International Journal on Artificial Intelligence Tools
Vol. 15, No. 1 (2006) 21–52

World Scientific
www.worldscientific.com

IMPLEMENTING THE MODEL EVOLUTION CALCULUS

PETER BAUMGARTNER

Programming Logics Group
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
baumgart@mpi-sb.mpg.de

ALEXANDER FUCHS*,†

Department of Computer Science
The University of Iowa
14 MacLean Hall
Iowa City, Iowa 52242, USA
fuchs@cs.uiowa.edu

CESARE TINELLI

Department of Computer Science
The University of Iowa
14 MacLean Hall
Iowa City, Iowa 52242, USA
tinelli@cs.uiowa.edu

Received 2 December 2004
Accepted 2 February 2005

Darwin is the first implementation of the Model Evolution Calculus by Baumgartner and Tinelli. The Model Evolution Calculus lifts the DPLL procedure to first-order logic. *Darwin* is meant to be a fast and clean implementation of the calculus, showing its effectiveness and providing a base for further improvements and extensions.

Based on a brief summary of the Model Evolution Calculus, we describe in the main part of the paper *Darwin*'s proof procedure and its data structures and algorithms, discussing the main design decisions and features that influence *Darwin*'s performance. We also report on practical experiments carried out with problems from the CASC-J2 system competition and parts of the TPTP Problem Library, and compare the results with those of other state-of-the-art theorem provers.

Keywords: Automated theorem proving; Davis-Putnam-Logemann-Loveland procedure.

*Part of the implementation work on *Darwin* was done by this author while a Master student at the University of Koblenz-Landau, Germany, and during employment at the Max-Planck-Institute for Computer Science, Saarbrücken, Germany.

†The work of the second and third author was partially supported by Grant No. 237422 from the United States National Science Foundation.

1. Introduction

In propositional satisfiability the DPLL procedure[1,2] is the most popular and successful method for building complete SAT solvers. Its success is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics and constraint propagation techniques for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and clauses.

Although the DPLL method is usually described procedurally, its essence can be captured declaratively by means of a sequent-style calculus.[3] The DPLL calculus has been recently lifted to the first-order level in (Ref. 4). The result is a sound and complete calculus, called the Model Evolution calculus, or $\mathcal{ME}$ calculus for short, for the unsatisfiability of first-order clauses (without equality).[a]

One of the main motivations for developing the Model Evolution calculus was the possibility of migrating to the first-order level some of those very effective search techniques developed by the SAT community for the DPLL procedure. This paper describes *Darwin*, a first implementation of the calculus designed to incorporate these techniques — or better, their first-order equivalents. The current version of *Darwin* implements a first-order version of unit propagation,[6] a form of simplification, and backjumping, a form of intelligent backtracking which seems to have been used for the first time for a first-order theorem prover in (Ref. 7). The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version.

Although *Darwin* is not as well developed as other theorem provers for previous calculi, it borrows many advanced techniques from the first-order theorem proving world — such as term indexing, subterm sharing, redundancy elimination, and so on.

The overall rationale for developing this system was to get an initial sense of the performance potential of the $\mathcal{ME}$ calculus, to constitute a robust code base for further improvements on the implementation, and for future extensions of the calculus.

This paper presents a fairly high level description of *Darwin*'s architecture and implementation, usually providing more details only on those implementation aspects that are specific to the $\mathcal{ME}$ calculus — as opposed to first-order calculi in general. The paper starts with a brief description of the $\mathcal{ME}$ calculus in Section 2. *Darwin*'s main proof procedure and how it relates to the $\mathcal{ME}$ calculus is explained in Section 3. Implementation issues are discussed in Section 4. Section 5 describes our experimental evaluation of *Darwin* and compares its performance to that of other state-of-the-art theorem provers. Section 6 concludes with further research directions.

[a]The $\mathcal{ME}$ calculus extends and significantly improves on the FDPLL calculus,[5] which was the first successful attempt to lift the DPLL calculus to the first-order level.

2. The Model Evolution Calculus

In this section, we introduce the Model Evolution calculus and its main features, concentrating on those aspects that are relevant to the understanding of the implementation. More details on the calculus can be found in Refs. 4 and 8.

The DPLL procedure can be described as one that attempts to find a model of a given formula, input as a set of clauses, by starting with a default interpretation in which all input atoms are false, and incrementally modifying it until it becomes a model of the input formula, or all alternative modifications have been considered with no success. The $\mathcal{ME}$ calculus is a lifting of this *model evolution* process to the first-order level.

The goal of the calculus is to construct a Herbrand model of a given set Φ of clauses, if any such model exists. To do that, during a derivation the calculus maintains a *context* Λ, a finite set of (possibly non-ground) literals. The context Λ is a finite — and compact — representation of a Herbrand interpretation I_Λ, serving as a candidate model for Φ. The denoted interpretation I_Λ might not be a model of Φ because it does not satisfy some instances of clauses in Φ. The purpose of the main rules of the calculus is to detect this situation and either *repair* I_Λ, by modifying Λ so that it becomes a model of Φ, or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number of simplification rules whose purpose is, like in the DPLL procedure, to simplify the clause set and, as a consequence, speed up the computation.

The rules of the calculus manipulate sequents of the form $\Lambda \vdash \Phi$, where Λ is the current context and Φ is the current clause set. The initial sequent is made of a context standing for an initial interpretation, and of the input clause set. We use the notation as in $\Lambda, L \vdash \Phi, C$ to stand for the sequent $\Lambda \cup \{L\} \vdash \Phi \cup \{C\}$.

To describe the rules we need to introduce a few technical preliminaries first.

2.1. *Technical Preliminaries*

Contexts are finite sets of possibly non-ground literals built over terms as usual, however over two types of variables: *universal* variables — or simply *variables* — drawn from an infinite set X and denoted here by x, y, z, and *parametric* variables — or simply, *parameters* — drawn from an infinite set V disjoint with X and denoted here by u, v, w. Context literals are either *universal*, that is parameter-free, or *parametric*, that is, variable-free. By contrast, clause literals, that is, literals occurring in the clause set Φ of a sequent, are all parameter-free. For all purposes, the literals of a context can be considered variable and parameter disjoint with each other — in tableaux terms, neither parameters nor variables are rigid.

Each context can be seen as the finite specification of a certain Herbrand interpretation. Roughly speaking, within a context both universal and parametric literals stand for their ground instances. However, a universal literal stands for all of its ground instances with no exceptions, whereas a parametric literal stands for all of

its instances that are not instances of another literal in the context with opposite sign.

The precise way in which context literals denote ground instances and how that is used to associate a Herbrand model to a context is formally defined in Refs. 4 and 8. Operationally, the main difference between universal and parametric literals in a context is that the former impose stronger restrictions on the extension of the context with additional literals (see Refs. 4 and 8 again for more details). Here we will limit ourselves to introduce a few notions that involve parameters and are needed to describe the rules of the calculus.

Let us consider the set of substitutions defined over the set $X \cup V$. We say that a substitution is *parameter-preserving*, or *p-preserving* for short, if its restriction to the set V of parameters is a renaming over V in the standard sense — i.e., it is a bijection of V onto itself. A substitution is a *p-renaming* if it is a p-preserving renaming. In other words, a p-preserving substitution maps parameters only to parameters, and in a bijective way, but can map variables to any term. A p-renaming substitution maps parameters to parameters, variables to variables, and is bijective.

We say that a term s is a *p-preserving variant* of a term t, or *p-variant* for short, if there is a p-renaming ρ such that $s\rho = t$. We say that s *is p-preserving more general than* t, iff there is a p-preserving substitution σ such that $s\sigma = t$. 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. These definitions, stated for terms, also apply to literals and clauses in the obvious way.

We assume an infinite supply of *Skolem constants* disjoint with the set of constants occurring in any given input clause set. We write L^{sko} to denote the result of applying some substitution to the literal L that replaces each variable in L by a fresh Skolem constant.[b] We write $\overline{L}$ to the denote the complement of L.

A literal L is *contradictory with* a context Λ iff there is a p-variant K of some literal in Λ and a p-preserving substitution σ such that $L\sigma = \overline{K}\sigma$.

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-preserving variants $K_1, \ldots, K_n$ of context literals 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* iff for all distinct $i, j = m+1, \ldots, n$, $L_i\sigma$ is either universal or parametric and

[b]Note that parameters are left untouched in L^{sko}.

$\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$. Each of the literals $L_{m+1}\sigma, \ldots, L_n\sigma$ is called a *remainder literal*.

Admissible context unifiers are needed in the main derivation rule of the calculus, the Split rule. An important property of admissible context unifiers is that they can be always derived from non-admissible ones with the help of an appropriate renaming.

Example 2.1. Consider the context $\Lambda = \{\neg p(u, v, w),\ \neg q(u', v')\}$ and the clause $C = p(x, y, a) \vee q(x, a)$. The substitution $\sigma = \{u \mapsto x, v \mapsto y, w \mapsto a, u' \mapsto x, v' \mapsto a\}$ is a context unifier of C against Λ. All the literals of its instance $C\sigma = C$ are remainder literals. Now, σ is not admissible because its remainder literals $p(x, y, a)$ and $q(x, a)$ are not variable disjoint. This problem can be solved by replacing both occurrences of x by, say, u. However, the resulting remainder $p(u, y, a) \vee q(u, a)$ (of the context unifier $\sigma' = \sigma \cdot \{x \mapsto u\}$) is not admissible either, because its literal $p(u, y, a)$ contains both a variable and a parameter. As above, replacing the offending variable y by a parameter, say, v will solve the problem, and the remainder $p(u, v, a) \vee q(u, a)$ (of the context unifier $\sigma'' = \sigma' \cdot \{y \mapsto v\}$) is admissible.

