E – A Brainiac Theorem Prover

Stephan Schulz
Institut für Informatik
Technische Universität München
Email: schulz@informatik.tu-muenchen.de

We describe the superposition-based theorem prover E. E is a sound and complete prover for clausal first order logic with equality. Important properties of the prover include strong redundancy elimination criteria, the DISCOUNT loop proof procedure, a very flexible interface for specifying search control heuristics, and an efficient inference engine. We also discuss strength and weaknesses of the system.

Keywords: theorem proving, superposition, rewriting, E, search control

1. Introduction

E is a fully automatic theorem prover for clausal logic with equality. It is a saturating prover based on a purely equational view, with a strong emphasis on rewriting. E is sound, and, unless it runs out of user-specified or physical resource limits, complete.

The prover has participated in the CADE ATP System Competitions since 1998. In 2000, it won the first place in the MIX division of CASC-17, and in 2001 achieved third places in the MIX *assurance* class, the MIX *proof* class, and the UEQ division of CASC-JC. In addition to this successes as a standalone system, it also is one of the core components of the strategy parallel prover E-SETHEO [25], which won various CASC categories in both 2000 and 2001.

We believe that the reason for these successes is a combination of good theoretical groundwork, careful engineering, and extensive experimental work. In this paper we will describe how E is constructed and which aspects of the prover are responsible for its power. There are four major elements that form the conceptual core of the prover:

Calculus: E is based on a variant of the superposition calculus [4] with literal selection. Superposition is generally recognized as one of the most powerful calculi for proof problems with equality. The major reason for this is the compatibility with a wide variety of redundancy elimination criteria. E implements most known redundancy elimination techniques, as well as some new ones. We describe the calculus in some detail in section 2.

Search Organization: The main proof procedure is based on the DISCOUNT loop, which strictly separates a small number of active clauses (used for both generating inferences and simplification) from the majority of passive clause which are only activated one by one. This allows a very good heuristic control of the proof procedure, a high inference rate even with a relatively simple engine, and easy control of memory consumption. Section 3 covers the DISCOUNT loop as used in E.

Heuristic Control: A good control of the proof search is probably the major strength of E. There are three major choice points: Clause selection, literal selection, and term ordering. E has a uniquely flexible way of selecting clauses for processing, involving an arbitrary number of differently evaluated priority queues. The selection of inference literals in a clause, which in many cases can restrict the number of necessary inferences, is less flexible. However, there are about 60 useful predefined literal selection schemes implemented in the prover. Finally, term orderings can be created according to a small number of simple weight- and precedence schemes. The major heuristic concepts used in E are described in section 4.

Inference Engine: E's inference engine is reasonably efficient. While many other provers (as e.g. Vampire [29,19] and Waldmeister [8]) are more optimized for raw throughput or subsumption rates, the inference engine in E has been specifically designed and optimized with its proof search organization in mind, and has proved fully adequate. The most distinguishing feature is the use of shared terms with shared, non-local rewriting [12]. We discuss

AI Communications
ISSN 0921-7126, IOS Press. All rights reserved

some aspects of the implementation in section 5.

We consider E to be a *brainiac*[1] prover for two reasons: First, a very large amount of work has been spend on developing and evaluating good search control heuristics. Thus, E's proof search often seems to be more intelligent than that of many other automated deduction systems. Secondly, especially in the rewriting engine, E typically substitutes a few very complex operations (shared, non-local rewriting, time-stamp constrained matching) for a large number of simpler operations.

This paper is organized as follows: After this introduction, we devote one section to each of the four major items mentioned above. We then discuss some aspects of the performance characteristics of the prover. The paper concludes with some sentences about possible future improvements.

2. Calculus

E implements the calculus **SP**, a variant of the superposition calculus $\mathcal{E}$ described in [4] with a slightly different notion of literal selection and explicit inference rules for simplification. Pure superposition (as used in E) is a refutation-based saturating calculus operating on the equational representation of formulae in clause normal form[2]. The *proof state* is represented by a set of equational clauses, which initially all come from the problem formalization. A proof derivation is a process that systematically infers new clauses and adds them to the proof state until the empty clause has been derived and thus the unsatisfiability of this set has been made explicit. It may also, optionally, simplify or remove redundant clauses from the proof state.

The major problem in automated theorem proving is the explosion of the number of clauses a prover has to consider in each step. Therefore the success of a calculus usually depends on the restrictions imposed on generating inferences and the ability to remove redundant clauses. Our prover uses a wide variety of efficiently implementable contraction techniques.

2.1. Preliminaries

We assume that the reader is familiar with basic concepts from equational theorem proving and only repeat some definitions to establish notation. See [2] for an introduction to rewriting and [5,16] for equational theorem proving. [22] covers the same material presented here in more detail.

$Term(F,V)$ denotes the set of (first order) *terms* over a set F of function symbols and set V of variables. We use s,t,u,v (possibly primed or subscripted) to denote terms, x,y,z for variables, f,g for non-constant function symbols and a,b,c for constants. We write $t|_p$ to denote the subterm of t at a position p and write $t[p \leftarrow t']$ to denote t with $t|_p$ replaced by t'. An equation $s \simeq t$ is an (implicitly symmetrical) pair of terms. A positive literal is an equation $s \simeq t$, a negative literal is a negated equation $s \not\simeq t$. We write $s \mathrel{\dot\simeq} t$ to denote an a literal of unspecified polarity, i.e. a literal that is either positive or negative. An (equational) *clause* is a multi-set of literals, sometimes written as $l_1 \vee l_2 \vee \ldots \vee l_n$ and interpreted as the disjunction of its literals. We usually consider two occurrences of the same clause as distinct objects and implicitly assume that any two clauses do not share any variable. If C is a clause, we denote by C^- the (multi-) subset of negative literals in C, and by C^+ the (multi-) subset of positive literals.

A substitution σ is a mapping from V to $Term(F,V)$ so that $\{x|\sigma(x) \neq x\}$ is finite. It is extended to terms, literals and clauses. A *ground reduction ordering* $>$ is a Noetherian partial ordering that is stable w.r.t. the term structure and substitutions and total on ground terms. An ordering $>$ can be extended to literals by comparing the multi-set representation of literals with $\gg\gg$ (the multi-set-multi-set extension of $>$), where a literal $s \simeq t$ is represented as $\{\{s\},\{t\}\}$ and $s \not\simeq t$ is represented as $\{\{s,t\}\}$. Similarly, it can be lifted to clauses by considering clauses as multi-sets of the literal multi-set representations.

Finally, if $s \simeq t$ is an equation (or a unit clause) with $s > t$, we say that $s \simeq t$ is *orientable*. Moreover, if $\sigma(s) > \sigma(t)$, we call $\sigma(s \simeq t)$ an *orientable* instance of $s \simeq t$.

[1] Note that the classical antonym to *brainiac* is *speed demon*. The terms describe different approaches to a given problem, and do not imply a value judgment.

[2] The equational representation of a non-equational atom A is the equation $A \simeq \top$, where $\top$ is a special function symbol. In order to make the naive transformation correct, it is necessary to introduce separate, disjoint *sorts* for atom terms and ordinary terms. We abstract from this details in the following and assume the pure equational case.

