved problems with at least 3 backjumps, as seen in Figure 4, clearly shows the positive effect of learning. For nearly all of the problems, the performance of the 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. In contrast, the plot for the propositional configuration looks considerably different, with few outliers for either configuration and basically all points closely clustered around the diagonal. Finally, the comparison of the grounded

Method	Solved Probls	Avg Time	Total Time	Speed up	Failure Steps	Propag. Steps	Decide Steps
no lemmas	563	3.3	1827.4	1.00	22741	495924	35831
propositional	562	3.5	1975.5	0.92	18478	476066	28959
grounded	561	3.0	1705.2	1.07	8336	294819	11620
lifted	562	3.1	1731.8	1.05	8610	300273	12004
no lemmas	193	1.9	364.4	1.00	22283	419920	35121
propositional	192	2.7	508.9	0.72	18020	399969	28249
grounded	191	1.2	234.7	1.55	7878	218739	10910
lifted	192	1.4	271.4	1.34	8152	224587	11294
no lemmas	89	2.9	255.6	1.00	21589	388200	34109
propositional	89	2.4	216.2	1.18	17390	352350	27328
grounded	90	0.8	68.2	3.74	7352	188032	10216
lifted	90	1.2	103.1	2.48	7615	194755	10581
no lemmas	61	3.7	226.4	1.00	20157	351521	32011
propositional	61	3.1	190.8	1.19	16169	317696	25570
grounded	61	0.9	54.0	4.19	6484	163481	9058
lifted	62	1.4	88.2	2.57	6748	170424	9429

Table 3: Unsatisfiable problems that respectively take at least 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 561, 191, 89, and 61 problems solved by all configurations.

and lifted learning methods shows that the gained generality of the latter almost never pays off in terms of run time, except that it allows the system to solve three additional problems.

Overall, the results above indicate that the propositional method is not nearly as effective at pruning the search space or decreasing the run time as the other two learning methods, confirming our hypothesis that generalizing pays off. They also show 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 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 Section 3).

Method	Solved Probls	Avg Time	Total Time	Speed up	Failure Steps	Propag. Steps	Decide Steps
no lemmas	333	1.7	569.6	1.00	2250	101362	9243
propositional	333	1.6	532.1	1.07	1578	94896	143
grounded	334	1.3	430.4	1.32	1140	96370	7315
lifted	336	1.3	441.6	1.29	1186	99252	7363
no lemmas	51	7.0	349.5	1.00	2198	60126	5645
propositional	51	6.2	312.2	1.12	1526	53608	4545
grounded	52	4.2	210.4	1.66	1088	55110	3717
lifted	54	4.4	222.3	1.57	1134	58013	3765
no lemmas	18	17.7	300.1	1.00	1964	47019	3970
propositional	19	9.4	160.3	1.87	879	40405	2063
grounded	21	10.1	171.3	1.75	920	43348	2107
lifted	18	15.4	262.6	1.14	1313	40266	2881
no lemmas	5	24.4	97.5	1.00	1398	19624	2277
propositional	5	24.7	98.9	0.99	875	15952	1456
grounded	6	14.4	57.4	1.70	489	19811	821
lifted	8	15.8	63.2	1.54	527	22673	865

Table 4: Satisfiable problems that respectively take at least 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 332, 50, 17, and 4 problems solved by all configurations.

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.

4.2 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 [BFT06a]. This application follows an approach similar to that of systems like Paradox [CS03]. To find a finite model of a given cardinality n, a clause set, with or without equality, is con-

