High-performance deduction for verification: a case study in the theory of arrays

Alessandro Armando
DIST
Università degli Studi di Genova, Italy
armando@dist.unige.it

Maria Paola Bonacina *
Aditya Kumar Sehgal†
Department of Computer Science
The University of Iowa, USA
{*bonacina, asehgal*}*@cs.uiowa.edu*

Silvio Ranise‡
Michaël Rusinowitch
LORIA & INRIA-Lorraine, Villers-lès-Nancy, France
{*Silvio.Ranise, Michael.Rusinowitch*}*@loria.fr*

Abstract

We outline an approach to use ordering-based theorem-proving strategies as satisfiability procedures for certain decidable theories. We report on experiments with synthetic benchmarks in the theory of arrays with extensionality, showing that a theorem prover – the E system – compares favorably with the state-of-the-art validity checker CVC.

1 Introduction

Satisfiability procedures for theories of standard data-types, such as arrays, lists, bit-vectors, are at the core of most state-of-the-art verification tools (e.g., ACL2 [8], PVS [12], Simplify [7], CVC [18]). They are required for a wide range of verification tasks and are fundamental for efficiency. Satisfiability problems have the form $T \cup S$, where S is a set of ground literals (read as conjunction), T is a *background theory*, and the goal is to prove that $T \cup S$ is unsatisfiable.

The endeavour of designing, proving correct, and implementing a satisfiability procedure for each decidable theory of interest is far from simple. First, most problems involve more than one theory, so that one needs to *combine satisfiability procedures* [11, 16]. Combination is complicated: for example, understanding, formalizing and proving correct the method in [16] required significant effort (e.g., [14]). With a theorem prover, one may simply give in input the union of the axiomatizations of the theories. Second, every satisfiability procedure needs to be proved correct and complete: a key ingredient is to show that whenever the algorithm reports "satisfiable," its

*Supported in part by NSF grant CCR-97-01508 and a Dean Scholar Award, The University of Iowa.
†Supported in part by NSF grant CCR-97-01508.
‡Also supported by Università degli Studi di Genova.

ouput represents a model of $T \cup S$. Model-construction arguments can be complex, and the more concrete is the description of the procedure, the more difficult are the proofs (e.g., [14, 19]). If one develops an abstract framework (e.g., [2]), the additional clarity is gained at the expense of proximity with concrete procedures. On the other hand, if a theorem-proving strategy has a sound and refutationally complete inference system, and a fair search plan, it is a semi-decision procedure for unsatisfiability, and we can use it without additional proofs.

Given these attractive features of theorem proving, it would be all the more precious if we could exclude the risk of non-termination on the decidable theories of interest. Results of this nature were presented recently in [1], for the theories of *lists*, *arrays with extensionality*, and their combination, among others. The analysis in [1] showed that a standard, paramodulation-based inference system, for first-order logic with equality, is guaranteed to terminate on $T \cup S$, if T is any of the above theories. The proofs of termination rest on case analyses demonstrating that the inference system can generate only finitely many clauses from such inputs. Thus, a strategy that combines this inference system with a fair search plan is *in itself* a decision procedure for satisfiability in those theories, and a theorem prover that implements it can be used *off the shelf* as a validity checker. One could question whether a specific theorem prover is a sound and complete implementation of such a theorem-proving strategy. However, this question applies also to a validity checker implementing decision procedures, and perhaps even more seriously, considering the common practice of designing and implementing from scratch both data structures and algorithms for each new procedure. In contrast, a deduction-based approach also has potential for better software reuse, since one can envision constructing satisfiability procedures, by combining the generic reasoning modules offered by state-of-the-art theorem provers.

