An Empirical Study of Term Indexing in the Darwin Implementation of the Model Evolution Calculus *

John W. Wheeler

Guarionex Jordan

Computer Science Department

The University of Iowa

Iowa City, IA 52242

wheeer@cs.uiowa.edu

gjordans@math.uiowa.edu

May 12, 2004

1 Introduction

The Model Evolution calculus by Baumgartner and Tinelli lifts to first order logic the DPLL procedure that has been very successfully employed for propositional logic. The Darwin solver is the first implementation of the Model Evolution calculus and, as with any new implementation, once core functionality is in place the process of incremental improvement begins. This report details our effort to implement discrimination tree term indexing in the Darwin solver. This new term indexing capability provides an alternative to the substitution tree term indexing that is currently available in the Darwin solver.

*Partially supported by funding from the U.S. Army Materiel Systems Analysis Activity.

This report is organized as follows. Section 2 gives background about the Model Evolution calculus and the Darwin implementation. Section 3 discusses term indexing in general and discrimination tree term indexing in particular. Section 4 discusses our implementation of discrimination tree term indexing for the Darwin solver. Section 5 gives a description of the tests we ran to compare three term indexing configurations of the Darwin solver and the results obtained. Finally, Section 6 presents our conclusions and outlines some directions for future work.

2 The Model Evolution Calculus and Darwin

The Model Evolution calculus by Baumgartner and Tinelli is a recent development in the field of solution methods for first order logic. Motivated by the success of the Davis, Putnam, Logemann, and Loveland [5, 4] (DPLL) solution procedure for propositional logic, Baumgartner and Tinelli have developed the Model Evolution calculus to lift the DPLL procedure from propositional logic to the first order logic. The Model Evolution calculus can be expressed in a set of sequent style rules that closely mirrors that used in [11] to declaratively express the DPLL procedure. Both calculi follow essentially the same process in building a Herbrandt model for an initial clause set. The difference between the two calculi is that, while in the DPLL calculus clauses consist of disjunctions of only ground literals, in the Model Evolution calculus clauses may contain general first order literals. Because of this difference, the tests used to determine the applicability of the individual rules are very different. Details of the Model Evolution calculus may be found in Baumgartner and Tinelli [2, 3].

Relevant to the current discussion is that the Model Evolution calculus maintains a collection of asserted literals known as a context. This collection of literals represents the current state of the search for a model of the input clause set. In the process of applying the calculus, the decision to apply any given rule depends upon the comparison of candidate literals with all literals already contained in the context. Four basic tests are used to compare candidate literals to those in the context. These four tests determine whether a literal in the context is a variant of, an instance

of, a generalization of, or unifiable with one of the candidate literal. This process of comparing candidate literals to those in the context generally constitutes a significant portion of the effort involved in solving a input problem as the context may grow exponentially.

Also relevant is that the Model Evolution calculus uses two disjoint infinite sets of variables in addition to the usual infinite set of Skolem constants. These two sets of variables are known as *universal variables*, identified as the set $X = \{x, y, \ldots\}$, and *parameters*, identified as the set $V = \{u, v, \ldots\}$. The presence of these disjoint sets of variables complicates the computation of the four basic tests mentioned above in a manner that will be discussed further below.

The Darwin solver is the first implementation of the Model Evolution calculus. It is written in the Ocaml language and is intended to be a clean, fast implementation that may be used to allow a first assessment of the potential of the calculus. The Darwin solver maintains the context in a term database which currently includes a substitution tree based term indexing capability. Further details on the Darwin solver may be found in [1].

3 Term Indexing

As the use of databases of asserted literals is common among theorem proving systems and the process of comparing candidate literals to those contained in such a database constitutes a significant portion of the solution effort, much research has gone into developing efficient methods for this process. One such method is term indexing. Based on the same idea as indexing in a general database, term indexing attempts to filter from the entire set of literals contained in context database only those that possess a given relationship with a query literal.

The process of term indexing can be expressed formally as follows: Given a set L of *indexed terms*, or literals, a binary relation R over terms to be used as the *retrieval condition*, and a term t, the *query term*, find the subset N of L of terms l for which $R(l, t)$ holds. If the the term indexing mechanism identifies exactly that subset of terms for which the condition holds, the index is said to perform *perfect filtering*. If the index instead returns a superset of the set N, it is said to

perform *imperfect filtering*. A term index operates in one of two modes which determines whether the entire set N or simply a representative term from the set is returned by the term index.