The existence of a context unifier Λ between a context and a clause indicates that the interpretation I_Λ denoted by Λ may falsify the clause.[c] The rules of the $\mathcal{ME}$ calculus use context unifiers as a way to discover that the interpretation associated with the current context falsifies one of the current clauses, and decide how to "repair" the context.

Context unifiers are at the core of the $\mathcal{ME}$ calculus because they are used by all of its non-optional derivation rules. In fact, context unification is the computational bottleneck of our current implementation as most of *Darwin*'s run time is spent on computing context unifiers. *Darwin*'s algorithm and data structure to compute context unifiers are described in Section 4.7.

2.2. *The Derivation Rules*

The derivation rules of the $\mathcal{ME}$ calculus are described below. We describe the rules as given in (Ref. 8), since those described in (Ref. 4) are a somewhat simplified but less powerful version. Except for Compact, which is a simplification rule that applies only to contexts with variables/parameters, all the other rules are direct first-order liftings of the rules of the DPLL calculus in (Ref. 3), and reduce to those rules when the input clause set is ground.

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\ C \vee L}{\Lambda,\ L\sigma \vdash \Phi,\ C \vee L \qquad \Lambda,\ (\overline{L\sigma})^{\text{sko}} \vdash \Phi,\ C \vee L}\quad \text{if } (*)$$

[c]More accurately, the clause is falsified if the context unifier is also *productive* (see Ref. 4). But we can gloss over this issue here.

$$\text{where } (*) = \begin{cases} C \neq \Box, \ (\Box \text{ is the empty clause}) \\ \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

Split is the only non-deterministic rule of the calculus. As mentioned earlier, the existence of a context unifier σ of $C \vee L$ against Λ indicates that I_Λ possibly falsifies $(C \vee L)\sigma$. The left conclusion of the rule tries to fix this potential problem by adding to the context a literal $L\sigma$ from σ's remainder. The alternative right conclusion — needed for soundness in case the repair on the left turns out to be unsuccessful — adds instead the skolemized complement of $L\sigma$, i.e. the result of replacing all universal variables of $L\sigma$, if any, by fresh Skolem constants. The addition of $(\overline{L\sigma})^{\text{sko}}$ prevents later splittings on L but leaves the possibility of repairing the context by adding another of σ's remainder literals. When the rule is applicable, we call $L\sigma$ a *split literal*.

$$\text{Assert} \quad \frac{\Lambda \quad \vdash \Phi, \ C \vee L}{\Lambda, L\sigma \ \vdash \Phi, \ C \vee L} \quad \text{if} \quad \begin{cases} \sigma \text{ is a context unifier of } C \text{ against} \\ \Lambda \text{ with an empty remainder,} \\ L\sigma \text{ is universal and} \\ \text{non-contradictory with } \Lambda, \\ \text{there is no } K \in \Lambda \text{ s. t. } K \text{ is} \\ \text{p-preserving more general than } L\sigma \end{cases}$$

When Assert applies, the only way to find a model for the clause set based on the current context or any extension of it is to satisfy every ground instance of $L\sigma$. The addition of $L\sigma$ makes sure that this is the case. Applications of Assert are highly desirable in practice because (i) they strongly constrain further changes to the context, thereby limiting the non-determinism caused by the Split rule, and (ii) they cause more applications of the three simplification rules below. When the rule is applicable, we call $L\sigma$ an *assert literal*.

$$\text{Subsume} \quad \frac{\Lambda, \ K \ \vdash \Phi, \ L \vee C}{\Lambda, \ K \ \vdash \Phi} \quad \text{if } \ K \text{ is p-preserving more general than } L.$$

Subsume removes clauses that are "permanently satisfied" by the context, that is, satisfied by the interpretation denoted by the current context or any context that extends the current one. Subsume is not needed for completeness but can in principle improve the performance of an implementation by reducing the number of clauses to be considered.

$$\textsf{Resolve} \quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda \vdash \Phi,\ C} \quad \text{if} \quad \begin{cases} \text{there is a context unifier } \sigma \text{ of } L \\ \text{against } \Lambda \text{ with an empty remainder} \\ \text{such that } C\sigma = C \end{cases}$$

Resolve simplifies the clause set by removing literals from clauses. Like Subsume it is not needed for completeness. Resolve is the only rule of the calculus that is not implemented in its full generality in *Darwin*. In the current implementation Resolve is only applied for the special case in which there is a K in Λ s.t. $\overline{K}$ is p-preserving more general than L. This unification test is done once for each literal L in Φ.[d] Thus, the check is more efficient than in the general case. Comparative evaluations with an early version of *Darwin* indicate that the restricted version fares better in general, despite its more limited applicability.

$$\textsf{Compact} \quad \frac{\Lambda,\ K,\ L \vdash \Phi}{\Lambda,\ K \quad \vdash \Phi} \quad \text{if } \ K \text{ is p-preserving more general than } L$$

In the rule's premise the literals K and L are meant to be distinct. Compact, which is another optimization rule, simplifies the context by removing literals that are parameter-preserving instances of other literals. Such literals are redundant and can be safely eliminated.

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

Close detects a context which falsifies the clause set and cannot be modified in order to satisfy it. When the rule is applicable, we call σ a *closing context unifier*.

2.3. *Derivation Tree*

Similarly to sequent calculi, derivations the $\mathcal{ME}$ calculus are formally defined in terms of derivation trees.

Definition 2.2. (Derivation Tree) A *derivation tree* is a labeled tree inductively defined as follows:

(1) a one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set;

[d]Since variable names in clauses are normalized in *Darwin*, it is possible for the same literal to occur in different clauses of Φ.

(2) A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new successor nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$ *is derived from* $\mathbf{T}$.

A branch in a derivation tree is *closed* if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$; otherwise, the branch is *open*. A derivation tree is *closed* if each of its branches is closed, and it is *open* otherwise. A *derivation* is a possibly infinite sequence of derivation trees $(\mathbf{T}_i)_{i<\kappa}$, such that for all i with $0 < i < \kappa$, $\mathbf{T}_i$ is derived from $\mathbf{T}_{i-1}$. For a given input clause set Φ, derivations are started with the single node tree containing the sequence $\neg v \vdash \Phi$. Here, the pseudo-literal $\neg v$ causes the interpretation denoted by the context to falsify every atom by default.

A derivation ending with a closed derivation tree is a proof of the unsatisfiability of Φ. An *exhausted* branch is a witness of the satisfiability of Φ, because it defines a model of the initial clause set. When the branch is finite this model is simply the interpretation induced by the context in the last node of the branch.

The formal definition of exhausted branch in $\mathcal{ME}$ is rather technical and based on the notion of *limit tree* of a derivation;[e] we refer the reader to (Ref. 4, 8) for it. Intuitively, however, it states that a branch in a limit tree is exhausted whenever, for each non-optional rule of the calculus that applies to a node N in the branch, the *intended*, if not literal, effect of the rule is achieved in some node down the branch. For instance, the intended effect of Split is that the possibility to falsify its selected clause by means of the context literals used in the context unifier is removed in some descendant node.

An important aspect to guarantee refutational completeness is to equip the calculus with a suitable notion of *fairness*. For $\mathcal{ME}$ this simply states that a derivation is fair it its limit tree is closed or has an exhausted branch. A proof procedure for the calculus is *fair* if it generates only fair derivations. We explain how *Darwin* produces fair derivations in the next section, where we present its proof procedure.

3. The Proof Procedure

The proof procedure implemented in *Darwin* follows the main loop described below. Similarly to DPLL, *Darwin*'s procedure basically explores the limit tree of a derivation in the calculus in a depth-first fashion. For fairness concerns, however, in *Darwin* this exploration is bounded, and done repeatedly with increasingly larger bounds. The present bounds are on the complexity of potential Assert or Split literals, as explained in Subsection 4.4.

At any moment, the procedure stores in its data structures a (sub)branch of the limit tree where *split nodes*, that is, nodes to which Split has been applied, correspond to choice points. The procedure grows the current branch until:

[e]This is obtained as the graph-theoretic union of all the trees in the derivation.

- the branch can be closed, in which case it backtracks to a previous choice point and regrows the branch in the alternative direction, or
- the branch cannot be grown further, which means that the branch is exhausted and a model of the input set has been found, or
- a certain limit is reached, in which case the procedure moves on to another branch or restarts with an increased limit, depending on the current search strategy.[f]

During the computation, in addition to the current context and the set of current clauses, the procedure maintains a set of *candidate literals*, literals that could be added to the context as a consequence of the application of the Assert or Split rule. Before entering the main loop, the candidate set is initialized with all the literals that could be added to the initial context by an application of Assert. By definition of Assert, these are exactly the literals occurring in unit input clauses.