Even after termination has been proved and a higher degree of assurance about the soundness of the procedure can be offered, the issue of efficiency remains. The general expectation is that an implementation of a satisfiability procedure, with the theory built-in as a background theory, will be always much faster than a theorem prover that takes T in input. In this paper, we suggest that this may not be obvious. We consider *synthetic benchmarks*, because they allow to assess the scalability of an approach by experimental asymptotic analysis. We propose two sets of synthetic benchmarks in the *theory of arrays with extensionality*, and we report on experiments with two tools: the E theorem prover [15], and the CVC validity checker [18]. E implements (a variant of) the inference system used in [1] with several search plans. CVC combines decision procedures in the style of [11], as described in [3], including that of [19] for the theory of arrays with extensionality, and featuring either GRASP [17] or Chaff [10] as propositional solver. The experiments show that, for both sets of benchmarks, there is a configuration of the general-purpose prover that is competitive with the validity checker. This is preliminary, encouraging evidence that the approach of [1], in addition to being theoretically elegant, is also applicable in practice.

2 A deduction-based approach and the E prover

The termination results of [1] require that the literals in S be *flat*. This means that the sum of the depths of the sides of an equation, or disequation, has depth at most 1, or at most 0, respectively (assuming constants and variables have depth 0). Literals that are not flat can be flattened by

introducing new constant symbols:

Example 1 *Assume the function symbols store*: ARRAY × INDEX × ELEM*ent* ⟶ ARRAY *and select*: ARRAY × INDEX ⟶ ELEM*ent denote the operations of storing and retrieving a value at a position in an array, respectively (e.g., [11]). The ground literals in* $S = \{store(s, a, v) = store(s_1, a_1, v_1);\ select(s, a) = v;\ select(s_1, a_1) = v_1;\ a = a_1;\ v \neq v_1\}$ *can be flattened by introducing new constants* c_1, c_2, c_3, c_4, *yielding* $S' = \{store(s, a, v) = c_1;\ store(s_1, a_1, v_1) = c_2;\ select(s, a) = c_3;\ select(s_1, a_1) = c_4;\ c_1 = c_2;\ c_3 = v;\ c_4 = v_1;\ a = a_1;\ v \neq v_1\}$. *This trasformation preserves satisfiability:* $T \cup S$ *is satisfiable if and only if* $T \cup S'$ *is, for arbitrary* T.

Flattening can be done in different ways: we call it *strict*, if all occurrences of a subterm are replaced by the same constant, and *non-strict*, if each subterm occurrence is replaced by a new constant. Non-strict flattening yields an under-constrained problem, whose unsatisfiability obviously implies unsatisfiability of the strictly flattened version. Strict flattening minimizes the number of new constants introduced, hence the number of clauses, "sharing" subterms as much as possible. A non-strictly flattened version has less sharing of subterms and more clauses. After this pre-processing, $T \cup S$ can be given to any fair theorem-proving strategy with the following inference system (e.g., [1]): *ordered superposition/paramodulation*, *reflection*, and *(ordered) equality factoring*, as expansion inference rules, and *subsumption*, *simplification* and *deletion* (of trivial equations), as contraction inference rules.

Like most ordering-based provers, E implements search plans based on the *given-clause loop* [9]. The prover works with two lists of clauses, say *To-be-selected* and *Already-selected*: at every iteration, it extracts a clause, the *given clause*, from *To-be-selected*, moves it to *Already-selected*, performs all expansion inferences between the given clause and clauses in *Already-selected*, and appends the normal forms of all new clauses thus generated to *To-be-selected*. Practical ordering-based strategies require that contraction be applied *eagerly*, to avoid generating clauses from clauses that can be deleted by contraction. Variants of the given-clause loop differ in the implementation of eager contraction: while the Otter version aims at keeping the union of *To-be-selected* and *Already-selected* inter-reduced, E implements a version that keeps only *Already-selected* inter-reduced, on the ground that all parents of expansion inferences are in *Already-selected*, with the downside that clauses in *To-be-selected* are not applied as simplifiers.