As mentioned before, the four main relations $R(s,t)$ used for retrieval conditions are variant, instance, generalization, and unification. These relations are generally expressed in terms of substitutions α and β, such that $s\alpha = t\beta$. The four relations differ by the restrictions placed on these substitutions. In the variant relation, both substitutions are constrained to be renamings. If β is restricted to be the empty substitution, the relation is that of instantiation, while if α is the empty substitution, the relation is that of generalization. Finally, if both substitutions are unrestricted, the relation is unification.

Because the Model Evolution deals with two disjoint sets of variables, it specifies further restrictions on the substitutions that may be used to define these relations. The *restriction* of a substitution σ to a set W of variables is defined as $x\sigma$ if $x \in W$, and x otherwise. A substitution is called parameter preserving, denoted *p-preserving*, if the restriction of the substitution to the set of parameters V is a renaming on V. In the Model Evolution calculus, each of the four retrieval relations come in two different forms, p-preserving and non-p-preserving, depending on whether or not the substitutions used to define the relation are parameter preserving.

Term indexing methods differ in the mechanism used to organize the indexed terms for retrieval. A discussion of many examples of the term indexing methods that have been studied may be found in [9] and [6]. These examples include methods such as path indexing, discrimination tree indexing, the use of adaptive automata, and substitution tree indexing. Of particular interest here is discrimination tree indexing.

In discrimination tree term indexing, all terms are compared using strings of symbols, called *p-strings*, that are generated by a preorder traversal of the term structure. This preorder traversal flattens the term structure and allows comparison of simple strings of symbols in place of the more complicated comparison of term tree structures. For example, a preorder traversal of the term $g(f(x,y),g(x,x))$ yields the p-string, or list of symbols, 'g, f, x, y, g, x, x'. In standard

discrimination tree term indexing, p-strings are simplified further by replacing each occurrence of any variable with a special symbol '$*$'. Using these p-strings, the task of identifying pairs of terms for which there exists substitutions meeting the retrieval condition requirements becomes one of comparing their p-strings symbol by symbol. When a variable occurs in one p-string that may be modified by an allowable substitution, the matching process accepts any single symbol in the other p-string as a match. If this matched symbol is a function symbol of arity n, then n additional symbols are matched. This process is applied recursively until a single complete function application is matched. Graf [6] gives a functional description of the processes involved in discrimination tree term indexing for the four basic retrieval conditions using p-strings in this manner.

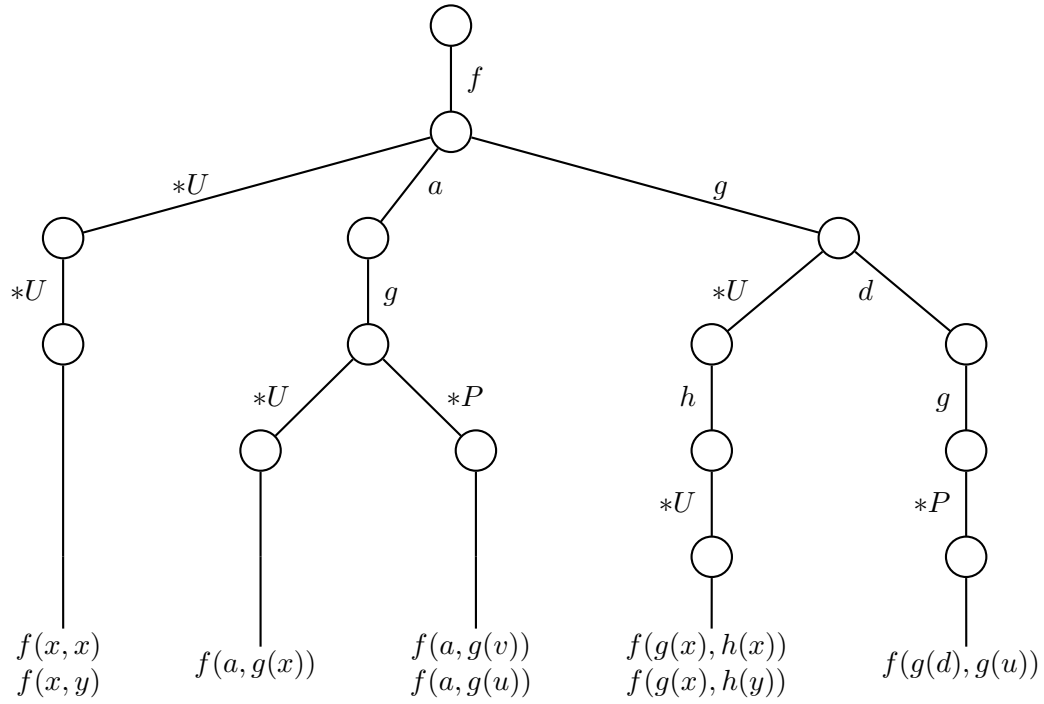
To accommodate the Model Evolution calculus, this standard discrimination tree term index method must be strengthened to address both universal and parametric variables. In the present case, this was done in a straight-forward manner by using two special symbols, $*U$ and $*P$, the first to represent all universal variables, the second to represent all parameters.