2.2. Generating Inferences

The superposition calculus restricts generating inferences to positions in maximal terms of maximal literals. Alternatively, it is possible to arbitrarily *select* certain literals in clauses which have at least one negative literal, and in this case to restrict inferences to selected literals. For **SP**, we use the following notions to formalize this:

A literal selection function *sel* is a function that maps a clause C to a multi-subset of C, with the property that $sel(C) \cap C^- = \emptyset$ implies that $sel(C) = \emptyset$. If $l \in sel(C)$, we say that l is selected (with respect to *sel*). The above definition then implies that if any literals are selected in a clause, then at least one of those is negative. Now assume that *sel* is a literal selection function and that $>$ is a *ground reduction ordering* that has been lifted to literals.

Let $C = l \vee R$ be a clause and σ a substitution. We say $\sigma(l)$ is *eligible for resolution* if either

- $sel(C) = \emptyset$ and $\sigma(l)$ is $>$-maximal in $\sigma(\mathcal{C})$ or
- $sel(C) \neq \emptyset$ and $\sigma(l)$ is $>$-maximal in $\sigma(sel(C) \cap C^-)$ or
- $sel(C) \neq \emptyset$ and $\sigma(l)$ is $>$-maximal in $\sigma(sel(C) \cap C^+)$

$\sigma(l)$ is *eligible for paramodulation* if $l \in C^+$, $sel(C) = \emptyset$, and $\sigma(l)$ is maximal in $\sigma(C)$.

Conceptually, a literal that is eligible for resolution is a passive inference partner. A literal eligible for paramodulation, on the other hand, is actively used as a (conditional lazy) rewrite rule, i.e. the equivalence expressed by the literal is actually used in the inference. Note that our definition allows the selection of literals that allow more inferences then strictly necessary for a complete calculus. A particular example is the selection of positive literals in mixed clauses.

With these definitions, we can now formulate the generating inference rules from **SP**, using the normal notation for inference rules:

Equality resolution (ER)

$$\frac{s \not\simeq t \vee R}{\sigma(R)}$$

if $\sigma = mgu(s,t)$ and $\sigma(s \not\simeq t)$ is eligible for resolution.

Superposition into negative literals (SN)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

Superposition into positive literals (SP)

$$\frac{s \simeq t \vee S \quad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

Equality factoring (EF)

$$\frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)}$$

if $\sigma = mgu(s,u)$, $\sigma(s) \not< \sigma(t)$ and $\sigma(s \simeq t)$ eligible for paramodulation.

The inference system formed by the rules (ER), (SN), (SP), and (EF) subsumes the system $\mathcal{E}$ [4] (with literal selection), i.e. for each **SP**-selection function there is a version of $\mathcal{E}$ that allows at most the same inferences as **SP**. Since $\mathcal{E}$ is known to be complete, this implies the the completeness of our calculus as well. Moreover, we can inherit the notions of *fairness* (all non-redundant generating inferences between persistent clauses have eventually to be performed), and can also apply the general redundancy elimination schemes described for the standard superposition calculus. The potential to also select positive literals as paramodulation targets allows us a greater flexibility in the heuristic control – of course at the cost of a larger local branching factor. Experimental results show that this greater flexibility often is useful.

2.3. Contraction

While generating inferences are theoretically sufficient for completeness, most modern provers spend a great part of their time in simplifying (or *contracting*) operations, i.e. in inferences that remove or modify existing clauses. E is no exception. It uses a wide variety of contraction techniques, from the simple deletion of redundant literals to rewriting and to special redundancy elimination techniques for associative and commutative function symbols.

All simplifying inferences in **SP** are based on the notion of *compositeness* as defined in [4]. A clause is composite with respect to a set of clauses F, if all its ground instances are implied by smaller ground instances from clauses in F[3]. Composite clauses can be deleted from the proof state without affecting completeness of the calculus, and inferences which involve composite clauses are redundant. Tautologies are an example of clauses that are directly composite. We call inferences that simply delete composite clauses *deleting inferences*. However, in many cases we first need to deduce a new, simpler clause in order to make a clause redundant. In this case, we speak of *modifying* or *simplifying inferences*.

We can distinguish between three different classes of redundancy elimination techniques, depending on the number of clauses used in the justification of the step. A clause can be simplifiable or redundant on its own, with respect to another clause, or with respect to a recognized first-order theory. We use a common notational convention to describe all of these cases. A *contraction rule* is a rule of the form:

$$\frac{< \mathit{precondition} >}{< \mathit{conclusion} >} \text{ if } < \mathit{condition} >$$

An application of a rule is possible if the condition holds. In this case, the proof state is modified by *replacing* all clauses in the precondition with the clauses in the conclusion. Note the we frequently have the case that $< \mathit{conlusion} >$ is empty. In this case we also speak of a *deleting rule.*

2.3.1. Intra-clause simplification

The most basic contraction techniques eliminate unnecessary literals from a clause:

Deletion of duplicated literals (DD)

$$\frac{s \dot{\simeq} t \vee s \dot{\simeq} t \vee R}{s \dot{\simeq} t \vee R}$$

Deletion of resolved literals (DR)

$$\text{(DR)} \quad \frac{s \not\simeq s \vee R}{R}$$

[3] A few techniques, in particular rewriting of maximal terms, subsumption, and AC tautology deletion, require a more general definition of compositeness, based on a more complex *admissible literal ordering*[5] taking into account not only the ground instance, but also the substitution used to generate it.

Superposition is also compatible with the eager deletion of tautologies (since any instance of a tautology is a tautology and hence implied by the empty theory). The deletion of tautological clauses is quite important, especially in proof problems with a significant unit-equational sub-theory, since in this case often many clauses can be rewritten to equational tautologies.

We can give three sufficient criteria to detect tautologies: A clause is tautological, if it contains two identical atoms with opposite signs, or if it contains a trivial equation $s \simeq s$. Both of these criteria can be tested very efficiently. The third method, suggested in [17], subsumes these two, but is significantly more expensive in terms of CPU time. A clause is tautological, if the equational theory induced by the set of negated negative literals implies one of the positive literals, where all variables in the clause are treated as new constants. Since ground completion is guaranteed to terminate with a convergent system, this property can be tested with some implementation work.

Written in the form of contraction rules, these methods can be stated as follows:

Syntactic tautology deletion 1 (TD1)

$$\frac{s \simeq s \vee R}{}$$

Syntactic tautology deletion 2 (TD2)

$$\frac{s \simeq t \vee s \not\simeq t \vee R}{}$$

Semantic tautology deletion (SD)

$$\frac{s_1 \not\simeq t_1 \vee \ldots \vee s_n \not\simeq t_n \vee s \simeq t \vee R}{}$$

if $\sigma(s_1 \simeq t_1), \ldots, \sigma(s_n \simeq t_n) \models \sigma(s \simeq t)$, where σ is a substitution that maps all variables in the clause to distinct new constants

The last single-clause simplification in **SP** again is a modifying inference. It covers a special case of equality resolution:

Destructive equality resolution (DR)

$$\frac{x \not\simeq s \vee R}{\sigma(R)}$$

if $x \in V$ and $\sigma = mgu(x, s)$

This allows us to destructively solve a negative literal with one variable side simply by applying a suitable substitution to the clause.