The features of the search plans in E that were most relevant to our experiments are *clause selection* and *term ordering*, and *literal selection* to a lesser extent. For clause selection, given a heuristic evaluation function f, the given-clause loop implements a *best-first search*, by selecting at each iteration a clause C such that $f(C)$ is minimum. E uses pairs of functions (f_1, f_2), where f_1 is the *clause priority function* and f_2 the *heuristic weight function*, to pick a clause of smallest weight among those of highest priority. Ties are broken by selecting the oldest clause. Every pair (f_1, f_2) defines a priority queue: E allows the user to activate and weight any number of them, resulting in a weighted round robin scheme.

Considering that our problems have the form $T \cup S$, one may think of a *set-of-support strategy*, with T as consistent set and S as set of support. However, E does not emphasize supported strategies, because using a set of support is complete for resolution, but not for (ordered) resolution and paramodulation, unless T is saturated. If T is saturated, by definition, all inferences from T

are redundant, and therefore using its complement as set of support does not add focus to the search: e.g., T may generate very few clauses that are subsumed right away. This is the case for the first presentation considered in our experiments (named T_1 and introduced in Section 3): its two axioms generate only one trivial clause. The second presentation we used (named T_2 and also introduced in Section 3), is not saturated, so that one could consider using T_2 as consistent set and S as set of support, since incomplete strategies are often used in experiments. However, E, unlike Otter, does not let the experimenter choose the input set of support: with its clause priority function `SimulateSOS`, which prefers supported clauses, the set of support is initialized by the prover to contain the input negative clauses. Nevertheless, we chose to use it.

Among weight functions, we tried both `Clauseweight` and `Refinedweight`. The former, e.g., `Clauseweight(x,y,z)`, is the number of symbols in the clause, with weights `x` for non-variable symbols and `y` for variable symbols, and the resulting weight of each positive literal multiplied by `z`. `Refinedweight(x,y,z,w,t)` is similar, but aims at taking the term ordering into account, by multiplying the resulting weight of each maximal term and maximal (or selected) literal by `w` and `t`, respectively. The *term ordering* is the ordering on terms and literals used for well-founded rewriting and to restrict paramodulation/superposition. E implements *Knuth-Bendix ordering* (KBO), and *lexicographic path ordering* (LPO) (e.g., [6]), and we experimented with both. These orderings require a *precedence* on function symbols that can be given by the user, built by the prover, or a combination of the two. KBO also requires to weight the symbols: by default, E assigns all symbols weight 1, except the first non-constant maximal symbol which gets weight 0. The *literal selection* functions select literals in clauses to restrict ordered paramodulation further. We tried a few and settled for `SelectComplex`: it selects the first literal in the form $x \neq y$; if the clause has none, it picks the smallest ground negative literal; if the clause has none, it picks an arbitrary negative literal among those with the largest difference in number of symbols between left and right side. In *automatic mode*, E determines automatically clause evaluation function, term ordering and literal selection function for the given input.

3 Synthetic benchmarks in the theory of arrays

The presentation of the theory of arrays with extensionality is given by the following axioms:

$$\forall A, I, E.\ select(store(A, I, E), I) = E \tag{1}$$

$$\forall A, I, J, E.\ (I \neq J \Longrightarrow select(store(A, I, E), J) = select(A, J)) \tag{2}$$

$$\forall A, B.\ (\forall I. select(A, I) = select(B, I) \Longrightarrow A = B) \tag{3}$$

where A and B are variables of sort ARRAY, I and J are variables of sort INDEX, and E is a variable of sort ELEMent. The clausal forms of axioms (1) and (2) are given in input to the prover together with the ground literals of the specific problem. Axiom (3), the *extensionality axiom*, is a universal-existential formula of sorted first-order logic with equality, which is not given to the prover, but handled by pre-processing the input set of ground literals [1]. This pre-processing step consists of replacing every disequality of the form $t \neq t'$, where t and t' are terms of sort ARRAY, by the disequality $select(t, sk(t, t')) \neq select(t', sk(t, t'))$, where sk is a skolem function of type ARRAY $\times$ ARRAY $\longrightarrow$ INDEX. Indeed, this is the result of applying a resolution step to $t \neq t'$ and

the clausal form of axiom (3), $select(A, sk(A,B)) \neq select(B, sk(A,B)) \vee A = B$. Unsatisfiability in the theory is clearly preserved. Intuitively, $sk(t,t')$ is an index where the arrays t and t' differ.