A discrimination tree term index stores the p-strings representing all indexed terms in a trie data structure in which edges of the trie are labeled with the symbols occurring the the p-strings. As some information is lost by using special symbols for the universal variables and parameters, lists of the original terms are stored in the leaf nodes of the trie. An example of a discrimination trie structure is shown in Figure 1.

4 Implementation Details

The current work was completed in the context of an implementation project for a course on automated reasoning. Because of this context, certain limitations in scope of the implementation were agreed upon to help ensure the successful completion of the class project. First, available open source implementations of discrimination tree term indexing were to be considered for pos-

Figure 1: Example Discrimination Tree.



sible adaptation in the current work. Second, that while the term indexing module signature in the Darwin code specified several retrieval conditions beyond the basic four relations discussed above, the current implementation need only respond to the p-preserving and non-p-preserving forms of those four basic retrieval conditions.

In regard the first of these limitations, the discrimination tree term indexing implementation found in KRHyper, a hyper-tableau solver, proved especially useful. Details of the Ocaml implementation of the KRHyper solver may be found at [12]. The KRHyper solver was developed at the University of Koblenz and Landau, is also written in the Ocaml language, and has been placed under the GNU General Public License. For these reasons, this implementation was used as a starting point for the current implementation.

The KRHyper code contained an implementation, in about 600 lines of Ocaml code, of the functional notation for standard discrimination tree term indexing given in [6]. This implemen-

tation uses hash tables in the trie interior nodes to encode the labels on trie edges and simple lists to record terms in leaf nodes. This implementation could not be used directly in the Darwin solver for two reasons. First, the structure of the term data used in the KRHyper solver is very different from that used in the Darwin solver. Second, the functional interface, or signature, used to allow the rest of the program to access the term indexing functions is very different in the two programs.

Due to these two differences, only those portions of the KRHyper code that provided basic functions for manipulating the trie data structure remained intact in current implementation. Building on this base, we implemented revised code to handle the term data used in the Darwin program and to respond to the different functional signature. Additional code was written to handle the two disjoint types of variable. In total, approximately 240 lines of the original code remained intact, 360 lines were deleted, modified, or rewritten, and about 300 lines of code were added.

A part of the originally agreed upon implementation task was to minimize the impact on existing Darwin code required by the addition of discrimination tree term indexing. We believe this portion of the implementation requirement was successfully met as only a single line of the core Darwin code need be modified to incorporate the revised term indexing method.

This near total separation of the current work from the main body of the Darwin code lead to an implementation challenge during integration testing of the our discrimination tree term indexing code. It was at this time we discovered the Darwin code relied upon the term index to function as a perfect filter. This feature is provided naturally by the current substitution tree term indexing code, however, it is not provided by the standard term indexing we had chosen to implement.

To minimize schedule risks, the decision was made to retrofit our existing implementation with a linear scan of the terms returned by the standard discrimination tree term index to remove terms that did not strictly comply to the retrieval condition. This allowed to provide the required perfect

filtering without reimplementing the standard discrimination tree as a perfect discrimination tree term index. This decision leaves as candidate for future work reimplementing our term index as a perfect discrimination tree to determine whether additional improvement can be made.