The main loop of *Darwin*'s proof procedure consists of the following steps:

(1) **Candidate Selection.** If the candidate set contains no candidates suitable for an application of Split or Assert, the procedure ends returning the current context, which denotes a model of the input clause set.[g] If all applicable candidates exceed the current complexity bound, the procedure abandons the current branch and continues the search with an increased bound, as described in Section 4.4. Otherwise, it chooses an applicable candidate from the candidate set according to the selection heuristics described in Section 4.8. The heuristics is based on various metrics, but it always prefers Assert candidates over Split candidates, in order to minimize the creation of choice points.

(2) **Context Evolution.** If the selected literal is a Split literal, a choice point is created — corresponding to the left part of the application of the Split rule. Then, the literal is added to the context, the Compact rule is exhaustively applied to the new context, and the Subsume and Resolve rules are exhaustively applied to the current clause set using the newly added context literal.

(3) **Context Unifier Computation.** All possible context unifiers between current clauses and the new context are computed which involve the new context literal. If this leads to the computation of a closing context unifier, the current branch is immediately closed, forcing the procedure to backtrack.

(4) **Backtracking.** If a closing context unifier is found in the previous step, the current context does not satisfy the input clause set and is unrepairable. The procedure then backtracks to a previous choice point, undoing all changes to the context, the clause set, and the candidate set done from that choice point

[f] The available strategies are described in Subsection 4.4

[g] Note that initially suitable candidates, i.e., candidates meeting the preconditions of Split or Assert, might be no longer applicable after new literals have been added to the context, because for instance they have become subsumed by or contradictory with the new extended context. Instead of purging the candidate set of these literals, *Darwin* simply tests that a candidate is still applicable at the time it is selected, discarding it and selecting another one if it is not.

on. Since the choice point corresponds to the left part of the application of the Split rule which added a literal L to the context, the right part of the application is then tried. The skolemized complement of L is selected for addition to the context and the computation continues with Step 2.
If there are no more choice points to backtrack to, the input set has been proven unsatisfiable, and the procedure quits.

(5) **Candidate Generation.** If no closing context unifier is found in Step 3, the procedure extracts from the computed context unifiers all literals suitable for an application of Assert and from each remainder the *best* literal suitable for an application of Split, adds them to the candidate set, and goes back to Step 1.

Selecting only one candidate per remainder in Step 5 above is enough for completeness. Here is the informal explanation of why — see (Ref. 9) for more details. Recall that a remainder literal L from an instance C' of an input clause C needs to be added to the current context only if C' is falsified by that context. Now, if L is eventually added to the context as the current branch grows, C' will be satisfied, and so no further remainder literals from it need be considered. If L is never selected for addition, it is because at some point another remainder literal of C' becomes satisfied, L becomes satisfied or L becomes contradictory with another literal K in the current context — and stays so in every extension of that context. In the first two cases, C' is again satisfied by the current context. In the third case, it is possible to show that when adding K to the context, the system will be able to compute a context unifier from C with a remainder that differs from the original one only for the absence of L from it. This shorter remainder, when non-empty, will provide a new candidate literal different from L.

A high-level pseudocode description of the proof procedure is provided in Figure 1. For simplicity, we describe a non-restarting (unfair) recursive version of the procedure implementing naïve chronological backtracking, which does not impose any complexity bound on candidate literals.

When it terminates, the procedure either returns a set of literals, representing the most recent context and denoting a model of the input clause set, or raises the exception `CLOSED`, to denote that the clause set is unsatisfiable. In the backjumping version, the exception `CLOSED` would also carry dependency information that allows the procedure to skip certain choice points. In principle, the computation of new candidates and the application of the Subsume, Resolve, and Compact simplification rules (lines 14-16 in the pseudocode) could be done in any order. We chose the given order because, first, the computation of new candidates may trigger Close and thus avoid the application of the simplification rules, and second, it better reflects our implementation, which requires that new candidates are computed before Resolve is applied.

Darwin

```
1  function darwin Φ
2    // input: a clause set Φ
3    // output: either "unsatisfiable"
4    //          or a set of literals encoding a model of Φ
5    let Λ = ∅   // set of literals
6    let L = ¬v  // (pseudo) literal providing default interpretation
7    let CS = set of assert literals consisting of the unit clauses in Φ
8                  // the initial candidate set
9    try me(Φ,Λ,L,CS)
10   catch CLOSED -> "unsatisfiable"
11
12 function me(Φ,Λ,L,CS)
13   // L is to be added to the context
14   let CS' = add_new_candidates(Φ,Λ,L,CS)
15   let Φ' = Φ simplified by Subsume and Resolve with L
16   let Λ' = (Λ simplified by Compact with L) ∪ {L}
17   if there is no candidate applicable for Assert or Split ∈ CS' then
18     Λ'      // Λ' encodes a model of Φ'
19   else
20     let K = select_best(CS',Λ')
21     if K is an assert literal then
22       me(Φ',Λ',K,CS' \ {K})          // assert K
23     else
24       try
25         me(Φ',Λ',K,CS' \ {K})        // left split on K
26       catch CLOSED ->
27         me(Φ',Λ',K^sko,CS' \ {K})    // right split on K
28
29 function add_new_candidates(Φ,Λ,L,CS)
30   adds to CS all assert literals from context unifiers involving L
31   and one split literal from each remainder of a context unifier involving L
32   raises the exception CLOSED if it finds a closing context unifier
33
34 function select_best(CS,Λ)
35   returns the best applicable assert or split literal in CS
```

Fig. 1. *Darwin*'s proof procedure as pseudo code.

The following example is intended to demonstrate the working of the proof procedure.

Example 3.1. Let Φ be the following clause set.

$$p(x,a) \vee s(a) \quad (1)$$
$$q(x,y) \vee q(y,x) \quad (2)$$
$$r(f(x,y)) \vee \neg p(x,y) \quad (3)$$
$$\neg p(a,a) \vee \neg q(x,y) \vee \neg r(f(a,y)) \quad (4)$$

After initializing its variables Λ and L, the proof procedure in Figure 1 first determines an initial set of candidates CS. Because Φ contains no unit clause, CS is the empty set and the function me is called as $me(\Phi, \emptyset, \neg v, \emptyset)$.

The set of new candidates CS' determined then consists of the two split literals $p(x,a)$ and $q(u,v)$. They respectively originate from clause 1 and from clause 2, each paired with two disjoint variants of $\neg v$.

Simplification on Φ has no effect, and so Φ' is the same as Φ. The current context Λ' becomes $\{\neg v\}$. Because CS' contains applicable candidates, line 20 is reached, and the selection heuristics chooses $p(x,a)$ as the literal K to consider for the next inference step — the literal $p(x,a)$ is preferred over the other split literal, $q(u,v)$, because it is universal, while $q(u,v)$ is not; cf. Section 4.8 for details. Because $p(x,a)$ is a split literal, line 25 is reached, which results in the call $me(\Phi, \{\neg v\}, p(x,a), \{q(u,v)\})$. In this call, the new assert candidate $r(f(x,a))$ is determined (from $p(x,a)$ and clause 3) and thus added to the given candidate set, yielding $CS' = \{r(f(x,a)), q(u,v)\}$. This time, simplification does have an effect: with the given literal $p(x,a)$, which belongs to the current context as noted on line 7, clause 1 is subsumed, and the first literal of clause 4 is resolved away. The new clause set Φ' thus is

$$q(x,y) \vee q(y,x) \quad (2)$$
$$r(f(x,y)) \vee \neg p(x,y) \quad (3)$$
$$\neg q(x,y) \vee \neg r(f(a,y)) \quad (4')$$

Next, $p(x,a)$ is moved to the current context, yielding $\Lambda' = \{\neg v, p(x,a)\}$. The execution of the pseudocode reaches line 20, and among the current candidates $CS' = \{r(f(x,a)), q(u,v)\}$ the literal $r(f(x,a))$ is selected by the heuristics for further processing — see again Section 4.8. Since $r(f(x,a))$ is an assert literal, line 22 is reached and $me(\Phi, \{\neg v, p(x,a)\}, r(f(x,a)), \{q(u,v)\})$ is called. On execution, the newly asserted literal $r(f(x,a))$ together with the clause 4' gives rise to the new assert candidate $\neg q(x,a)$. Notice that in the underlying Assert rule application the context literal $r(f(x,a))$ gets instantiated to $r(f(a,a))$.[h] Now, $\neg q(x,a)$ is chosen to be asserted, and the next call thus is $me(\Phi, \{\neg v, p(x,a), r(f(x,a))\}, \neg q(x,a), \{q(u,v)\})$. Because for the context $\{\neg v, p(x,a), r(f(x,a)), \neg q(x,a)\}$ a closing context unifier

[h]With a parametric literal like $r(f(u,a))$ instead, $\neg q(x,a)$ could not be derived as an assert candidate.

exists (using clause 2), the exception `CLOSED` is raised. Notice that the parametric literal $p(u, v)$ from the set of candidate literals was never chosen to derive this closed branch.

The exception raised is caught by the first recursive call of *me*. Its execution thus reaches line 27 and tries the right alternative of that Split application. Because the split literal was $p(x, a)$ the corresponding call to *me* uses the complement of the Skolemized version of $p(x, a)$, say, $\neg p(c, a)$. On the execution of $me(\Phi, \{\neg v\}, \neg p(c, a), \{q(u, v)\})$, the new assert candidate $s(a)$ is derived from $\neg p(c, a)$ and clause (1). It will indeed be asserted, and for the next call to *me* only one candidate will be available, which is the split literal $q(u, v)$. After choosing it and calling *me* again no more candidate can be determined. The execution of *me* thus terminates and returns the context $\{\neg v, \neg p(c, a), s(a), p(u, v)\}$ to indicate satisfiability of the given clause set.