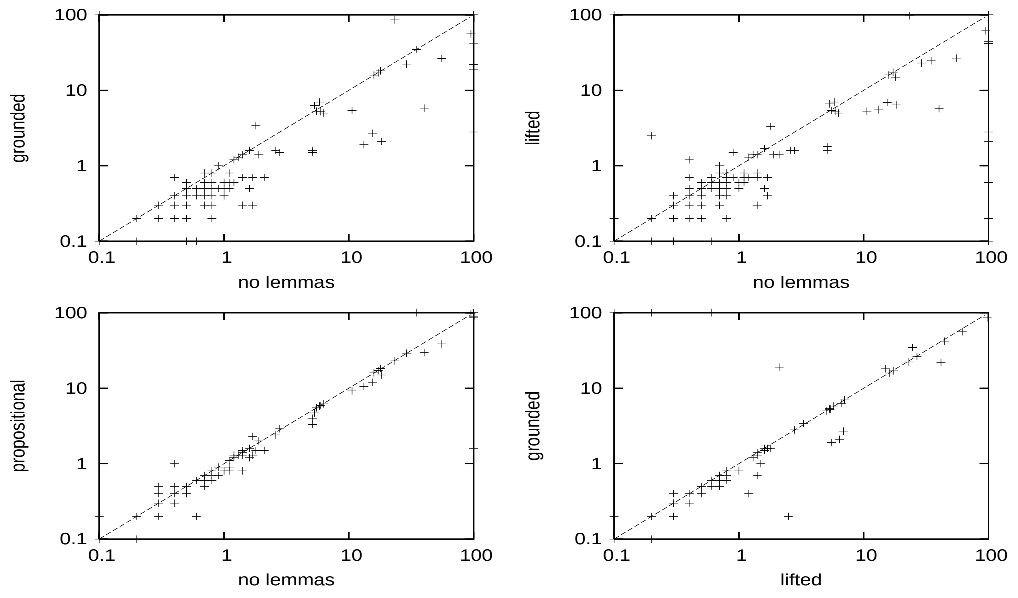

Figure 4: Comparative performance, on a log-log scale, for different configurations for problems with at least 3 applications of Backjump. For readability, the cutoff is set at 100s instead of 300s, because in all cases less than a handful of problems are solved in the 100-300s range.

verted 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 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 5 shows our results for the 815 satisfiable problems of the TPTP library. To give an idea how we compare to other systems, we remark that Mace 4 [McC03] 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.

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

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
propositional	658	4.4	2827.1	1.27	198023	7859965	351236
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
propositional	163	13.0	1971.1	1.37	193302	7659591	338074
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
propositional	53	20.5	961.9	1.77	161851	6540084	291492
grounded	64	10.5	495.3	3.44	53486	3100339	64845
lifted	57	11.5	538.7	3.16	26154	678319	39873

Table 5: 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.

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 significantly reduced, 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 confirms that lemmas do have a significant positive effect if the focus in solving a problem lies on search instead of constraint propagation.

5 Conclusion and Further Work

We have introduced three methods for implementing conflict-based learning in proof procedures for the Model Evolution calculus. The methods have various degrees of generality, implementation difficulty, and practical effectiveness. 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 [SE94], 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 besides those in the TPTP library.

References

[AS92] Owen L. Astrachan and Mark E. Stickel. Caching and Lemmaizing in Model Elimination Theorem Provers. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, LNAI 607, pages 224–238. Springer-Verlag, 1992.

[BFT06a] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. First-order methods for computing finite and minimal finite models. In Preparation, May 2006.

[BFT06b] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.

[BFT06c] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. Technical report, The University of Iowa, 2006.

[BT03a] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[BT03b] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model building. In Peter Baumgartner and Christian G. Fermüller, editors, *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, 2003.

[Fuc04] Alexander Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master’s thesis, University of Koblenz-Landau, 2004.

[GP00] Matthew L. Ginsberg and Andrew J. Parkes. Satisfiability algorithms and finite quantification. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR'2000)*, pages 690–701. Morgan Kauffman, 2000.

[Lov78] D. Loveland. *Automated Theorem Proving - A Logical Basis*. North Holland, 1978.

[LS01] Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 28, pages 2017–2114. Elsevier, 2001.

[McC03] William McCune. Mace4 reference manual and guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, 2003.

[NOT05] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[SE94] A. Segre and C. Elkan. A high-performance explanation-based learning algorithm. *Artificial Intelligence*, 69:1–50, 1994.

A Proofs

Lemma 2.7 *If $A_0 \Longrightarrow^*_{\mathrm{gr}} \underline{E} \mid S$ and the clause E is obtained from $\underline{E}$ by replacing each constant of $\underline{E}$ not in Φ by a fresh variable, then E is conflicting in any context that contains E^Λ.*

Proof. Suppose a regression derivation $A_0 \Longrightarrow^*_{\mathrm{gr}} \underline{E} \mid S$ of length $l \geq 0$ as given. We directly prove the claim by induction on l.

$l = 0$) By construction of A_0, $\underline{E}$ is the clause $C_0\underline{\sigma_0}$, a ground instance of some clause C_0 which is conflicting in Λ because of the context unifier σ_0. If $C_0 = L_1 \vee \cdots \vee L_n$, for some $n \geq 0$, the context unifier σ_0 of C_0 exists against any context containing $\{L_1^{\sigma_0}, \ldots, L_n^{\sigma_0}\}$, which is, by definition, the set E^Λ.