2.3.2. Clause-clause contraction

Clause-clause simplification steps are the most important inferences in most current theorem provers. In particular, *rewriting* is a relatively efficient way to compute many functions defined by sets of equations. We distinguish between rewriting of positive literals (which is slightly more restricted) and rewriting of negative literals.

Rewriting of positive literals (RP)

$$\frac{s \simeq t \qquad u \simeq v \vee R}{s \simeq t \qquad u[p \leftarrow \sigma(t)] \simeq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$, and if $u \simeq v$ is not eligible for paramodulation or $u \not> v$ or $p \neq \lambda$ or σ is not a variable renaming.

Rewriting of negative literals (RN)

$$\frac{s \simeq t \qquad u \not\simeq v \vee R}{s \simeq t \qquad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}$$

if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$.

Rewriting only uses orientable instances of positive unit clauses. We can also use unorientable and negative unit clauses for simplification if we can eliminate a literal in one step.

Positive simplify-reflect (PS)

$$\frac{s \simeq t \qquad u[p \leftarrow \sigma(s)] \not\simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t \qquad R}$$

Negative simplify-reflect (NS)

$$\frac{s \not\simeq t \qquad \sigma(s \simeq t) \vee R}{s \not\simeq t \qquad R}$$

Rewriting is a modifying inference that has been primarily developed in the context of Knuth-Bendix completion [10] and its variants [9,3]. Subsumption, on the other hand, is a deleting contraction rule, and has its root in the field of resolution [21]. Subsumption allows us to delete a clause if the proof state already contains a more general clause that is, in a certain sense, smaller. In the case of equational logic, we also have a new kind of unit subsumption induced by positive unit clauses. The two subsumption rules can be specified as follows:

Clause subsumption (CS)

$$\frac{T \qquad R \vee S}{T}$$

if $\sigma(T) = S$ for a substitution σ

Equality subsumption (ES)

$$\frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$$

2.3.3. AC redundancy elimination

Associativity and commutativity are frequently encountered properties of binary operators. The most common examples are arithmetic addition and multiplication, but the property is also shared by many logical connectives occurring in hardware verification problems. As can be seen from these examples, both properties often occur together. In this case, they are particularly troublesome, because provers are bound to enumerate super-exponentially growing classes of permutative equations. Although equality modulo AC is easily decidable, conventional unfailing completion will not terminate if given just the AC axioms for a single function symbol.

Means of dealing with AC function symbols include AC unification and AC matching, combined with rewriting modulo AC. Such methods have been used to good effect in the prover EQP [14]. However, none of the current general purpose provers includes one of these techniques. They require significant implementation work, are in practice hard to use efficiently, and, if completeness of the prover is desired, restrict the choice of reduction orderings. In the case that dynamic AC recognition is desired, i.e. that the prover should also handles AC properties deduced only during the proof search, this restriction applies to most equational problems.

An alternative approach is to perform a normal proof search including explicit AC axioms, and only use the AC theory for the simplification and deletion of redundant clauses. This idea is based on the work done for the unit equational prover Waldmeister [1]. The core ideas for the AC case carry over to full superposition. Basically, it can be shown that there exists a small, ground convergent system for describing the AC property (as well as commutativity alone), and that no other consequences of the AC axioms are needed for the completeness. Moreover, we can simply delete neg-

ative equational literals if both terms are equal modulo the recognized AC theory.

More formally, consider the following definitions: A commutativity axiom for the binary function symbol f is a positive unit clause $f(x,y) \simeq f(y,x)$. An associativity axiom is a clause $f(x,f(y,z)) \simeq f(f(x,y),z)$. An AC specification AC is a set of associativity and commutativity axioms such that AC contains either only a commutativity axiom or both a commutativity and an associativity axiom for each operator. We say two terms s and t are equal modulo AC, written as $s =_{AC} t$, if $AC \models s \simeq t$. If f is associative and commutative, we define $\overline{AC_f}$ as the set of unit clauses
$\{f(x,y) \simeq f(y,x),$
$f(f(x,y),z) \simeq f(x,f(y,z)),$
$f(x,f(y,z)) \simeq f(z,f(x,y)),$
$f(x,f(y,z)) \simeq f(y,f(x,z)),$
$f(x,f(y,z)) \simeq f(z,f(y,x))\}$
If f is only commutative, we define $\overline{AC_f}$ as $\{f(x,y) \simeq f(y,x)\}$. For an AC specification AC, we define $\overline{AC} = \cup_f \overline{AC_f}$. Note that each $\overline{AC_f}$ is equivalent to the AC axioms used to induce it. It also is ground convergent for any Knuth-Bendix ordering or lexicographic path ordering[4]. It is easy to see that any $\overline{AC}$ also is convergent, since there exist no non-trivial overlaps between the clauses from different $\overline{AC_f}$.

Given these definitions, we can now specify our AC contraction rules:

AC tautology deletion (ACD)

$$\frac{s \simeq t \vee R \qquad AC}{AC}$$

if AC is an AC specification, $s =_{AC} t$, and $s \simeq t \vee R \notin \overline{AC}$

AC simplification (ACS)

$$\frac{s \not\simeq t \vee R \qquad AC}{R \qquad AC}$$

if AC is an AC specification, $s =_{AC} t$.

In addition to the rules explicitly stated in this section, we have recently implemented *splitting without backtracking* as originally introduced in Saturate [17] in the form of hyper-splitting without naming [18]. However, our preliminary evaluation rarely showed an improvement, and hence this option is not activated by default in the current version.

[4]We are aware of the fact that for the case of a function symbol that is both associative and commutative, $\overline{AC_f}$ is not minimal. However, this definition simplifies implementation and, perhaps surprisingly, seems work even better than the minimal 3 clause system.

3. Proof Procedure

The calculus allows arbitrary application of inferences rules. To create an efficient and complete proof procedure, we have to determine in which order inference rules should be applied. This saturation algorithm takes the form of a loop, and interleaves generating and contracting inferences. It terminates in two cases: If it runs out of (non-redundant) generating inferences, or if a proof (in the form of the empty clause in the proof state) is found. The three primary virtues we aim at with of our proof procedure can be stated as *fast, cheap, and lazy*:

- The proof procedure should be fast, i.e. allow for a high inference rate. Moreover, each individual traversal of the main saturation loop should only take a small amount of time. This allows for a reasonably fine-grained control of the proof search.
- The proof procedure should be cheap to implement. In particular, it should be possible to easily guide the proof search using only a small number of choice points, and to avoid to much administrative overhead. To achieve this, we opt to replace explicitly stored data with strong invariants on the proof state.
- Perhaps the most important property is laziness: The proof procedure should not do any work it can avoid. Contracting inferences take most of the time in current theorem provers. However, generating inferences are the ultimate cause of this work. So we try to avoid generating inferences if we can show that they are redundant. Moreover, we delay expensive contraction inferences as much as possible.