This context contains a literal with a Skolem constant: $\neg p(c, a)$. Since c is not part of the the input signature $\Sigma = \{p, s, q, r, f, a\}$, the literal $\neg p(c, a)$ can be in fact removed, and the resulting context $\{\neg v, s(a), p(u, v)\}$ will still describe a (Σ-)model of the input clause set.[i] *Darwin* reports the latter context instead of the previous one because it is more informative to the user.

4. Implementation

The description of the proof procedure in the previous section omits most implementation details and also leaves room for certain improvements. We provide some of these details as implemented in *Darwin* next, generally focusing more on those that are significant for its performance.

4.1. *Programming Language*

Darwin is implemented in *OCaml*,[j] a fast, strongly-typed functional language based on ML. OCaml, and thus *Darwin*, is available for several Unix-like operating systems including Linux and Mac OS X, and for the Windows family. OCaml has previously been successfully used for the implementation of the theorem prover KRHyper[10] at the University of Koblenz-Landau and for the solver ICS[11] at SRI International, among others.

Though the programming background of the second author, the main developer of *Darwin*, was mostly in OO-style C++, he quickly enjoyed using OCaml. Among other things OCaml's strong-typing, garbage collection, extremely short compile times, and informative news group made up for the paradigm shift. At the current stage of development we find that the higher level of abstraction provided by

[i] It follows from the way an interpretation is associated to a context[4] that the truth value of a Σ-literal is independent from the non-Σ-literals in a context, which are precisely those containing Skolem constants.

[j] See `http://caml.inria.fr/`.

OCaml constructs — and thus the better readability and maintainability of the code, compared to, e.g., C — amply compensate for possible performance losses when compared to implementations in lower level languages such as C.

4.2. *Term Database*

During the derivation tens of thousands of terms might exist at the same time, easily consuming hundreds of megabytes of memory. The same term or subterm might be used in many places, occurring for instance in different context unifiers or candidate literals. Many of these terms are dropped soon after creation, e.g., in backtracking or when a newly computed remainder is detected to be non-productive, causing a lot of garbage collection. To reduce the high memory consumption *Darwin* uses a database technique similar to the ones used for instance in Otter,[12] Vampire,[13] and E.[14]

This technique ensures that each (sub)term physically exists at most once in the system, i.e., all occurrences of the same term are references to the same physical instance, leading to perfect (sub)term sharing.

Terms are represented as tree-like data-structures. Building a term is done by creating a tree where the root node consists of a function or predicate symbol and its children nodes consist of subterms. Terms are managed by the *term database*. Term creation is done solely inside the database, all other parts of the system request terms from the database (by passing the function/predicate symbol and the subterms), but never directly create them. If a requested term is already contained in the database (a reference to) it is simply returned. Otherwise, the term is transparently created and then returned.[k]

Compared to a naïve representation of terms, this allows for vastly reduced memory consumption because of the perfect sharing of the common subterms. Terms are stored in a normalized manner in the database. This leads, for instance, to a unique representation of remainder literals. The significance of this property lies in the fact that remainder literals are the *only* kind of literals created by the system during proof search.

Internally, the terms are stored in a set of weak references. Weak references are ignored by the garbage collector. Thus, as long as a term is used and referenced anywhere in the system from outside the database it is kept alive. But as soon as the only remaining references to the term are from inside the database the term becomes automatically available for garbage collection, as it is considered to be unreferenced and thus disposable.[l]

The consultation of the database for each term creation and the management of the weak set introduces noticeable overhead. However, the retrieval of a term from the database is done by means of an efficient hashing on the terms, we gain the

[k]This technique is sometimes called *hash consing* in the literature.

[l]We are thankful to one of the reviewers of an earlier version of this paper[15] who suggested the use of OCaml's weak references to us.

reduction of the term equality test to a constant-time pointer equality test, and we save in garbage collection. Thus, besides the memory savings, the term database even leads on average to a slight performance improvement.

Our tests have shown that as a percentage of the total, the space required to store the clause sets along the current branch, which is essentially constant during the computation, is insignificant. Also small is the space needed for contexts, which rarely grow to more than a few thousand literals. The data structures most responsible for the memory consumption are the set of partial context unifiers, described later, and the set of candidate literals. Candidate literals consistently take most of the space. In contrast, partial context unifiers can take from almost no memory to as much memory as the candidate set.

4.3. *Backjumping and Dynamic Backtracking*

The simplest backtracking strategy for a search procedure is (naïve) chronological backtracking, which backtracks to the most recent choice point in the current branch of the search tree. A more effective form of chronological backtracking, implemented instead in *Darwin*, is *backjumping*, which takes into account dependencies between choice points. The idea of backjumping is best explained in terms of the calculus: suppose the derivation subtree below a left node introduced by a Split rule application is closed *and* the literal added on the left conclusion by that application is not needed to establish that the subtree is closed. Then, the Split rule application can be viewed as not being carried out at all. The proof procedure thus may neglect the corresponding choice point on backtracking and proceed to the previous one.

Backjumping is well known to be one of the most effective improvements for propositional SAT solvers. Its implementation is not too difficult and is based on keeping track of which context literals and clauses are involved in particular in Assert and Close rule applications. Backjumping is an example of a successful propositional technique that directly lifts to the proof procedure of *Darwin*.

A smarter technique than backjumping has been proposed under the name of *dynamic backtracking* by Ginsberg.[16] It can be adapted to our proof procedure and is currently implemented in *Darwin* as an alternative to backjumping. The idea is that a choice point not involved in establishing that a branch is closed is not discarded as in backjumping, but it is kept if it does not depend on any discarded choice points. Conceptually, the choice points are no longer seen as nodes in a tree but as nodes of a dependency graph. Discarding a choice point does not automatically invalidate all later choice points in a branch, only those dependent on it. Thus dropping and possibly recomputing a still valid and potentially useful part of the derivation is avoided.

A disadvantage of dynamic backtracking versus backjumping is that its implementation is more involved and requires a more complex type of dependency analysis. Furthermore, some implementation optimizations based on the assumption that on backtracking only the most recently added context literals are retracted are not

possible anymore. This causes non-negligible runtime overhead, leading in general to worse performance than backjumping despite the fact that dynamic backtracking sometimes produces shorter derivations.

4.4. *Iterative Deepening over Term Depth*

Since the $\mathcal{ME}$ calculus is refutationally complete only for *fair* derivations, a proof procedure for it is complete only if it gives rise to fair derivations. The completeness of *Darwin*'s proof procedure is ensured by performing a sort of iterative deepening search. In contrast to standard iterative deepening, however, *Darwin*'s iterative deepening is not on the depth of the search tree but of the *term depth* of the candidate literals.[m]

Specifically, the proof procedure never adds to the context a literal whose term depth exceeds a current term depth bound. Note that by the design of the inference rules, it is impossible for a context to contain two or more p-variants of the same literal. This implies the termination of any exhaustive sequence of inference rule applications under the term depth bound. Thus, when all inference rules have been applied exhaustively with respect to the current bound without closing the current branch, the proof procedure has to check if the current branch is *incomplete*, that is, if during the generation of the branch a candidate literal has been ignored because it exceeded the current depth bound. If the current path is not incomplete, a model of the input set has been found and is reported. Otherwise, the procedure behaves according to one of the following strategies, as initially selected by the user.

(1) **Eager Restart.** Any candidate literals that exceed the depth bound are dropped immediately. After producing an incomplete branch, the procedure simply restarts from scratch, but with an increased depth bound.
(2) **Deferred Restart.** As with the previous strategy, none of the candidate literals that exceed the depth bound are kept around. However, this time the procedure does not immediately restart on discovering an incomplete branch. Instead, it treats the branch as if it was closed and backtracks to the most recent choice point up to which no candidates had been dropped due to the depth limit, and continues the exploration of the remaining search space from there. If the procedure then generates an exhausted branch which is neither closed *nor* incomplete, it simply ends, reporting a model for the input clause set. Otherwise, it continues as before until it has explored the whole search space. Only at that point, when all remaining branches have been found to be either closed or incomplete, does the procedure restart, with an increased depth bound.
(3) **Reluctant Restart.** This strategy stresses the search for models even more by not immediately dropping candidates exceeding the depth bound, but instead saving them until the current branch is determined to be incomplete. Then, these candidates are tested against the context for applicability, as they

[m]By term depth we mean the depth of a literal when seen as a tree.

might well be not applicable anymore — because for instance they are now being subsumed by the shallower context literals. If *none* of them is applicable anymore, the branch can be considered complete and so a model is reported. Otherwise, the procedure considers those candidates that are still applicable, and backtracks to the most recent choice point where one of these still applicable candidates had been computed. Then it continues as in Strategy 2.

By eagerly dropping candidates exceeding the depth bound, Strategy 1 and 2 significantly decrease the memory requirements for a range of problems which have a refutation or a model using only comparatively shallow terms, but have lots of candidates with deeper terms.

Strategy 2 is less suited for unsatisfiable problems, as the futile search for a model is prolonged even further by deferring the restart. This may pay off instead for satisfiable problems, as the search for a model within a low depth limit is pursued more intensely, possibly avoiding unnecessary restarts.