The constants in $C_0\underline{\sigma_0}$ and not in Φ are just the fresh constants introduced by $\underline{\sigma_0}$. Thus, $E = C_0\sigma_0$. Because any standard unification algorithm computes idempotent unifiers, it is safe to assume that the context unifier σ_0 is idempotent. It follows $C_0\sigma_0 = C_0\sigma_0\sigma_0$, and thus σ_0 is a context unifier of E against any context containing $\{L_1^{\sigma_0}, \ldots, L_n^{\sigma_0}\}$.

$l > 0$) We use notation similar as introduced in the GRegress rule above. Thus let

$$\underline{D} \vee \underline{M} \mid S', \underline{M} \mapsto L\sigma \Longrightarrow_{\mathrm{gr}} \underline{D} \vee C\sigma\underline{\mu} \mid S', T$$

be the last GRegress application (i.e., $\underline{E} = \underline{D} \vee C\sigma\underline{\mu}$). We assume by induction the result to hold for $\underline{D} \vee \underline{M}$, i.e., that $D \vee M$ is conflicting in any context that contains $(D \vee M)^\Lambda$, where $D \vee M$ is obtained from $\underline{D} \vee \underline{M}$ by replacing each constant of $\underline{D} \vee \underline{M}$ not in Φ by a fresh variable. We will directly show that under these assumptions $E = D \vee C\sigma\mu$ is conflicting in any context that contains E^Λ.

Let $(D \vee M)^\Lambda = \{K_1^\delta, \ldots, K_m^\delta, M^\delta\}$, where $D = K_1 \vee \cdots \vee K_m$, for some $m \geq 0$, and δ the context unifier such that $D \vee M$ is conflicting with Λ because of δ. As $D \vee M$ is conflicting with Λ because of δ, δ moves parameters to parameters only. More precisely, by construction $D \vee M$ is parameter-free, and the only parameters moved by δ can thus be assumed to be those in $\{K_1^\delta, \ldots, K_m^\delta, M^\delta\}$, and it holds $(\mathcal{P}ar(\{K_1^\delta, \ldots, K_m^\delta, M^\delta\}))\delta \subseteq V$. We will need this result further below.

By the above notation, the clause $\underline{E}$ is of the form $\underline{D} \vee C\sigma\underline{\mu}$, for some context unifier σ of a clause $L \vee C$ against some context literals of Λ, where $L\sigma$ is a propagated literal and μ is a most general unifier of $\underline{M}$ and $\overline{L\sigma}$ (in fact, μ is a matcher of $\overline{L\sigma}$ to $\underline{M}$, as $\underline{M}$ is ground). For further use below, we write the clause $L \vee C$ as $L \vee L_1 \vee \cdots \vee L_n$, for some $n \geq 0$. The literals paired with $L \vee C$ by σ then are denoted by $\{L^\sigma, L_1^\sigma, \ldots, L_n^\sigma\}$.

The substitution μ can be written as $\mu = \mu' \circ \gamma$, where μ' is a mgu of M and $\overline{L\sigma}$ (in fact, a matcher of $\overline{L\sigma}$ to M) and γ is a substitution that moves all the parameters and variables in M to the fresh constants such that $M\gamma = \underline{M}$. Assume that γ has furthermore been extended to move all the remaining parameters and variables in $C\sigma\mu$ to fresh constants. It follows $C\sigma\underline{\mu} = C\sigma\mu'\gamma$, and, with $\underline{E} = \underline{D} \vee C\sigma\underline{\mu}$ we get $E = D \vee C\sigma\mu'$.

With $(D \vee M)^\Lambda = \{K_1^\delta, \ldots, K_m^\delta, M^\delta\}$ from above and $\{L^\sigma, L_1^\sigma, \ldots, L_n^\sigma\}$ being the literals paired with $L \vee C$ by σ we get $(D \vee C\sigma\mu')^\Lambda = \{K_1^\delta, \ldots, K_m^\delta, L_1^\sigma, \ldots, L_n^\sigma\} (= E^\Lambda)$.

It remains to prove that $D \vee C\sigma\mu'$ is conflicting in any context that contains $(D \vee C\sigma\mu')^\Lambda$. For this, we show that the substitution $\sigma\mu'\delta$ is a context unifier of $D \vee C\sigma\mu'$ against Λ with paired literals $\{K_1^\delta, \ldots, K_m^\delta, L_1^\sigma, \ldots, L_n^\sigma\}$. It suffices to take a literal K_i from D and a literal L_j from C arbitrary and to show that

(1) (a) $K_i\sigma\mu'\delta = \overline{K_i^\delta}\sigma\mu'\delta$ and (b) $(L_j\sigma\mu')\sigma\mu'\delta = \overline{L_j^\sigma}\sigma\mu'\delta$, and

(2) (a) $(\mathcal{P}ar(K_i^\delta))\sigma\mu'\delta \subseteq V$ and (b) $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$.