An alternative axiomatization of the theory of arrays with extensionality (e.g., [11]), leaves axioms (1) and (2) unchanged, and replaces (3) by:

$$\forall A, I.\ store(A, I, select(A, I)) = A \tag{4}$$

$$\forall A, I, E, F.\ store(store(A, I, E), I, F) = store(A, I, F) \tag{5}$$

$$\forall A, I, J, E.\ (I \neq J \Longrightarrow store(store(A, I, E), J, F) = store(store(A, J, F), I, E)) \tag{6}$$

where A is a variable of sort ARRAY, I and J are variables of sort INDEX, and E and F are variables of sort ELEMent. We shall refer to the first axiomatization as T_1 and to the second one as T_2. An axiomatization for *finite maps* similar to T_2 was given in [5] together with a model built in HOL: it includes axioms (1), (2), (5), (6), plus an induction principle that allows one to derive (3) and (4) as theorems. One can easily prove by hand that T_1 entails T_2, so that if $T_2 \cup S$ is unsatisfiable, $T_1 \cup S$ is unsatisfiable also. T_2 was not used in [1] and there is no termination result for this presentation. Not surprisingly, saturation of T_2, that we tried on the side of our experiments with E, did not terminate. Thus, when working with T_2, the theorem-proving strategy acts as a semi-decision procedure, taking in input the clausal form of $T_2 \cup S$.

We present two sets of synthetic benchmarks for this theory. For the first one, the idea is to express the "commutativity" of storing elements at distinct places in an array a. Let $\{k_1, \ldots, k_N\}$ be N indices and C_2^N denote the set of 2-combinations over $\{1, \ldots, N\}$. To say that they are distinct, we write $\bigwedge_{(p,q)\in C_2^N} k_p \neq k_q$: e.g., for $N = 3$, $k_1 \neq k_2 \wedge k_1 \neq k_3 \wedge k_2 \neq k_3$. Then, if $i_1, ..., i_N$ and $j_1, ..., j_N$ are two distinct permutations of $1, \ldots, N$, the equation $store(\ldots(store(a, k_{i_1}, e_{i_1}), \ldots k_{i_N}, e_{i_N})\ldots) = store(\ldots(store(a, k_{j_1}, e_{j_1}), \ldots k_{j_N}, e_{j_N})\ldots)$ captures the desired property. For example, for $N = 3$, and permutations $(1,2,3)$ and $(2,1,3)$, we get $store(store(store(a, k_1, e_1), k_2, e_2), k_3, e_3) = store(store(store(a, k_2, e_2), k_1, e_1), k_3, e_3)$. Altogether we have the following schema:

$$(\textstyle\bigwedge_{(p,q)\in C_2^N} k_p \neq k_q) \Rightarrow$$
$$store(\ldots(store(a, k_{i_1}, e_{i_1}), \ldots k_{i_N}, e_{i_N})\ldots) = store(\ldots(store(a, k_{j_1}, e_{j_1}), \ldots k_{j_N}, e_{j_N})\ldots).$$

Each choice of permutations generates a different instance of the schema, and since there are $N!$ permutations of $\{k_1, \ldots, k_N\}$, the number of instances is the number of 2-combinations of $N!$ permutations, hence $\binom{N!}{2}$ or $(N!(N!-1))/2$. In our experiments, for each value of N, we sampled at most 10 permutations, hence 45 instances, in order to reduce the dependence of the results on the structure of the formula. We use $storecomm(N)$ to denote the generated instances for size N and the problem of checking their validity, or the unsatisfiability of their negation.