Considering only the still applicable candidates often allows Strategy 3 to determine a more recent choice point than Strategy 2 would — the information on whether candidates are still applicable once a branch is complete is simply not available to Strategy 2. Compared with Strategy 2, Strategy 3 loses the memory savings achieved by eagerly dropping candidates, and suffers the overhead of repeatedly checking exceeding candidates for applicability. On the other hand, it further increases the likelihood of finding a model within a low depth limit.[n] Since in the current implementation Strategy 3 has a considerably worse performance than the previous two strategies, its use seems sensible only if finding shallow models is a top priority.

Another considerable advantage of Strategy 2 and Strategy 3 is that they potentially diverge less often on satisfiable problems. Strategy 3 can be further modified to consider as non-applicable any candidates that are satisfied by the context. Thus, by exploring all possible branches within the term depth bound (modulo pruning due to intelligent backtracking), the proof procedure is guaranteed to find a finite exhausted branch, if one exists, and to return the corresponding model. With the previous strategies on the other hand, the procedure can diverge even in the presence of a finite exhausted branch because it can end up (incrementally) expanding the *same* infinite branch. However, experiments showed that this variation merely adds significant computational overhead, but does not produce better derivations.

All the three restart strategies above can be formally shown to generate fair derivations. Intuitively, the idea is simply that (i) all applicable candidate literals within the current depth bound are guaranteed to be added to the context, and (ii) for each candidate literal whose depth exceeds the current bound, there is a later iteration of the proof procedure in which either a model is found or the candidate's depth is within the current bound.

[n] Another potential advantage of the strategy is that keeping discarded literals for a branch sometimes allows the proof procedure to close the branch earlier. See Section 4.10 for details.

Currently, no information from a previous round is kept after a restart. A valuable improvement of *Darwin*, with any of the three restart strategies we described, might be to compute permanent lemma clauses as a side effect of derivations, as commonly done in SAT solvers, and keep them across restarts. As in the propositional case, the idea is that the kept lemmas would help cutting the search space by preventing later repetitions of certain sets of (Split) choices that are guaranteed to lead to a closed branch.

Another potential improvement would be to use Strategy 1, but to avoid restarting altogether and instead keep growing the current branch under an increased term depth bound. Unfortunately, due to the way context unifiers are now computed (See Sec. 4.7) there is no easy way to recompute *only* the dropped candidates. Thus, either all candidates have to be kept until needed, even if they exceed the depth bound, or all candidates have to be recomputed after an increase of the depth bound. A general drawback of this approach compared to Strategy 1 is that it tends to produce significantly longer derivations because the candidate selection heuristics cannot consider from the beginning the candidates that become available only after the increase in the depth bound. Thus, although at first glance saving on computation by reusing the existing derivation seems worthwhile, in our experience with *Darwin* this helps only very rarely in practice. In most cases simply dropping candidates, restarting, and applying the heuristics to all candidates during the whole derivation leads to considerably better performance. This conforms to what can be expected from an iterative deepening approach.

Alternative measures of literal complexity than the term depth could be considered as well. For instance, the hyper tableau prover KRHyper[10] uses iterative deepening over term *weights*, computed as the number of symbols in a term. The resolution prover Otter[12] offers sophisticated control facilities to weigh a term. *Darwin* supports iterative deepening over fixed term weights as an alternative to the term depth approach. As we show in Section 5 however, using the term depth bound is usually better.

Some other provers limit the derivation tree length, i.e. the maximum length of a derivation branch. We have not tried this strategy in *Darwin* yet. So there is considerable room for further experimentation.

4.5. *Initial Default Interpretation*

As mentioned in Section 2.3, the pseudo-literal $\neg v$ that constitutes the initial context assigns by default false to all ground atoms. Instead of $\neg v$, the pseudo-literal v may be used, assigning true to all ground atoms. It is indeed often plausible to take v, given that many theorem proving benchmarks consist of an "axiom part", and a "theorem part" which quite often consists of one or more negative clauses. These theorem clauses are falsified in the interpretation associated with the pseudo-literal v. Now, the calculus considers for Split rule applications only clause instances that are falsified in the current interpretation. This means that then theorems are used

early in the derivation, de-emphasizing, in particular, the use of positive clauses from the axiom part. This way, the calculus becomes more goal-oriented than it would be with $\{\neg v\}$ as the initial context.

Nevertheless, using v has several drawbacks. First, problems are often specified in a rule-like way, i.e. $a_0, \ldots, a_n \leftarrow b_0, \ldots, b_m$ where in general $n < m$. As $\neg v$ unifies with each a_i, whereas v unifies with each b_i, v yields considerably more partial and complete context unifiers (see Section 4.7). Thus, significantly more time is spent in context unifier computation, and many more candidates are computed, leading to a (sometimes vast) increase in memory usage. Secondly, $\neg v$ is more suited for developing models that can be described with a relatively small set of atoms, whereas v is more suited for models that satisfy most of the ground atoms. The former case seems to be more common for the problems tested so far.

All in all, the performance and memory consumption on TPTP problems is in general much better with $\neg v$ than with v.[15]

4.6. *Unification*

Unification operations in theorem provers often require that the participating literals have no variables in common. For *Darwin* this is in particular the case when context unifiers are computed, i.e. when a clause is unified with fresh variants of context literals. Renaming variables by physically creating a new term is expensive in terms of memory and performance. There are several methods in use to avoid this.

For instance, SPASS does not explicitly rename common variables, but instead uses a modified unification algorithm and computes different substitutions for each participating clause or literal.[17] Otter and KRHyper use so called contexts — not to be confused with contexts in the sense of $\mathcal{ME}$ and *Darwin*. A compile time limit is imposed on the number of variables per term, e.g. 64 variables per term in the case of KRHyper. A variable is represented by a number lower than the limit. A context defines a multiplier, a number unique to this context. For the purpose of unification each literal, resp. clause, is associated with its own context. During unification a variable is identified by its effective id which is computed as the limit multiplied by the associated context multiplier, plus the variable's id.[o]

Darwin extends this idea avoiding the compile time limitation.[p] Again, a variable is represented by a number, and for unification each literal is associated with a second number, here called *offset* instead of context multiplier. Now, the effective id of a variable is not computed as a number but is simply the pair of the offset and the variable's id. For example, if the clause $p(x) \vee p(f(x))$ is unified with two variants of the context literal $\neg p(u)$, the offset 0 may be associated with the clause, and the offsets 1 resp. 2 with the two occurrences of the context literal. Then the pairs $0{:}p(x)$ and $1{:}\neg p(u)$ are unified, and the pairs $0{:}p(f(x))$ and $2{:}\neg p(u)$ are unified,

[o]For details see *unify.c* of Otter's source, resp. *term.ml* of KRHyper's source.
[p]Like similar compile time limitations this is a serious problem when working with a closed source application and still an inconvenience with an open source one.

yielding the unifier $\{0{:}x \mapsto 1{:}u,\ 2{:}u \mapsto 0{:}f(1{:}u)\}$ where $1{:}u$ and $2{:}u$ are in fact two different variables.

4.7. *Context Unifiers*

Recall that Step 3 of *Darwin*'s proof procedure computes all possible context unifiers involving the context literal just added. To be precise, the system computes context unifiers of input clauses in order to identify literals that can be added to the context by the Split rule, and computes context unifiers of subsets of input clauses in order to identify literals that can be added by the Assert rule. To speed up this computation, context unifiers are partially precomputed and cached as described below. For simplicity, we start by describing the computation of context unifier for Split only. Figure 2 illustrates this process and its embedding in the proof procedure.

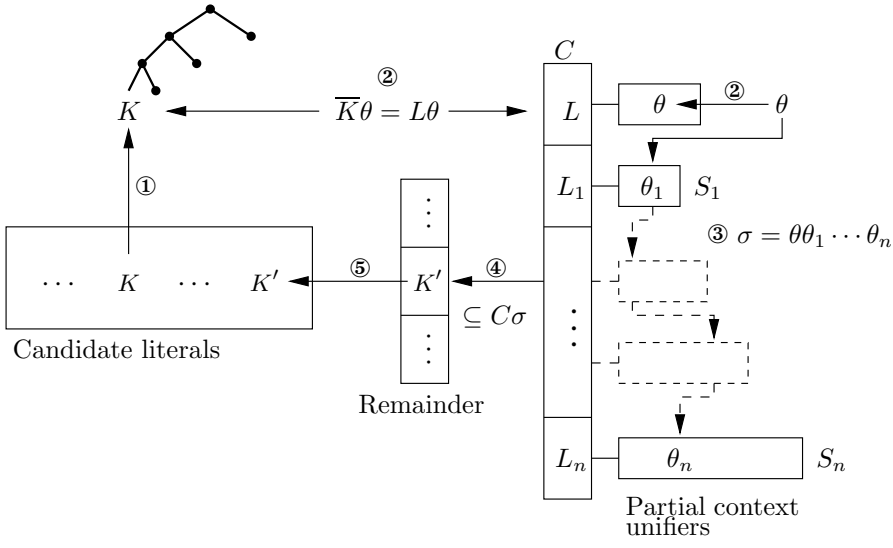

Fig. 2. Computation of context unifiers and its embedding in the proof procedure.