Nearly all current saturating high performance theorem provers use a variant of the *given-clause* algorithm. This algorithm represents the proof state by two sets of clauses, a set P of processed clauses and a set U of unprocessed clauses. Initially, all clauses are in U and P is empty. At each traversal of the main loop, the algorithm tries to

pick a clause c (the *given clause*) from U. If U is empty, the original clause set is satisfiable (and the clauses in P describe a model). If c is the empty clause, the unsatisfiability of the original clause set has been shown. Otherwise the algorithm performs all possible generating inferences where at least one premise is c and all other premises are arbitrary clauses from P. The generated clauses are added to U, and the process repeats.

The different variants of the given-clause algorithm differ in their handling of contraction. Most provers use a variant of the loop popularized by Otter [15], and hence sometimes called the *Otter loop*. The Otter loop is characterized by a pragmatic approach to contraction rules: It uses rewrite- and subsumption relations induced by different (disjoint) subsets of the clauses in the proof state in different situations. For an example, see the variant of the Otter loop described in [20] and implemented in Vampire, which uses three different rewrite relations within a single main loop traversal.

E, on the other hand, uses a different variant of the *given-clause* algorithm, the DISCOUNT loop, named after the first well-known system that implemented it [6]. It uses only the set P (and the given clause) to perform simplifications. A sketch of the algorithm, adapted for E's more general calculus, is shown in Fig. 1. To our knowledge, E is the only general purpose prover explicitly designed around this loop. It is, however, shared by Waldmeister, a system for pure unit equational problems, and both Vampire and SPASS added it later.

The algorithm uses a number of subroutines:

select_best(U) selects the (according to a heuristic evaluation) best clause from the set U.

simplify(c, P) performs all modifying inferences where c is the main premise (the clause that is modified) and potential other premises are clauses from P. This applies inference rules (RN) and (RP), followed by (PS), (NS), (ACS), (DD) and (DR).

cheap_simplify(c, P) is similar, but only applies inferences rules that are implemented particularly efficiently in E: (RN), (RP), (ACS), (DD) and (DR). Optionally, the user can restrict this function to use only orientable units or to not perform rewriting at all. Except for rewriting, the inference rules are implemented in a way that they depend only on the clause to be simplified, not on U or P (AC properties are detected at activation time and stored as function symbol properties), and hence can be applied efficiently regardless of the size of the proof state.

redundant(c, P) returns true if and only if c can be shown to be redundant with respect to P using one of the rules (SD), (CS), (ES) or (ACD).

trivial(p, P) similarly returns true if and only if c can be shown to be redundant with respect to P using only the efficiently implemented rules (TD1), (TD2) and (ACD).

generate(c, P) generates all direct conclusions between c and P using the rules (ER), (SN), (SP) and (EF), where at least one premise is c.

```
 1: while U ≠ ∅ begin
 2:    c :=select_best(U)
 3:    U := U\{c}
 4:    simplify(c, P)
 5:    if not redundant(c, P) then
 6:       if c is the empty clause then
 7:          success; clause set is unsatisfiable
 8:       else
 9:          T := ∅
10:          foreach p ∈ P do
11:          if c simplifies a maximal literal of
12:             p such that the set of maximal
13:             terms, the set of maximal literals or
14:             the number of literals in p potentially
15:             changes
16:          then
17:             P := P\{p}
18:             T := T ∪ {p}
19:             U := U\{d|d is direct descendant of p}
20:          fi
21:          simplify(p, (P\{p}) ∪ {c})
22:       done
23:       T := T∪ generate(c, P)
24:       foreach p ∈ T do
25:          p :=cheap_simplify(p, P)
26:          if not trivial(p, P) then
27:             U := U ∪ {p}
26:          fi
28:       done
29:       fi
30:    fi
31: end
32: Failure: Initial U is satisfiable, P describes model
```

Fig. 1. E's proof procedure

As can be seen, the basic proof procedure of E is quite straightforward. The proof state is represented by two clause sets U and P as in the basic given clause algorithm. The algorithm selects a new clause from U, simplifies it w.r.t. to P, then uses it to simplify the clauses in P in turn. Critically modified processed clauses are removed from P, their unprocessed direct descendents are deleted.

After this *interreduction* phase, the algorithm performs equality factoring, equality resolution and superposition between the selected clause and the set of processed clauses. The generated clauses are simplified and added to the set of unprocessed clauses. The process stops when the empty clause is derived or no further inferences are possible.

This algorithm has a number of desirable features, both from the theoretical and the implementation point of view:

- Most importantly, there is a strong invariant: In addition to the general given-clause invariant (at the start of the loop all, necessary generating inferences between clauses in P have been performed), P is interreduced, i.e. maximally simplified with respect to itself. Moreover, when the generating inferences are performed, even $P \cup \{c\}$ is interreduced. A similar invariant also can be achieved with the Otter loop, however, since full interreduction of $P \cup U$ is very expensive, most systems only approximate it.
- There is only one rewrite relation used throughout the whole main loop. With each traversal of the loop, the strength of the relation increases monotonically. Since this relation is induced by the set P only, which is typically much smaller than U, indexing is relatively cheap, and indices do not consume significant amounts of memory. Moreover, insertion and removal of terms from the index are relatively rare events.
- Clauses in U are truly passive, i.e. they are not used unless they are selected as the given clause and activated. Thus, they do not cause any significant amount of work. As long as sufficient memory is available, all clauses can be kept. If memory is tight, clauses with a bad evaluation can be deleted, in most cases without affecting the search process at all, since U typically is so large that only a small part of it is ever processed.
- Since clauses in U are not used in the proof process unless selected, we can easily and efficiently use an *orphan-criterion* (line 19 in the algorithm) to delete direct descendants of a clause d from P that can be shown to be redundant, where a *direct descendant* of d is any clause resulting from a generating inference where at least one premise is d, and an arbitrary number of modifying inferences.
- The influence of the heuristic clause selection (implemented by the function *select_best()*) is much more important than in the Otter loop, where unprocessed clauses play an important role and, if kept, cause significant amounts of work. Heuristic control is more fine-grained and typically more important than in the Otter loop.
- Since only the relatively small clause sets P and T (newly generated clauses and clauses removed from P by back-simplification) are involved in all costly operations, the number of processed clauses per unit time is typically very high.

The major disadvantage compared to the Otter loop is, of course, that clauses in U may be useful for rewriting, and may thus contribute to a proof. However, this disadvantage is mitigated in two ways: First, the higher inference rate also means that more clauses are processed. Secondly, even when using quite naive heuristic evaluation functions like plain symbol counting, small clauses will be selected early on. So most clauses in U are fairly large and specialized, and rarely useful for rewriting.

All in all, we made excellent experiences with our proof procedure.

4. Search Control

Good search heuristics are crucial for any fully automatic theorem prover for first-order logic. As we already stated in the previous section, this is particularly true for provers based on the DISCOUNT loop. Since E was always intended as a test bed for search control heuristics, we implemented a very flexible framework for integrating different heuristics, and we believe that heuristic search control is the major strength of E.