For the second group, the intuition is that swapping pairs of elements in an array a in two different orders yields the same array. The equations can be defined recursively. In the base case, $p = 0$, $k = 2p = 0$, and for $N = k + 2 = 2$ elements, the equation is $L_2 = R_2$, where

$$L_2 = store(store(a, i_1, select(a, i_0)), i_0, select(a, i_1))$$
$$R_2 = store(store(a, i_0, select(a, i_1)), i_1, select(a, i_0))$$

assuming $L_0 = R_0 = a$. In the recursive case, for any $p > 0$, $k = 2p$, the number of elements swapped is $N = k + 2$, and the equation is $L_{k+2} = R_{k+2}$, where

$$L_{k+2} = store(store(L_k, i_{k+1}, select(L_k, i_k)), i_k, select(L_k, i_{k+1}))$$
$$R_{k+2} = store(store(R_k, i_k, select(R_k, i_{k+1})), i_{k+1}, select(R_k, i_k)).$$

For example, for $N = 4$ ($k = 2$), we get $L_4 = R_4$ with

$$L_4 = store(store(L_2, i_3, select(L_2, i_2)), i_2, select(L_2, i_3))$$
$$R_4 = store(store(R_2, i_2, select(R_2, i_3)), i_3, select(R_2, i_2)).$$

For every N we get different instances by choosing different permutations of the operations, e.g., for $N = 4$, we can also generate $L'_4 = R'_4$, where $L'_4 = L_4$, and

$$R'_4 = store(store(R_2, i_3, select(R_2, i_2)), i_2, select(R_2, i_3)).$$

The above recursive definition only generates equations where all the pairs are exchanged. We also consider instances where only some of the pairs are exchanged. Thus, for N elements, there are $N!$ permutations, and $N!(2^{N/2} - 1)$ instances, where $2^{N/2} - 1$ is obtained from $\Sigma_{i=1}^{N/2}\binom{N/2}{i} = 2^{N/2} - 1$. Indeed, $\binom{N/2}{i}$ is the number of i-combinations over the set of $N/2$ pairs, or the number of ways of picking i pairs (for exchanging them) out of $N/2$. This expression counts each equation twice (e.g., $L = R$ and $R = L$), so that the number of distinct instances is $1/2(N!(2^{N/2} - 1))$. In our experiments, for each value of N, we sampled at most 16 permutations and 20 instances. We use $swap(N)$ to denote both the set of generated instances and the corresponding problem.

4 The experiments with E and CVC

We ran E (version 0.62 "Mullootar") and CVC on a dual AMD Athlon 1.2GHz machine, with 512MB of RAM, running Linux 2.4.7. We used both CVC/GRASP and CVC/Chaff, because they are two different versions of CVC, the latter released later. The SAT solver should not play a role with the theory of arrays. For all experiments, we gave precedence $select \succ store \succ sk$, completed by E by making all constant symbols smaller than the function symbols. As an exercise, we tried eight problems in the theory of arrays from the SVC distribution (SVC was CVC's predecessor: `http://sprout.Stanford.EDU/SVC/`). E in automatic mode solved each problem in 0.01 sec or less, with or without flattening, and CVC did each problem in approximately 0.04 sec.

We wrote a Prolog program that, given N, generates the instances of $storecomm(N)$ and $swap(N)$, in either E-LOP syntax, the Prolog-like syntax of E, or CVC syntax; it applies flattening for the experiments with E, and pre-processes the generated equations with respect to extensionality, for the experiments with E and T_1. Different instances of the same problem may have different numbers of distinct subterms: since the latter determines the number of new constants introduced by flattening, and each new constant is defined by an equation, different instances yield sets of equations of different size. The reported performance of a system on $storecomm(N)$, or $swap(N)$, is the *average* performance over all generated instances for size N.