Each *input* literal — i.e., a literal in the input clause set — has an associated list of *partial context unifiers*. A partial context unifier is merely a unifier between the input literal and a literal from the current context. According to the unification mechanism described in Subsection 4.6, the literals of a clause are numbered starting with 1. During unification, the offset 0 is assigned to the clause and thus the input literal, while the input literal's number in the clause is used as the offset of the context literal. This makes it easy to merge several partial context unifiers between different literals of a clause and the same context literal. Note that the computation of partial context unifiers does not happen per clause but per input literal, thus if a

literal occurs at the same position in several clauses its partial context unifier will be computed and stored only once for all occurrences.

The bindings of the stored partial context unifiers are kept in a database similar to the term database. Especially for Bernays-Schönfinkel and some Horn problems, where mostly very similar terms are computed, the unifiers tend to share most bindings. Using the database leads to significant memory savings.

When a new literal K is selected for addition to the context (Step (1) in the proof procedure, Step ① in Figure 2), the system computes all partial context unifiers between (a fresh variant of) $\overline{K}$ and each literal in the current clause set. Then it stores each computed unifier, when it exists, in the list of the corresponding input literal. This is depicted in Figure 2 as Step ②, however for only one input literal. After that, for each literal L that unifies with $\overline{K}$ and for each input clause C containing L, the system attempts to find all possible context unifiers of C against the current context. This is done as follows.

Assume that C is of the form $L \vee L_1 \vee \cdots \vee L_n$, θ is the partial context unifier between L and $\overline{K}$, and S_i is the set of partial context unifiers stored in L_i's list for $i = 1, \ldots, n$. Then the system considers each tuple of partial unifiers in $\{\theta\} \times S_1 \times \cdots \times S_n$ and attempts to merge the elements of that tuple into a single unifier (Step ③ in Figure 2). When the merge succeeds, the resulting substitution is a context unifier of C against the current context.[q]

To minimize recomputation, the merged unifiers are computed incrementally by traversing the partial context unifier lists for the clause C in a depth-first fashion. The root node of the depth-first traversal is θ, its children are all the partial context unifiers of L_1, the children of each of the root's children are all the partial context unifiers of L_2, and so on. Partial context unifiers are merged incrementally as they are visited along a path of this search tree, and the merged unifier computed along a path is reused for all the extensions of that path. Clearly, less work is done if the tree is slim at the root, as less merge operations are then necessary. To achieve this the lists associated with the literals $L_1, \ldots, L_n$ in C are actually first ordered by increasing length before starting the traversal. This is indicated in Figure 2 by boxes of growing length for S_1 to S_n in this order.

Each newly computed context unifier determines a remainder (Step ④ in Figure 2), and every such (non-empty) remainder provides one new candidate literal, selected as explained in Section 4.8, that gets added to the candidate set in Step 5 of the proof procedure[r] (Step ⑤ in Figure 2, where the new candidate literal is denoted as K').

For each candidate literal L, the system maintains a reference to the input clause and the context literals used to compute the context unifier and the remainder

[q]The context unifier is converted into an admissible context unifier afterwards. The underlying algorithm is illustrated in Example 2.1 and explained in detail in (Ref. 9). But we can ignore this issue here.

[r]Recall from Section 3 that it is enough to consider only one Split literal per remainder without affecting the calculus' completeness.

that contained L. This allows it to quickly recompute the context unifier and the remainder later, if needed.[s]

The computation of Assert candidates is done in a similar way thanks to the following expedient. Recall that to apply Assert to a clause $L \vee C$ it is necessary to compute a context unifier for the subclause C only. To do that we use a fictitious context literal, let us call it *Assert* here, that unifies with any input literal but can be used only once in the computation of each context unifier.[t] After the context unifier has been computed, the result of applying that unifier to the clause literal that had been paired with *Assert* is a possible Assert candidate. This way, Assert and Split candidates can be computed at the same time using the very same incremental algorithm described earlier for merging partial context unifiers. This leads in practice to significant performance gains with respect to computing Assert and Split candidates separately.

At times, it is convenient *not* to interleave the computation of Assert and Split candidates. In fact, since Assert candidates are always preferred to Split candidates by the selection heuristics, it makes sense to delay the computation of Split candidates as long as Assert candidates exist. Such delaying pays off when a derivation branch can be closed by a sequence of Assert applications, and in particular when a problem can be proven unsatisfiable by means of Assert alone, which is the case for instance for Horn input sets. Thus, the current version of *Darwin* starts by computing only Assert candidates until a Split is really needed. From that point on Split and Assert candidates are computed together, as described above. This simple combination yields in general a better performance than either computing Split and Assert candidates separately or interleaving their computation from the beginning.

It is interesting to point out hat contrary to what is done in *Darwin*, the $\mathcal{ME}$ calculus does not require the computation of *all* the possible context unifiers involving a given context literal: the Split inference rule (and similarly Assert) admits implementations that compute remainders only *locally*, during the Split rule application, and discard them afterwards. Thus, for a given context, the possible context unifiers of a clause could be computed, say, one after the other until an admissible one is found. At this point Split could be applied using that unifier and the unifier could then be immediately discarded. Memory consumption under such a scheme would be obviously greatly reduced.

Nevertheless, the approach used in *Darwin* has a big advantage: because at any point in the derivation all the theoretically necessary context unifiers and their remainders are known, they are available for inspection and comparison. Given that both the choice of a remainder from the set of all possible (admissible) remainders,

[s]In an earlier version of *Darwin*, described in (Ref. 15), more information was kept for the "best" candidates, called *active* candidates, such as for example the complete remainder. The separation into active and *passive* candidates has been dropped as it merely increased the complexity of the system without improving its performance.

[t]So, for instance, with the clause $L \vee C$ above, if paired with L, *Assert* cannot be paired with any literal in C.

and the choice of a literal from it to split with are *don't-care* nondeterministic choices, arbitrary heuristics can be employed for their computation. Furthermore, for each pairing of an input literal L with a context literal, the computation of the context unifier for the clause containing L is attempted in *Darwin exactly once* in the current derivation tree branch. This avoids the recomputation of the same context unifier that would happen in the more naïve scheme indicated above.

These considerations are the main rationale for the data structures and algorithms described above. Computing context unifiers incrementally and non-locally is clearly memory intensive, as all possible partial unifiers are kept in memory. However, all of our experiments so far indicate that the current levels of memory consumption does not impair the performance of the system. But more experimental results explicitly monitoring memory consumption are probably needed.

4.8. *Selection Heuristics*

As explained in the previous section, all theoretically necessary remainders are at any point in the derivation available for inspection. This supports the effortless implementation of heuristics to select a literal to split with. The heuristics for selecting a literal from the candidate set to be added to the context is based on the following criteria. The overall heuristics is determined by the induced lexicographic ordering over these criteria, with **Universality** being the most significant criterion, and **Generation** the least significant one. Assert is always preferred over Split in order to emphasize redundancy elimination.

(1) **Universality.** Universal Split literals (which includes ground literals as well) are preferred to parametric Split literals as the addition of universal literals impose stronger constraints on the current context, generally leading to context unifiers with smaller remainders and to fewer applicable Split candidates later.
(2) **Remainder Size.** Recall that candidate literals for Split are drawn from the remainder of some context unifier. Now, if the problem is satisfiable, at least one remainder literal of every remainder must be satisfied by the context. Because of this, candidate literals originating from smaller remainders are preferred over literals from larger remainders. The rationale is that backtracking is minimized this way. For an extreme case, note that for Split literals coming from a singleton remainder applying the right side of Split is pointless because it immediately produces a closed branch. In fact, *Darwin* does not even generate a choice point when it adds such literals to the context.
(3) **Term Weight.** The number of symbols in a literal has shown to be useful information that should be exploited. This emphasizes the use of "lighter" literals. Because variables are excluded from counting, additional preference is given to literals with variables instead of parameters or other terms at the variable positions.
(4) **Generation.** This is a measure of how close in the derivation the candidate is to the original clause set. The generation of a context literal is -1 for $\neg v$, and

the generation of the corresponding candidate otherwise. The generation of a candidate is the maximum of the generations of the context literals used in its context unifier incremented by one. That is, candidates whose context unifier is solely based on $\neg v$ are of generation 0.
Candidates with a smaller generation are preferred. The intention is to keep the derivation close to the problem set, similar to bidirectional search. For some problems this is the key to their solutions; on average it is a slight improvement.

Recall that the term depth is not needed as part of the heuristics as it is implicitly imposed by the depth bound (see Section 4.4).

The lexicographic ordering on candidate literals induced by the criteria above is not total. To simplify debugging and the comparison of different data structures, we make it total by using as last ordering criterion an arbitrary, but fixed, total ordering on the candidates based on an enumeration of the input clauses and of the context literals.

Criteria 1 and 3 above are also applied in Step 5 of the proof procedure when choosing a literal from a remainder as the candidate literal for that remainder.