The three most important choice points are clause selection, literal selection, and selection of the term ordering.

4.1. Conventional Clause Selection

Good literal selection strategies can significantly improve the performance of a prover in the non-unit case, and the choice of the right term ordering sometimes is crucial especially for unit-equational problems. However, we believe that the order in which clauses are processed usually is the most important choice point. In fact, most proofs are short, and rarely need more than a few hundred inferences in total. On current hardware, E can usually select and activate several hundred clauses in a few seconds, theoretically enough for the vast majority of all proofs even for problems considered hard. However, a-posteriori analysis of proof search protocols shows that in practice between 90% and 99% of all processed clauses are superfluous, i.e. do not contribute to the proof. If we consider generated clauses, this ratio is even worse. The major reason for this is that typically, for non-trivial proof problems the *select_best()* function has to pick the *given clause* out of several tens of thousands to several million clauses.

We are not aware of any work offering a strong theoretical argument for any particular clause selection heuristic, and we believe that such an argument will be very hard or even impossible to make. Thus, major progress has to come as a result of experimental work. To facilitate such work, we implemented a very flexible interface for clause selection heuristics in E, allowing on the one hand easy combination and configuration of all implemented heuristics, and on the other hand easy addition of completely new clause evaluation schemes.

We will now give a short overview of clause selection in E. For a more thorough discussion, see [22]. Fig. 2 shows a simplified functional description of the *select_best()* function of a given-clause based theorem prover. The clauses in U are mapped to a totally ordered set E by a *heuristic evaluation function eval()*. The select function picks an arbitrary clause from those with the lowest (best) evaluation. Since the set U is very large, the evaluations of the clauses in U are computed *once* (after the initial simplification and before the clause is moved from T to U), and stored with the clauses. The set U is then organized as a priority queue ordered by the evaluations, new clauses are inserted at the proper position, and at each traversal of the main loop the first clause of the queue becomes the given clause.

```
1: function select_best(U)
2:     e :=min_{>_E}{eval(c)|c ∈ U}
3:     select c arbitrarily from {c ∈ U|eval(c) = e}
4: return c
```

Fig. 2. A simple *select_best()* function

It is important to notice that the order in which clauses are processed is fully determined by the evaluation function. The most common evaluation functions are based on *symbol counting*, i.e. they return the number of function symbols and variables (possibly weighted in some way) of a clause as the evaluation and thus prefer small clauses.

One common variation of this general scheme is the introduction of a second priority queue, sorted by age (time at which the clause was generated), and the alternating selection of clauses from either queue with a fixed ratio (the *pick-given ratio*. The pick-given ratio was popularized by Otter [15] and is e.g. used in Waldmeister [8] and Vampire [19]. We call this heuristic the *standard clause selection heuristic*.

E generalizes this concept and allows the user to specify an arbitrary number of priority queues and a weighted round-robin scheme that determines how many clauses are picked from each queue. Each clause is entered into and removed from each queue, but, of course, potentially ranked differently in each queue. Moreover, E uses composite evaluations, consisting of a *clause priority* (encoded as an integer value) and a *heuristic weight* (encoded as a floating point number)[5]. Evaluations are compared lexicographically in this order. Typically, the clause priority is used to specialize one queue for one class of clauses (by assigning a lower weight to this class), while the heuristic weight represents a measure for the expected usefulness of a clause within this class.

Consider the following example: To emulate a standard clause selection heuristic with pick-given ratio 5, E sets up two priority queues. In both queues, it assigns the same priority to all clauses. In the first queue, the heuristic weight is the number of symbols in the clause. In the second queue, the heuristic weight is the creation date of the clause. Finally, the first queue is assigned a weight of 5 and the second a weight of 1.

[5] In order to break ties in a defined way, E also adds the clause age as a third, hidden component, and, if the first two components are equivalent, prefers the older clause.

As a second example, consider a unit-equational proof problem with a single ground goal (a negative unit clause). Except in rare cases, there will always be exactly one goal, and to solve this goal it has to be rewritten to a trivial (in-)equation. For such cases, E can simulate a completion-based prover by setting up a single queue with an arbitrary heuristic weight function and a priority function that always prefers unit ground goals, ensuring that the single goal will always be chosen first and hence is guaranteed to be in normal form most of the time.

E currently implements about 15 different priority functions. The most important ones either assign a constant priority, prefer all negative clauses (assumed to be goals), prefer non-negative clauses, or prefer ground clauses. New priority functions can be added in a few lines of C code.

At least a part of the success of E is based on the strength of its heuristic weight functions. E has more than 15 different generic weight functions, which can be instantiated with different parameters to yield a clause evaluation function. However, only three of these are used frequently:

Clauseweight: This evaluation function implements simple symbol counting. The generic function takes three parameters: The weight to use for a function symbol, predicate symbol, or constant, the weight used for a variable, and a modification factor for positive literals. The basic idea behind symbol counting is two-fold. First, small clauses typically represent general concepts that are likely to be applicable in many situations. Moreover, small clauses are often useful for contracting inferences, especially subsumption and rewriting. Secondly, small clauses, with fewer term positions, typically generate fewer descendants. Thus, working with smaller clauses will delay the inevitable explosion of the search space.

FIFOweight: To realize oldest-first clause selection, this evaluation function simply returns the value of a counter that is incremented for each new clause. Thus, it implements the first-in-first-out heuristic, a refinement of breadth first search. If we ignore contraction rules, this heuristic will always find the shortest possible proofs (by inference depth), since it enumerates clauses in order of increasing depth.

Refinedweight: Perhaps the most important and novel evaluation function in E is *Refinedweight()*. This function modifies the symbol counting heuristic by assigning a higher weight to maximal terms and maximal (or selected) literals. Again, the intention is two-fold. First, for unit clauses, this will prefer orientable clauses (rewrite rules) to unorientable ones. Thus, it will lead to a stronger rewrite relation earlier. Secondly, note that only maximal terms in maximal (or selected) literals are available for generating inferences. By preferring clauses with few (and small) terms eligible for inferences, we again curtail the early explosion of the search space.

In practice, we found that we get the best results with clause selection functions that combine three or four different priority queues, using two different instances of Refinedweight() and concentrating on goals and non-goals, respectively, with the remaining queues using a FIFO scheme and sometimes simple clause weight with constant priority.

4.2. Learning Good Clause Evaluations

The great flexibility of specifying clause selection heuristics allows us to optimize E for different domains with relative ease. However, this process still requires tedious manual experimentation and proof analysis. To relieve the users and developers from this task, E can also learn good clause evaluation heuristics by automated analysis of its own proof search protocols. The prover builds a data base of patterns of clauses successfully used in the proof and of superfluous clauses derived in at most a small number of inferences from those. Each pattern is assigned an evaluation, and one of several term-based machine learning algorithms is used to generate an evaluation function for new patterns (and hence clauses). This feature is described in detail in [22,23] and has not been used in any CASC competition so far. Hence we refrain from a more detailed discussion here.