The following functional specifications detail the intent of our implementation of discrimination tree term indexing for the Darwin solver. This functional specification follows the form of that given in Graf [6]. We have extended Graf's notation slightly by using an *if-then-end if* construct to allow a succinct representation of both the p-preserving and non-p-preserving forms of the query conditions. Our contribution here is in extending the functional specification to handle both p-preserving and non-p-preserving forms of these conditions.

These specifications depend upon the following definitions. For term t, the denotation $[t]$ represents the p-string of t. The label of the edge leading from node N to node N' in the trie is denoted by $label(N, N')$. The list of terms store in a leaf node N of the trie is denoted by an overloading of the set notation $\{N\}$. An empty tree is denoted ϵ. Nodes in the discrimination trie are reached from their parent node by a lookup function $next(N, s)$ defined as follows.

$$\begin{aligned} next(N,s) &:= N' \text{ if there is a } N' \text{ s.t. } label(N,N') = s \\ next(N,s) &:= \epsilon \quad \text{otherwise} \end{aligned}$$

One additional function, the Skip function, is required by this notation. The Skip function is defined as follows.

$$\begin{aligned} skip(n) &:= \bigcup_{N' \in \text{sons}(N)} skip'(N', arity(label(N,N'))) \\ skip'(N,0) &:= \{N\} \\ skip'(N,l) &:= \bigcup_{N' \in \text{sons}(N)} skip'(N', l - 1 + arity(label(N,N'))) \end{aligned}$$

With these preliminaries, the functional description of our implementation of discrimination tree term indexing for the Darwin solver may be stated as follows.

Let N be the root node of the term index, t be a candidate term, x be a universal variable,

and p be a parameter. The notation $[x|p]$ is used to indicate that either universal variable x or parameter p may appear in this location. The candidates set for variants of query term t in term index rooted at N is computed by

$$
\begin{aligned}
variants(N, t) &:= var(N, [t]) \\
var(\epsilon, [X]) &:= \emptyset \\
var(N, []) &:= \{N\} \\
\text{if p\_preserving then} & \\
var(N, [x, X]) &:= var(next(N, *U), [X]) \cup var(next(N, *P), [X]) \\
var(N, [p, X]) &:= var(next(N, *P), [X]) \\
\text{else} & \\
var(N, [[x|p], X]) &:= var(next(N, *U), [X]) \cup var(next(N, *P), [X]) \\
\text{end if} & \\
var(N, [a, X]) &:= var(next(N, a), [X]) \\
var(N, [f(t1, ...tn), X]) &:= var(next(N, f), [t1, ..., tn, X])
\end{aligned}
$$

Similarly, let N be the root node of the term index and t be a candidate term, x be a universal variable, and p be a parameter. The candidates set for generalization of the query term t in term index rooted at N is computed by

$$
\begin{aligned}
general(N,t) &:= gen(N,[t]) \\
gen(\epsilon,[X]) &:= \emptyset \\
gen(N,[]) &:= \{N\}
\end{aligned}
$$

if p_preserving then

$$
\begin{aligned}
gen(N,[x,X]) &:= gen(next(N,*U),[X]) \\
gen(N,[p,X]) &:= gen(next(N,*U),[X]) \cup gen(next(N,*P),[X]) \\
gen(N,[a,X]) &:= gen(next(N,*U),[X]) \cup gen(next(N,a),[X]) \\
gen(N,[f(t1,...tn),X]) &:= gen(next(N,*U),[X]) \cup \\
&\quad gen(next(N,f),[t1,...,tn,X])
\end{aligned}
$$

else

$$
\begin{aligned}
gen(N,[[x|p],X]) &:= gen(next(N,*U),[X]) \cup gen(next(N,*P),[X]) \\
gen(N,[a,X]) &:= gen(next(N,*U),[X]) \cup gen(next(N,*P),[X]) \cup \\
&\quad gen(next(N,a),[X]) \\
gen(N,[f(t1,...tn),X]) &:= gen(next(N,*U),[X]) \cup \\
&\quad gen(next(N,*P),[X]) \cup \\
&\quad gen(next(N,f),[t1,...,tn,X])
\end{aligned}
$$