First we show that neither σ nor μ' act on K_i, i.e., that $K_i\sigma = K_i$ and $K_i\mu' = K_i$ hold: the literal K_i is a literal from D, which is obtained from the (ground) clause $\underline{D}$ by replacing each constant in $\underline{D}$ not in Φ by a fresh variable. Thus, K_i is parameter-free and all its variables are disjoint from the variables in L and $L\sigma$. Thus, the context unifier σ (of the clause $L \vee C$) need not act on the clause $D \vee M$, and hence in particular not on K_i, which implies $K_i\sigma = K_i$. Similarly, recall that μ' is a matcher of $\overline{L\sigma}$ to M. Clearly we may assume μ' to move the variables and parameters of $\overline{L\sigma}$ only. With the freshness of the variables in $D \vee M$, thus, $K_i\mu' = K_i$.

Next we show, similarly, that neither σ nor μ' acts on $\overline{K_i^\delta}$, i.e., that $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$ hold. For this, we need the fact that context literals used in context unifiers are fresh. Thus, neither σ nor μ' acts on $\overline{K_i^\delta}$, which is a context literal of the context unifier δ, and the stated equalities follow.

From $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$ and together with$K_i\sigma = K_i$, $K_i\mu' = K_i$ the above equation (1-a) is equivalent to $K_i\delta = \overline{K_i^\delta}\delta$, which holds trivially by notation.

With $\overline{K_i^\delta}\sigma = \overline{K_i^\delta}$ and $\overline{K_i^\delta}\mu' = \overline{K_i^\delta}$, condition (2-a) is equivalent to $(\mathcal{P}ar(K_i^\delta))\delta \subseteq V$. Recall that $D \vee M$ is conflicting with Λ because of δ. By definition, δ thus does not have a remainder, and $(\mathcal{P}ar(K_i^\delta))\delta \subseteq V$ follows in particular.

It remains to prove conditions (1-b) and (2-b).

Regarding (1-b), it is safe to assume that σ is idempotent. Recall that μ' is a matcher form $L\sigma$ to M, and all variables of M are fresh, as argued further above. It is not difficult to see that $\sigma\mu'$ must be idempotent, too. Thus (1-b) is equivalent to $L_j\sigma\mu'\delta = \overline{L_j^\sigma}\sigma\mu'\delta$. This, however, follows trivially from $L_j\sigma = \overline{L_j^\sigma}\sigma$, which holds by notation.

Regarding (2-b), notice first $(\mathcal{P}ar(L_j^\sigma))\sigma \subseteq V$. This holds because $L\sigma$ is a propagated literal and, by definition of Propagate, none of the literals in $C\sigma$ is a remainder literal. In particular, thus $(\mathcal{P}ar(L_i^\sigma))\sigma \subseteq V$. Next we will extend this inequality and obtain $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$, which will complete the proof.

Recall from the regression step we are considering that $\underline{M}$ is paired with $L\sigma$. Recall further that $D \vee M$ is conflicting with Λ because of δ. The literal paired with M in the context unifier δ is thus a fresh p-variant of $L\sigma$, say, $L\sigma\rho$ for some appropriate p-renaming ρ. That is, $M\delta = \overline{L\sigma}\rho\delta$. We also know that $\overline{L\sigma}$ can be instantiated to M by μ'.

That is, $\overline{L\sigma}\mu' = M$. Applying δ yields $\overline{L\sigma}\mu'\delta = M\delta = \overline{L\sigma}\rho\delta$. Now, as $D \vee M$ is conflicting because of δ, δ has no remainder literals. In particular, thus, $(\mathcal{P}ar(L\sigma\rho))\delta \subseteq V$. From this it is not too difficult to see that $\rho\delta$ maps all parameters of $L\sigma$ to parameters. With $\overline{L\sigma}\mu'\delta = M\delta = \overline{L\sigma}\rho\delta$ it follows that $\mu'\delta$ maps all parameters of $\overline{L\sigma}$ to parameters. Recall that μ' can be restricted to move parameters and variables of $L\sigma$ only. All other parameters that $\mu'\delta$ moves are moved by δ to parameters (because δ has no remainder literals). Together thus we obtain from $(\mathcal{P}ar(L_j^\sigma))\sigma \subseteq V$ the desired result $(\mathcal{P}ar(L_j^\sigma))\sigma\mu'\delta \subseteq V$. $\square$