4.3. Literal Selection

If we only consider completeness, literal selection is a case of *don't care non-determinism*. Any valid literal selection function will lead to a com-

plete calculus. This freedom gives us the possibility to apply heuristic criteria to literal selection. In fact, our experiments have shown that the choice of a good literal selection function can have an enormous influence on the efficiency of the proof search.

We can split the problem of literal selection into two distinct sub-problems. First, we have to decide if we want to select any literals in the clause. If we have decided in favour of literal selection, we then have to decide which literals to select. In our experience, the first decision is harder. We use the following definitions: A literal selection function is called *strict*, if $C^- \neq \emptyset$ implies that $sel(C) \neq \emptyset$ for all clauses C, i.e. if literals are selected whenever possible. A literal selection function is *minimal*, if $|sel(C)| \leq 1$ for all clauses. Finally, *sel* is called *non-redundant* if $sel(C) \cap C^+ = \emptyset$ for all clauses.

Literal selection functions potentially reduce the local branching factor of the proof derivation by two mechanisms. First, if there is more than one maximal literal within a clause, literal selection can be used to reduce the number of inference positions by selecting a smaller number of literals. Secondly, if a clause has a positive literal that is maximal, the clause is used both as the active and as the passive partner for superposition inferences. If a literal selection function is used, the clause is only available as the passive inference partner. Strict literal selection functions lead to a *positive* strategy, i.e. a strategy where the active partner in a superposition inference always is a positive clause. Since in practice most long clauses are mixed clauses, this reduces the effect of the *duplication problem* [11] encountered in resolution and paramodulation inferences. The duplication problem originally refers to the fact that in a resolution inference, all but two literals are instantiated and copied from the premises to the conclusion. This often leads to an exponential increase in clause length over time. This is even worse for paramodulation inferences, since typically all but one literals are copied to the conclusion. By restricting at least one partner to a positive (and hence usually shorter) clause, this effect can be reduced significantly. In particular, for the Horn case strict literal selection results in a unit strategy, and thus clause length will never be increased by any inference. Non-strict literal selection still approximates the effects of strict selection.

From the above, it appears that strictness, minimality and non-redundantness are desirable properties, since they all help to keep the branching factor low and hence limit the explosion of the search space. However, in practice we found that this is only true in some circumstances. We obtain our very best results with literal selection functions that have none of the above properties.

We can see a likely reason for this if we consider a clause as a conditional rewrite rule. The Horn clause $f(x,y) \simeq x \vee y \not\simeq s(0)$ can be e.g. be seen as the rule $y \simeq s(0) \rightarrow f(x,y) \simeq x$. In any conventional literal ordering, the positive literal is maximal, and the clause can be used to paramodulate into a term, binding the variable y. For this fixed y we can then try to verify if the condition $y = s(0)$ holds. If, on the other hand, we select the negative literal, only paramodulation into $y \not\simeq s(0)$ is possible, and we rely on pure chance to find a useful instantiation for x and y. Similar effects can be seen with many proof problems specified in a top-down manner, especially in problems written by people used to the PROLOG depth first search strategy. Consider a goal clause $\neg P(f^i(a))$, and an arbitrary theory including $\neg P(x) \vee P(f(x))$, $P(a)$ and $P(b)$ and $P(c)$. Without literal selection, the goal can be reduced directly to $\neg P(a)$ in i steps. However, with a strict literal selection function, we have to generate all consequences of the theory, potentially including the superfluous $P(f^j(b))$ and $P(f^j(c))$. This and similar effects can be observed very often in the PLA domain of TPTP.

In E, we use two approaches to deal with this problem. First, we use a variety of non-strict literal selection functions, each of which uses a different heuristic to determine clauses in which no literal selection should be performed. We will describe one of these heuristics below. Secondly, we use non-minimal and redundant literal selection functions that allow us to overlap into positive literals to (hopefully) generate useful instances of the clause.

If we have decided to select literals in a clause, we next have to decide *which* literals to select. Note that if we use a strict selection function, we have to remove *all* all negative literals before we can use the clause (or one of its descendants) as the active partner in a generating inference. Thus, we should generally try to solve the hardest literal first. If we cannot solve the hardest literal, all work spent on other literals is wasted. Unfortunately, determining the hardest literal to solve is, in general undecidable, and even reasonable approximations require consideration of the whole proof state, and

hence are expensive. However, we think the following criteria are useful:

- Ground literals are generally hard to solve, since they can only be solved by (conditional or unconditional) rewriting, not by instantiation. Moreover, selecting ground literals has two other benefits as well. First, there will be (nearly) no generating inferences with unit clauses necessary, since most potential superposition inferences also are valid rewrite steps. The only exception are unit clauses where a free variable occurs in the non-matching side (a very rare case in practice). Secondly, solving ground literals will not instantiate the remaining clause.
- In the equational paradigm, negative literals are solved if both terms can be shown to be equal under the equational theory at hand. The difference in the size of the terms of the inequation is a very rough measure of the difficulty of this proof.

Following these general ideas, we have created and experimented with about 60 different literal selection functions in E, about 15 of which we found to be useful for a reasonably large class of problems. At the moment, each literal selection function is implemented as a separate piece of C code in the prover. The functions have a simple and uniform interface and usually consist only of a few lines of code. Each function decides both if it should select literals at all, and if yes, which literals. While we aim at a more general interface separating these questions, due to the ease of adding new hard-coded functions there has not yet been any strong pressure to do so.

Some of our most interesting or useful literal selection functions are the following:

NoSelection never selects a literal. The proof search is performed using standard superposition. This strategy is useful e.g. in the planning domain (PLA) of the TPTP problem library [27].

SelectDiffNegLit is a strict, minimal and non-redundant literal selection function. It always selects an arbitrary negative literal among the literals with the largest difference in symbol count for both sides.

SelectComplex also is a strict, minimal and non-redundant selection function. If there is at least one literal of the form $x \not\simeq y$, it selects the first one. If there is no such literal, but at least one negative ground literal, it selects the smallest negative ground literal. Otherwise it selects as SelectDiffNegLit. This is one of our strongest strict literal selection function and is used as the base of most non-strict selection functions.

SelectComplexExceptRRHorn is a non-strict, minimal and non-redundant selection function. If the clause is a range-restricted Horn clause (i.e. a clause where all variables occur in the single positive literal), it assumes that this clause should be used as a conditional rewrite rule, and hence refrains from selection. Otherwise, it selects as SelectComplex. This is one of our strongest minimal literal selection functions.

All of these literal selection functions (except for NoSelection) have a non-minimal, redundant counterpart that also selects *all* positive literals. In this case only the selected negative literal and the literals maximal among the positive literals are eligible for resolution. In general, we found that if we use relatively weak clause selection functions, minimal and non-redundant literal selection functions work best. However, if we use better clause selection functions, the non-minimal literal selection functions outperform the minimal ones, although usually by a small margin only.