end if

Continuing, let N be the root node of the term index and t be a candidate term, x be a variable, and p be a parameter. The candidates set of instances of the query term t in term index rooted at N is computed by

$$
\begin{aligned}
instances(N,t) &:= inst(N,[t]) \\
inst(\epsilon,[X]) &:= \emptyset \\
inst(N,[]) &:= \{N\}
\end{aligned}
$$

if p_preserving then

$$
\begin{aligned}
inst(N,[x,X]) &:= \bigcup_{M \in skip(N)} inst(M,[X]) \\
inst(N,[p,X]) &:= inst(next(N,*P),[X])
\end{aligned}
$$

else

$$
inst(N,[[x|p],X]) := \bigcup_{M \in skip(N)} inst(M,[X])
$$

end if

$$
\begin{aligned}
inst(N,[a,X]) &:= inst(next(N,a),[X]) \\
inst(N,[f(t1,...tn),X]) &:= inst(next(N,f),[t1,...,tn,X])
\end{aligned}
$$

Finally, let N be the root node of the term index and t be a candidate term, x be a variable, and p be a parameter. The candidates set for unification with the query term t in term index rooted at N is computed by

$$
\begin{aligned}
unifiable(N,t) &:= uni(N,[t]) \\
uni(\epsilon,[X]) &:= \emptyset \\
uni(N,[]) &:= N
\end{aligned}
$$

if p_preserving then

$$
\begin{aligned}
uni(N,[x,X]) &:= \bigcup_{M \in skip(N)} uni(M,[X]) \\
uni(N,[p,X]) &:= uni(next(N,*V),[X]) \cup uni(next(N,*P),[X]) \\
uni(N,[a,X]) &:= uni(next(N,*U),[X]) \cup uni(next(N,a),[X]) \\
uni(N,[f(t1,...tn),X]) &:= uni(next(N,*U),[X]) \cup \\
&\quad uni(next(N,f),[t1,...,tn,X])
\end{aligned}
$$

else

$$
\begin{aligned}
uni(N,[[x|p],X]) &:= \bigcup_{M \in skip(N)} uni(M,[X]) \\
uni(N,[a,X]) &:= uni(next(N,*U),[X]) \cup uni(next(N,*P),[X]) \cup \\
&\quad uni(next(N,a),[X]) \\
uni(N,[f(t1,...tn),X]) &:= uni(next(N,*U),[X]) \cup uni(next(N,*p),[X]) \cup \\
&\quad uni(next(N,f),[t1,...,tn,X])
\end{aligned}
$$

end if

5 Experimental Procedure and Results

With the addition of our discrimination tree term indexing module, the Darwin solver could be run in three configurations: one using no term indexing, one using substitution tree term indexing, and one using discrimination tree term indexing. To compare the performance of these tree configurations, we ran each configuration of the solver against the set of problems used in the CADE-19 Automated Theorem Prover System Completion (CASC-19), see URL: http://www.cs.miami.edu/ tptp/CASC/19/ for details.

As an exploratory experiment, the solution of all problems in the the CASC-19 competition suite was attempted once for each problem using each of the three configurations. These runs were done on a 2.4GHz Pentium IV computer with 1GByte of memory. Each problem was terminated

if it had not completed within 500 CPU seconds. Of the 380 problems in the CASC19 suite, only 106 completed and returned execution data within the allowed 500 CPU seconds. Judging the problems that did not complete in this time to be beyond the capability of our experimental method, we analyzed the data provided by the remaining problems.

A manual review of the results of this experiment revealed that, with a very few exceptions, all three configurations appeared to solve the problem in the same manner, that is, all statistics, such as context size, candidate literals, numbers of application of rule types, etc. were identical for all configurations when run on the same problem. For only two of the problems, the solver with no term indexing appeared to be traversing the solution space in a different manner than both configurations that include term indexing. The output statistics which varied from one configuration to the next were the time required to complete problem solution and the amount of memory used. As the differences in memory usage appeared insignificant when compared to the differences in run time, we analyzed the later differences. Average run times, taken over the set of problems that completed, were 17.46 seconds for the discrimination tree configuration, 17.67 seconds for the substitution tree configuration, and 27.07 seconds for no term indexing. The run time data from this experiment is provided in Appendix A.