4.9. *Term Indexing*

The current context is basically a set of literals. The preconditions of Split, Assert, and Subsume require, in essence, to search the context for literals that unify with, subsume, or are subsumed by a given literal. Some of these queries are applied to every computed candidate at least once in order to immediately drop invalid, e.g. subsumed, candidates. In order to avoid a linear scan of the context to perform each of these checks, *Darwin* uses by default term indexing for the context based on *substitution trees*.[18]

Substitution trees index terms by abstracting over identical subterms. For example, the terms $f(g(a))$ and $f(g(b))$ are represented by a node containing $f(g(x))$ and two children containing the substitutions $\{x \mapsto a\}$ and $\{x \mapsto b\}$. Thus, for the term $f(h(a))$ the non-unifiability is detected at the node $f(g(x))$ for both children. In general, substitution trees seem to be best suited for deep terms containing variables. For shallow ground terms, e.g., for clause sets stemming from Bernays-Schönfinkel problems, using the current implementation of substitution trees makes *Darwin* actually slower than using no term indexing at all.

For comparison, an alternative indexing scheme based on imperfect *discrimination trees*[19] has been implemented. Discrimination trees index on common term prefixes, where a term is seen as a sequence of symbols given by its pre-order traversal, and for imperfect discrimination trees all variables are represented by the same special constant. In the current implementation, their performance is quite close to that of substitution trees for non-Horn problems and slightly superior for Horn problems. As the crucial productivity check (Ref. 8) is currently only implemented for substitution trees, those are still the preferred choice for non-Horn problems.

4.10. Close *Look-ahead*

A branch is detected as unsatisfiable as soon as Close applies, which happens when a context unifier with an empty remainder is computed for a clause in the current clause set. It is easy to see however that when in the current branch of the derivation two contradictory Assert candidates are computed, asserting one of them would immediately close the branch. Now, due to the fact that candidate literals wait for their turn in the candidate set, in unlucky cases two contradictory candidates might be ignored for a long time. To avoid this problem, Assert candidates and Split candidates from remainders containing only one literal are stored in a term index (Section 4.9). Each new candidate is checked against the index for a contradiction. As soon as this check succeeds Close can be triggered by adding the new candidate to the context.

If all candidates were stored in the look-ahead index the introduced overhead of maintenance and contradiction checks would in general far outweigh the benefits of a shortened derivation. Empirically, an index size of about 10,000 for candidates obeying the current term depth bound has turned out to introduce only moderate overhead while making problems solvable which were previously out of reach. It is not clear yet if including the candidates exceeding the depth bound in the look-ahead mechanism as well is worthwhile. This adds such significant overhead in general that some problems are no longer solved in a reasonable amount of time.[u] On the other hand, some previously unsolvable problems become instead solvable.

4.11. *Horn Problems*

The $\mathcal{ME}$ calculus allows for several optimizations for the important class of Horn problems. First, $\mathcal{ME}$ is complete for Horn problems without the Split rule, i.e., extending the context by means of Assert alone is sufficient.[9] Therefore, with Horn clause sets, *Darwin* does not compute any Split context unifiers. Second, negative Assert literals can also be ignored without losing completeness if the default interpretation is $\{\neg v\}$. Thus, the current context is only extended by asserting positive literals. For many problems this saves time, by avoiding the fruitless assertion of negative literals, which are satisfied by the context $\{\neg v\}$ anyway, and the subsequent possible computation of further unnecessary Assert candidates. Note that while never asserted, negative Assert candidates do still play a role as they are used for the Close look-ahead (See Section 4.10).

5. Performance Evaluation

We evaluated the performance of *Darwin* against the TPTP problem library version 2.7.[v] Since *Darwin*'s input language covers only clause logic, and *Darwin* does not

[u] The overhead is caused mostly by index maintenance operations, as candidate literals are removed from the index when selected or during backtracking.
[v] See `http://www.tptp.org/`.

(yet) have dedicated inference rules for equality, we concentrated on the clausal problems without equality. In order to compare *Darwin* with other current provers we list the results for a subset of the last CASC competition, CASC-J2,[w] and we evaluated those provers over a subset of the TPTP. All tests were run on a Pentium IV 2.4Ghz computer with 512MB of RAM. The imposed time limit was 300 seconds for the tests on the clausal problems of the TPTP without equality, and 500 seconds for the CASC tests; the memory limit was 500 MB in both cases. Experiments showed that this is comparable to the setup of the CASC-J2 competition, where AMD Athlon XP 2200+, 1.8Ghz computers with 512MB of RAM where used with a time limit of 600 seconds.

Table 1. Several *Darwin* configurations over the TPTPv2.7

Name	# Problems	Default	Eager Restart	Reluctant Restart	Exceeding Look-ahead	Weight Limit
HNE - SAT	63	24/0.1	–	24/0.1	24/0.1	24/0.1
HNE - UNSAT	680	602/4.9	–	597/6.5	592/4.9	602/7.8
HNE - Total	758	626/4.8	–	621/6.3	616/4.7	627/7.6
NNE - SAT	400	345/2.4	333/1.7	346/2.4	346/2.4	330/3.3
NNE - UNSAT	654	552/7.1	552/6.3	537/6.8	560/7.2	491/5.4
NNE - Total	1172	897/5.3	885/4.6	883/5.1	906/5.3	821/4.6

Note: The problem classes consist of clausal problems without equality and are divided in Horn problems ("HNE") and non-Horn problems ("NNE"), and in satisfiable ("SAT") and unsatisfiable ("UNSAT") problems. Note that some problems are classified as unknown or open instead of satisfiable or unsatisfiable, so the total numbers of problems is higher than the sum of satisfiable and unsatisfiable problems. "#Problems" gives the number of problems in a class, table entries are of the form "Number of problems solved"/"average CPU time". See text for an explanation of the different *Darwin* configurations; for an evaluation of older features see (Ref. 15) and (Ref. 9).

Table 1 summarizes the results for several configurations of *Darwin*. "Default" represents the default configuration, in which all derivation rules are used, back-jumping is used for backtracking, the initial context is $\{\neg v\}$, iterative deepening is over the term depth, the initial term depth bound is 2, Close look-ahead is performed only for candidates within the depth bound, and restarting follows the non-eager Strategy 2 described in Section 4.4.

All other configurations differ by exactly one option. Specifically, "Eager Restart" uses the restart strategy 1, "Reluctant Restart" uses the restart strategy 3, "Exceeding Look-ahead" includes candidates exceeding the depth bound in the Close look-ahead, and "Weight Limit" does iterative deepening over the term weight. Notice that for Horn problems eager restarting and the default strategy are identical, as the derivation tree degenerates to a branch (see Section 4.11), and so backtracking to another branch is not possible.

[w]See `http://www.tptp.org/CASC/J2/`

As expected, "Eager Restart" is faster for unsatisfiable problems but solves less satisfiable problems than the default strategy. The "Reluctant Restart" strategy is worst. Disappointingly, it solves only one more satisfiable problem than the default strategy. "Exceeding Look-ahead" adds a significant overhead, so that even a number of problems solved with the default configuration are lost. On the other hand, a larger number of problems which are too hard for the default configuration can be solved almost instantly, so some measure of when to apply the look-ahead is needed. For NNE problems, iterative deepening over the term weight is clearly inferior to iterative deepening over the term depth. For Horn problems, it is still much slower but it solves the same number of problems. More interestingly, the two methods differ in the actually solved problems. For example, the Horn problem PUZ050-1 is solved in less than 70 seconds with term weight, but not at all with term depth, with 73260 applications of Asserts, while computing 194762 Assert candidates. It has a rating of 1.0 and the classification "unknown", denoting that the problem is very hard for most provers.[x] This problem is also the reason that the number of satisfiable and unsatisfiable problems do not sum up to the total number of solved problems in Table 1. Combined, the two approaches solve 650 problems. All in all, the default configuration seems to be a reasonable choice, unless specific information about the problem at hand indicates another configuration.

Table 2. Several provers over the TPTPv2.7

Name	# Problems	Darwin 1.1	DCTP 1.31	DCTP 10.21p	Vampire 7.0	E 0.82	Spass 2.1
HNE - SAT	63	24/0.1	34/0.0	35/1.0	30/19.0	32/0.1	29/0.1
HNE - UNSAT	680	602/4.9	538/5.1	599/7.7	662/14.4	644/2.9	539/9.9
HNE - Total	758	626/4.8	572/4.8	634/7.4	693/14.7	676/2.8	568/9.4
NNE - SAT	400	345/2.4	290/6.2	342/10.0	176/25.4	234/6.8	220/14.4
NNE - UNSAT	654	552/7.1	510/7.3	589/7.1	635/7.5	604/6.4	521/10.5
NNE - Total	1172	897/5.3	800/6.9	931/8.2	812/11.6	838/6.5	741/11.7

Note: The structure of the table is analog to Table 1. *Darwin* was run in the default configuration, while the other provers were run with the same settings as in the CASC-J2 competition, i.e. DCTP 10.21p with the time limit set to 300. DCTP 1.31 with "-negpref -complexity -fullrewrite -alternate -resisol", Vampire 7.0 with "–mode casc-j2 -t 300", E 0.82 with "-s -xAuto -tAuto", and Spass 2.1 with "-PProblem=0 -PGiven=0 -PStatistic=0 -Auto".

In Table 2 *Darwin* is compared with state-of-the-art first-order theorem provers. Of the selected provers, DCTP implements the disconnection tableaux calculus,[20] where DCTP 1.31 uses a single strategy, while DCTP 10.21p employs several strategies and restarting. The other provers are based on saturation methods such as resolution and superposition. Regarding the total number of problems solved, *Darwin* performs in the mid range for Horn problems without equality, and is second only

[x]It is solved by Vampire 7.0 as well, though.

to the multi-strategy version of DCTP for non-Horn problems without equality, which we consider to be a very good result. Regarding the average time spent on the problems solved, the differences among the provers are not that significant.