To illustrate the effect of literal selection: If we use the standard clause selection heuristic with a pick-given ratio of 5, E can solve 1581 out of the 3952 non-unit problems from TPTP 2.4.1 on a SUN Ultra 60/300 MHz within a 300 second time limit. With SelectDiffNegLit, it can solve 1814 problems, with SelectComplex 1811, and with SelectComplexExceptRRHorn 1908 problems. There is no significant difference between the minimal and the non-minimal variants for these parameters.

4.4. Term Orderings

While literal selection is only important for non-unit problems, the choice of the right term ordering is particularly important for unit-equational problems. E implements both the Knuth-Bendix ordering (KBO) and the lexicographic path ordering (LPO). Both classes of orderings are pa-

rameterized. The KBO requires a weight function assigning weights to individual function symbols (and a fixed weight for all variables), and both orderings require a precedence on the function symbols.

We believe that the selection and generation of term orderings is the weakest part of E's heuristic control component. Basically, E offers a number of simple, problem-independent weight- and precedence generation schemes which can be selected by the user. Alternatively, the user can specify the precedence explicitly. As a typical example, consider the term ordering most often used by the automatic mode described in the next section. This is a Knuth-Bendix ordering. All function symbols receive the same weight 1 also used for variables. Function symbols are ordered by arity, with function symbols with a higher arity being bigger than function symbols with a lower arity in the precedence. The precedence between symbols of the same arity is chosen arbitrarily.

This ordering has the nice property that it never allows a rewrite step that would increase the symbol count. Moreover, it tends to eliminate function symbols with a high arity over time. Thus, it is generally a reasonable choice. However, the success of Waldmeister [8] shows that problem-specific orderings, generated after careful analysis of the axioms, can result in dramatic improvements.

4.5. Automatic Prover Configuration

Otter introduced an automatic mode that analyzes a proof problem and select a suitable strategy, thus relieving the user from this task. Such an automatic mode by now is standard for most leading theorem provers. In E, the automatic mode picks a clause selection heuristic, a literal selection heuristic, a term ordering, and sets a number of minor parameters. The novel property of E's automatic mode, however, is that it is not written by the developers, but generated automatically from a predefined partition of the problem space and a list of test results.

First, we use a set of features to partition the space of all proof problems into different classes. The set of features used changes frequently, however, the most important properties we are currently using are the following:

- Are the axioms (non-negative clauses) in the problem unit, Horn, or non-Horn clauses?
- Are the goals (negative clauses) unit or Horn?
- Does the problem contain only equational literals, some equational literals, or no equational literals?
- Do the goals contain variables or are they ground?
- Is the maximal arity of any function symbol 0, 1, 2, or greater than 2?
- Is the maximal arity of a predicate symbol 0, 1, or greater than 2?
- Is the number of clauses in the problem specification small, medium or large? The limits of these subclasses are automatically selected so that they split a (hopefully representative) training set into 3 approximately equal parts.

Each of the resulting classes is (potentially) assigned a separate search heuristic[6]. To determine this assignment, we run as many heuristics as feasible with our limited computing resources over a training set. We order the set of heuristics by overall performance (measured primarily by number of solutions found, and using the average time per success in the case of ties). A small program traverses the set of heuristics in descending order and assign to each class the first (i.e. the most general) heuristic that solves the maximal number of problems in this class. Output of this program is the C code implementing the automatic mode.

Since the classes are predefined by the developer, there is a certain risk of over-specialization. We have very occasionally observed this fact, especially if a class contains only a small number of problems. However, in general the automatic mode is much better than the first heuristic picked even by an experienced user. This also showed in the recent CASC-JC competition [26,28]. For the first time since the inception of CASC, some of the problems were new for all participants. While the number of such problems was too small for a reliable judgment, E performed at least as good as other high-performance theorem provers on the new problems in both the MIX and the UEQ categories.

5. Implementation

E is implemented in ANSI C, using the GNU C compiler. One of our aims is wide portability. The

[6] In practice, most heuristics cover multiple classes.

latest version of E has been tested on different versions of Solaris, GNU/Linux and HP-UX. Reasonably current versions have been compiled successfully for Intel/X86 processors, the Intel/Itanium, HP's PA-RISC, and various generations of SPARC processors.

5.1. Term Representation and Shared Rewriting

The inference engine of E is built around *perfectly shared terms* as the core data type. This means that any unique subterm in the current proof state is only represented once. There are two exceptions to this. First, a short-lived temporary copy of the given clause is used to ensure that all premises for an inference are actually variable disjoint. The second exception are individual term nodes that represent top positions of maximal terms in literals eligible for resolution. This terms can only be rewritten under stricter conditions, and need to be distinguished from syntactically identical terms that occur at different positions.

The shared term data structure is realized as a general *term bank* where terms are indexed by top symbol and pointers to the argument terms. A combination of hashing and splay trees [24] is used for efficient access to the terms stored in the bank. Terms are administrated using reference counting and superterm-pointers. Consequently, they are inserted bottom-up (starting with the subterms) and removed top-down. For reasonably hard problems, sharing typically saves between 80% of all term nodes for unit-equational problems, and 99.995% of all term nodes for hard non-Horn problems.

As term nodes are shared between a large number of clauses, we can afford to store several precomputed values with each term. In our case this includes the term weight (which is computed automatically during normal form building), a flag to denote reducibility with respect to a currently investigated rule or equation, and, most importantly, *normal form dates*. As stated in section 3, the strength of the rewrite relation increases monotonically over time. Hence, if a term is in normal form with respect to the rewrite relation at a certain certain time, only newer unit clauses have to be considered in later rewrite attempts.

E not only shares terms to save memory, but also performs rewriting on the shared term representation. If a rewrite rule is applied to any subterm in any clause, all shared occurrences of this subterm in all clauses will be replaced. As this may influence superterms, the change is propagated recursively to all superterms. For a more detailed discussion of the shared rewriting paradigm, see [12].

5.2. Indexing and Subsumption

Like almost all other high-performance saturating provers, E uses indexing techniques to speed up common operations. In particular, E uses perfect discrimination trees [13,7] with age and size constraints to speed up most simplifying unit operations: Subsumption, forward-rewriting, and simplify-reflect. As current *hot spots* in the code do not involve unification or, for most proof problems, non-unit-subsumption, we have not yet implemented indexing for generating inferences and non-unit-subsumption.

Matching is at the core of most contracting inferences, unification at the core of most generating ones. Since generating inferences are only performed between the selected clauses, the effort for contraction usually outweighs the effort for generation by far. In particular, we found unification to be very cheap despite its theoretically exponential behaviour. Consequently, unification is implemented in a straightforward manner. Nevertheless, it is still less costly than e.g. the checking of ordering constraints or even the construction of new terms for newly generated clauses.

Matching attempts (and the associated tests for orientability of the generated instances), on the other hand, are a major contributor to the overall CPU usage of most theorem provers which perform rewriting. While each individual matching attempt is cheap, the search for matching rules and equations from the set of processed clauses is quite expensive. We have therefore implemented an indexing scheme that makes use of our shared term representation to optimize the access to these clauses.