Statistical analysis techniques discussed in Snedecor and Cochran [10], Jain [8], and Hinton [7] were applied to the data to test the significance of the difference between these average times. A single factor, repeated measurements analysis of variation of the the raw run times produced by this experiment indicated that using either form of term indexing resulted in significantly shorter run times than not using term indexing ($p < .0001$). When considered as three conditions of a single factor, the difference between term indexing and no term indexing masked the smaller performance difference between the two types of term indexing. Additional analysis, considering only the two configurations that used term indexing indicated the difference between discrimination tree and substitution tree indexing was significant at a confidence level of $p < .01$.

Concern over whether the model assumptions of the standard analysis of variation were met

by the data prompted us to repeat the analysis using two data transformations. These transformation were a logarithmic transformation, meant to test for a multiplicative model, and a ranking transformation, in which the configurations were rated 1, 2, or 3 for each problem with 3 being the fastest configuration. In both of these analyses, the difference between term indexing and no term indexing remained significant at a confidence level $p < .01$, but, the difference between the two term indexing methods no longer proved significant.

A second experiment was conducted to clarify the findings of the initial experiment and to test whether using only a single replication of each problem had affected our results. In the second experiment, seven replications of each problem that had run to completion in the first experiment were attempted using each configuration of the Darwin solver. This experiment was run on a 1.8GHz Pentium IV machine with 512MBytes of memory. In this experiment, only 102 of the attempted problems ran to completion in the allowed 500 CPU seconds. In this experiment, the average times were 15.14 seconds for the discrimination tree configuration, 16.19 seconds for the substitution tree configuration, and 22.86 seconds for no term indexing. Run time data for this experiment is also provided in Appendix A.

Statistical analysis of the data resulting from this experiment indicated that the difference between term indexing and no term indexing was significant ($p < .03$). However, the data did not support a claim of significance for the difference between the two indexing methods.

6 Conclusions and Directions for Future Work

Our work has confirmed the very strong influence including term indexing has on the performance of an automated theorem prover. In this regard, Darwin joins a long list of other theorem proving systems that have reported speed improvements ranging from one to several orders of magnitude based on the use of term indexing [9].

Additionally, our results seem to indicate that on average, for the limited set of problems we ran, discrimination tree term indexing ran faster than substitution tree indexing by a small

percentage. However, additional testing and further statistical analysis are needed to determine whether this difference is significant.

These results make a strong case for reimplementing the standard discrimination tree as a perfect discrimination tree in an attempt to further improve the performance of this term indexing method.

7 Acknowledgments

We would like to thank Cesare Tinelli for the opportunity to participate in the effort to improve the Darwin solver implementation. Special thanks goes also to Alex Fuchs, who's assistance on technical details of the Darwin implementation allowed us to complete this project without becoming throughly familiar with the main body of the Darwin code. We also wish to acknowledge the U.S. Army Materiel Systems Analysis Activity for their generous funding of this work.

References

[1] Peter Baumgartner, Alex Fuchs, and Cesare Tinelli. Darwin: A theorem prover for the model evolution calculus., 22 April 2004. http://www.mpi-sb.mpg.de/ baumgart/publications/darwin.pdf.

[2] Peter Baumgartner and Cesare Tinelli. The model evolution calculus., Jan 2003. http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-1-2003.pdf.

[3] Peter Baumgartner and Cesare Tinelli. The model evolution calculus. In Franz Baader, editor, *Automated Deduction - CADA-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*. Springer, 2003.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[6] Peter Graf. *Term Indexing*. Number 1053 in Lecture Notes in Artificial Intelligence. Springer, New York, NY, 1991.

[7] Perry Hinton. *Statistics Explained: A Guide for Social Science Students*. Routledge, London, UK, 1995.

[8] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, NY, 1991.

[9] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, chapter 26, pages 1853–1962. Elsevier, Amsterdam, NL, 2001.

[10] George Snedecor and William Cochran. *Statistical Methods*. Iowa State Press, Ames, IA, 8 edition, 1989.

[11] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovanbattista Ianni and Sergio 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.

[12] Christoph Wernhard. KRHyper, Oct 2003. URL: http://www.uni-koblenz.de/ wernhard/krhyper/index.html.

Appendixes have been removed to reduce file size.

A Test Data