We must mention though that about 300 of the NNE problems belong to the Bernays-Schönfinkel class, a class for which *Darwin* and DCTP are decision procedures, while the other provers are not. Interestingly, *Darwin* is the weakest prover for satisfiable Horn problems, but fares best for satisfiable non-Horn problems. In general, *Darwin* and DCTP seem to be much stronger for satisfiable non-Horn problems than the saturation provers Vampire and E, but much weaker for unsatisfiable problems. For example, *Darwin* basically instantly solves a number of hard (rating 0.86) satisfiable problems, which are too hard for Vampire, E, and DCTP 10.21p, or take a few minutes to be solved by DCTP 10.21p. One of these, SYN803-1, is found to be satisfiable within a term depth of 2 in 0.0 seconds, with 2 applications of Assert, 1 of Split, and 2 of Subsume, producing a final context consisting of merely three literals. For most unsatisfiable problems Vampire or E are faster, but for example FLD052-4 is solved in 2.3 seconds by *Darwin*, in 33.7 seconds by Vampire, and not at all by E and DCTP. It it is proven unsatisfiable by means of Assert (932 applications) alone. Unsatisfiable Bernays-Schönfinkel problems like GRP128-2.006 (8 Close, 428 Assert, 7 Split) and GRP128-3.005 are almost immediately solved by *Darwin* and DCTP, and not at all or using significantly more resources by Vampire and E.

In these comparisons one should take into consideration that *Darwin* uses only one strategy, whereas DCTP-10.21p, Vampire, and E use different strategies based on the specific problem, which helps to increase the number of problems solved.

Finally, In Table 3 *Darwin* is evaluated on some of the divisions of the CASC-J2 competition restricted to clausal problems. The actual problems used in the

Table 3. Some problem divisions of the CASC-J2 competition

Name	# Problems	Darwin 1.1	DCTP 1.31	DCTP 10.21p	Vampire 7.0	E 0.82	Otter 3.3
HNE	35	18	19	27	35	31	13
HEQ	35	2	3	8	31	31	3
EPS	40	40	37	40	9	–	–
EPT	40	38	35	39	37	–	–
NNE	35	17	12	22	34	32	3
SNE	50	19	19	25	–	–	–

Note: Problem names: HNE – (unsat.) Horn with No Equality; HEQ – (unsat.) Horn with some (but not pure) Equality; EPS – (sat.) Effectively Propositional non-theorems); EPT – (unsat.) Effectively Propositional Theorems; NNE – (unsat.) Non-Horn with No Equality; SNE - SAT with No Equality. "#Problems" gives the number of problems in a class, table entries are of the form "Number of problems solved". *Darwin* was run in the default configuration; the results of the other provers are taken from the CASC-J2 web page, "–" means that a prover did not participate in this class.

competition are randomly selected from so-called eligible TPTP problems. A problem is eligible, if additionally to meeting its division's requirements its rating classifies it as neither trivial nor too hard. Furthermore, each division must contain some previously unseen problems and must not include an excessive number of very similar problems.[y]

As expected, *Darwin* fares extremely well in the divisions EPS and EPT, which consist of satisfiable and unsatisfiable clause sets with a finite Herbrand universe, i.e., essentially the Bernays-Schönfinkel class. The only competitive prover for these problems is DCTP, especially in its multi-strategy version. As could also be expected from CASC's bias towards unsatisfiable problems, *Darwin* turns out to be very weak for Horn problems with equality, and also comparatively weak for Horn and non-Horn problems without equality, though significantly better than Otter. Note that while *Darwin* performs quite well for SNE problems when compared with the other provers listed, systems specialized in satisfiable problems such as Paradox and Gandalf are clearly superior, solving almost all SNE problems.

Updates of experimental results and more detailed information, including *Darwin*'s time and memory consumption individually for each problem, can be found on *Darwin*'s web page.[z]

6. Conclusions and Future Work

The purpose of this paper was to describe the design of the *Darwin* theorem prover, its proof procedure, data structures and algorithms. One of the main motivations for developing *Darwin*'s calculus, Model Evolution, was the possibility of migrating to the first-order level some of those very effective techniques developed by the SAT community for the DPLL procedure. This goal has been achieved to a certain degree: the current version of *Darwin* implements a first-order version of unit propagation, a form of simplification, and backjumping, a form of intelligent backtracking. These features, which are considered absolutely critical for the good performance of propositional DPLL-based SAT solvers, where the most immediately implementable given that the Model Evolution calculus itself [4,8] was already designed with them in mind.

Yet, much remains to be done. Various alternatives and modifications to *Darwin*'s data structures and algorithms have been identified in Section 4. Among these, perhaps the most significant one concerns the selection heuristics explained in Section 4.8.

We plan to to adapt to *Darwin* a few more of the heuristics that have proven useful with the propositional DPLL procedure. For instance, we are considering implementing a literal selection heuristics that prefers candidates from recent *conflict sets*, i.e., literals recently responsible for the closure of a previous branch.[21] Since

[y] See `http://www.tptp.org/CASC/J2/Design.html#Problems` .
[z] Currently at `http://goedel.cs.uiowa.edu/Darwin/` .

conflict sets are already computed in *Darwin* as they are used for backtracking (see Section 4.3), this heuristics should be quite easy to incorporate. The incorporation of another staple technique for DPLL-based solvers, lemma learning, is planned for the next version. Adding lemmas, however, will require some more theoretical work at the calculus level first.

As another extension we are currently investigating ways to equip the calculus with dedicated inference rules for (efficient) equality reasoning. When implemented, this will address the issue that *Darwin* generally performs poorly in domains with equality.

Fairness of derivations is currently achieved through iterative deepening over term depth or term weight. It would be interesting to experiment with alternatives like iterative deepening over derivation length. Different iterative deepening strategies are known to have a drastic impact on the search space exploration of model *elimination* provers,[22] and it is reasonable to expect the same for *Darwin*.

We also reported on practical experiments carried out with problems from the CADE-J2 system competition, as well as on results on parts of the TPTP problem library. When assessing the performance of *Darwin* compared to other provers, we believe one should take into account that the Model Evolution calculus is a very recent development. A great deal of know-how has been developed over the last decades for the implementation in particular of resolution and model elimination based systems. Although the techniques employed there can be partially exploited (and we tried so for *Darwin*), new algorithms and data structure tailored for the Model Evolution calculus, such as those used in *Darwin* for computing context unifiers, are probably needed. Similarly, more work is necessary to identify successful proof strategies and heuristics for the calculus. The same applies to other instance-based methods such as, e.g., the disconnection tableau calculus,[20] which presently seems to be the only calculus of this kind for which a competitive prover exists.[23] Despite a lack of established know-how, we find our first experimental results very encouraging. In particular, *Darwin* performs very well on clause sets stemming from Bernays-Schönfinkel problems. It is among the best provers for the EPS and EPT divisions of the TPTP library. More generally, it is also among the best provers over the non-equational divisions of TPTP.

Acknowledgments

We thank the anonymous referees for their insightful comments on how to improve the paper and for their valuable suggestions on how to improve *Darwin*'s implementation as well.

References

1. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
2. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

3. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In G. Ianni and S. Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence* (*Cosenza, Italy*), volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
4. P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In F. Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.
5. P. Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In D. McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.
6. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics* (*AI-MATH'96*), Fort Lauderdale (Florida USA), 1996.
7. F. Oppacher and E. Suen. HARP: A Tableau-Based Theorem Prover. *Journal of Automated Reasoning*, 4:69–100, 1988.
8. P. Baumgartner and C. Tinelli. The Model Evolution Calculus. Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.
9. A. Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau, 2004.
10. C. Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.
11. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.
12. W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, National Laboratory, Argonne, IL, 1994.
13. A. Riazonov and A. Voronkov. Vampire 1.1 (system description). In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 242–256. Springer-Verlag, 2001.
14. S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
15. P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin: A theorem prover for the model evolution calculus. In G. Sutcliffe, S. Schultz, and T. Tammet, editors, *Proceedings of the 1st Workshop on Empirically Successful First Order Reasoning* (*ESFOR'04*), *Cork, Ireland, 2004*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.
16. M. L. Ginsberg, J. M. Crawford, and D. W. Etherington. Dynamic backtracking, 1996.
17. C. Weidenbach. *The Theory of* SPASS *Version 2.0*. Max-Planck-Institut für Informatik.
18. P. Graf. Substitution tree indexing. Research Report MPI-I-94-251, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1994.
19. R. Sekar, I.V. Ramakrishnan, and A. Voronkov. *Handbook of Automated Reasoning*, volume II, chapter Term Indexing, pages 1855–1964. Elsevier Science, June 2001.
20. R. Letz and G. Stenz. Proof and Model Generation with Disconnection Tableaux. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial*

Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, volume 2250 of *Lecture Notes in Computer Science*. Springer, 2001.
21. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.
22. R. Letz and G. Stenz. Model elimination and connection tableau procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 28, pages 2017–2114. Elsevier, 2001.
23. G. Stenz. DCTP 1.2 - System Abstract. In U. Egly and C. G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings*, volume 2381 of *Lecture Notes in Computer Science*, pages 335–340. Springer, 2002.