The aim of an index for rewriting is the following: Given a term t and a set of unit-clauses P, find (sequentially or all at once) all clauses $l \simeq r$ from P such that $\sigma(l) = t$. E uses perfect discrimination trees to implement this operation. A perfect discrimination tree basically treats a term as linear word, and branches on the symbol at each position in this word. Each node in the tree thus represents a set of terms with a common initial sequence, each leaf represents complete term and

points to the set of all clauses which are indexed via this term. In E, we annotate all nodes in the tree with age constraints. Each node contains the activation date of the youngest clause indexed by any leaf reachable from this node. When searching for a rule that simplifies a term t, we usually know that t is in normal form with respect to some earlier rewrite relation. Thus, if we reach a node in the discrimination tree which only indexes older clauses, we can ignore this branch and immediately backtrack. We similarly use size constraints to cut of branches, using the fact that a term can only match terms of smaller or equal size. Use of age and size constraints safes as much as 30% of CPU time for matching, or about 10% of overall run time for the prover.

While matching for unit operations is quite satisfactory for E, non-unit subsumption is performed using sequential search through the set of candidate clauses. While this is potentially costly, we use weight comparisons and pre-matching to reduce the number of candidates for full subsumption significantly. Moreover, subsumption is only performed when a clause is selected for processing, and only processed clauses are used as subsumption candidates. Thus, in practice subsumption is not a major problem for most proof problems. Nevertheless, extending the indexing scheme to deal with non-unit clauses is a high priority for us.

6. Performance

We have evaluated the current version of E, E 0.62dev, on all clause normal form problems from the TPTP problem library, version 2.4.1. E was running in fully automatic mode[7], with a memory limit of 192 MB and a time limit of 500 seconds on our cluster of SUN Ultra 60/300MHz machines. Table 1 shows the results.

If we compare these results with other state-of-the-art systems, we find that E performs very well for unit and Horn problems. It is relatively weak in the general class without equality. It should also be noted that E is complete as long as it does not run out of memory, and thus is able to show the satisfiability for a significant number of problems.

[7]The automatic mode was generated from test results obtained on a previous TPTP release, 2.3.0.

Problem class	Size	Proofs	Models	Total
Unit, no equality	11	8	3	11
Unit, equality	456	368	3	371
Horn, no equality	644	564	16	580
Horn, equality	563	403	47	450
General, no eq.	870	345	143	488
General, equality	1875	643	75	718
Overall	4419	2331	287	2618

Table 1
Performance of E on different problem classes

For this test run, there is only a single problem for which the prover terminates within the time limit, but does not give a definitive answer on the satisfiability of the problem. However, E terminates less often than e.g. SPASS [30] on satisfiable problems. This is understandable, since we have never optimized the prover for this case.

7. Conclusion

We have described the current state of our equational theorem prover E. E is a powerful, fully automatic theorem prover combining a number of well-known and novel features. It is stable, portable, and, thanks to its powerful automatic mode, easy to use.

The home page of E can be found on the web at `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`. It contains further information, links to many relevant papers, and the source distribution of the latest released version.

Despite the success of E, there still is a lot to do. We intend to work on three of the four major areas of E.

Calculus: On the calculus level, more advanced clause splitting techniques may be useful. They cam be combined with restricted or even full versions of contextual rewriting, i.e. the eager use of conditional clauses for rewriting if the conditions can be solved in the context of the clause to be rewritten. In the longer term, we will also experiment with inheritable ordering constraints.

Heuristic Control: This area offers the most potential for improvements. Future work will include a more flexible interface to the literal se-

lection component, new generic weight functions, and the possibility to dynamically adjust the generalized pick-given ratio used by E. Moreover, better analysis of the proof problem should enable us to pick more suitable term orderings in the future.

Inference Engine: The most urgent issues are the use of non-unit indexing for subsumption, and the online generation of detailed proof objects. Currently, proof objects are created in an expensive reproduction process, and are not very detailed.

References

[1] J. Avenhaus, T. Hillenbrand, and B. Löchner. On Using Ground Joinable Equations in Equational Theorem Proving. In P. Baumgartner and H. Zhang, editors, *Proc. of the 3rd FTP, St. Andrews, Scottland*, Fachberichte Informatik. Universität Koblenz-Landau, 2000. (revised version to be published in the Journal of Symbolic Computation).

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

[4] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[5] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[6] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[7] P. Graf. *Term Indexing*, volume 1053 of *LNAI*. Springer, 1995.

[8] T. Hillenbrand, A. Jaeger, and B. Löchner. System Abstract: Waldmeister – Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 232–236. Springer, 1999.

[9] J. Hsiang and M. Rusinowitch. On Word Problems in Equational Theories. In *Proc. of the 14th ICALP, Karlsruhe*, volume 267 of *LNCS*, pages 54–71. Springer, 1987.

[10] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Algebra*, pages 263–297. Pergamon Press, 1970.

[11] S.-J. Lee and D.A. Plaisted. Eliminating Dupliction with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[12] B. Löchner and S. Schulz. An Evaluation of Shared Rewriting. In H. de Nivelle and S. Schulz, editors, *Proc. of the 2nd International Workshop on the Implementation of Logics*, MPI Preprint, pages 33–48, Saarbrücken, 2001. Max-Planck-Institut für Informatik.

[13] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[14] W.W. McCune. 33 Basic Test Problems: A Practical Evaluation of Some Paramodulation Strategies. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 5, pages 71–114. MIT Press, 1997.

[15] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[16] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 373–439. Elsevier Science and MIT Press, 2001.

[17] P. Nivela and R. Nieuwenhuis. Saturation of First-Order (Constrained) Clauses with the Saturate System. In C. Kirchner, editor, *Proc. of the 5th RTA, Montreal*, volume 690 of *LNCS*, pages 436–440. Springer, 1993.

[18] A. Riazanov and A. Voronkov. Splitting without Backtracking. In B. Nebel, editor, *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001), Seattle*, volume 1, pages 611–617. Morgan Kaufmann, 2001.

[19] A. Riazanov and A. Voronkov. Vampire 1.1 (System Description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 376–380. Springer, 2001.

[20] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, (to appear).

[21] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[22] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000. Ph.D. Thesis, Fakultät für Informatik, Technische Universität München.

[23] S. Schulz. Learning Search Control Knowledge for Equational Theorem Proving. In F. Baader, G. Brewka, and T. Eiter, editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer, 2001.

[24] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[25] G. Stenz and A. Wolf. E-SETHEO: An Automated^3 Theorem Prover – System Abstract. In R. Dyckhoff, editor, *Proc. of the TABLEAUX'2000*, volume 1847 of *LNAI*, pages 436–440. Springer, 2000.

[26] G. Sutcliffe. The CASC-JC Web Site. `http://www.cs.miami.edu/~tptp/CASC/JC/`, 2001.

[27] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[28] G. Sutcliffe, C.B. Suttner, and J. Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, 2002. (to appear).

[29] A. Voronkov. The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees. *Journal of Automated Reasoning*, 15(2):238–265, 1995.

[30] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, G. Jung, E. Keen, C. Theobalt, and D. Topic. System Abstract: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 378–382. Springer, 1999.