We start with $storecomm(N)$, with presentation T_1 and strict flattening for the input to E. In Figure 1, E-Auto refers to E in automatic mode, while E-SOS refers to the plan `(SimulateSOS,Refinedweight(2,1,2,1,1))`, with selection function `SelectComplex` and ordering LPO, which was

among the best we observed. The curve for E-SOS is just a bit above those for CVC/GRASP and CVC/Chaff up to $N = 110$, then crosses them, and stays clearly below them for $N > 110$. The curve for E-Auto remains above the others, and is very smooth, which appears a welcome sign of regularity, considering how theorem provers have been often considered very sensitive to even minor input variations. Altogether, the theorem prover fared very well.

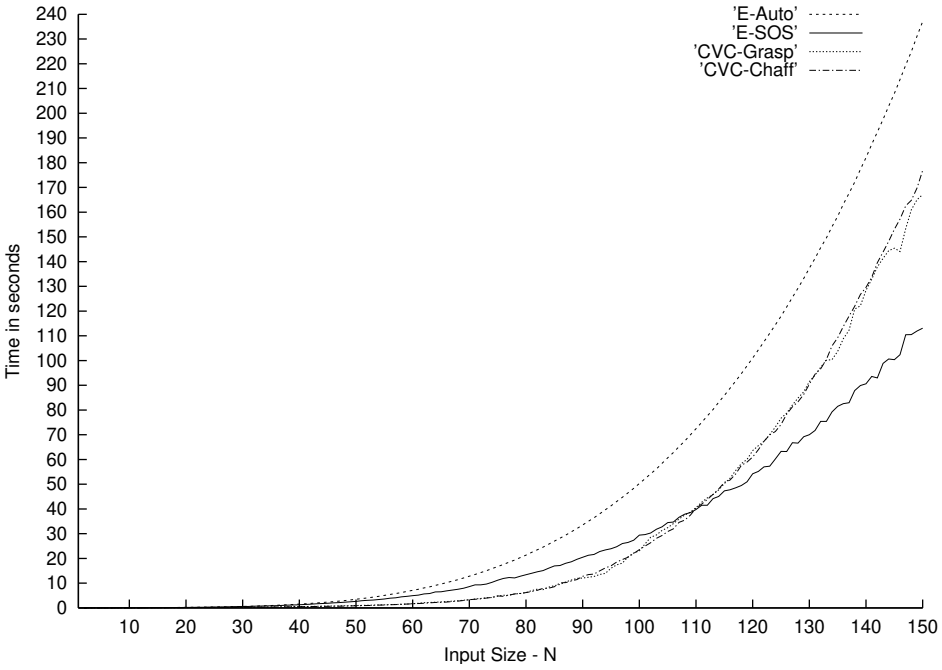

Figure 1: Behavior of E, with presentation T_1, and CVC on $storecomm(N)$, for N ranging from 2 to 150.

For $swap(N)$, we report the data in Table 1, because N ranges only from 2 to 10, since both E, with presentation T_1, and CVC, with either GRASP or Chaff, ran out of memory on any instance of $swap(12)$. We tried both strict and non-strict flattening, and E did best with the latter and a slight modification of the above search plan: `(SimulateSOS,Refinedweight(3,2,3,2,1))` with `SelectComplex` and KBO. With the exception of $swap(2)$, CVC performed better than E by one order of magnitude. The outcome is strikingly different, however, if we give T_2 in input to E, while using strict flattening. Figure 2 compares the performance of E on this input with that of CVC-Chaff from Table 1. Both E-Auto and E-SOS terminate successfully also for $N \geq 12$: the curve for E-SOS (with LPO and weights as for $storecomm(N)$) grows extremely slowly, while that for E-Auto is much higher but still smooth for the most part.

When we submitted in error redundant versions of $storecomm(N)$, E surpassed CVC sooner (in Figure 1, the E-SOS curve was below the CVC curves for $N \geq 60$ instead of $N \geq 110$). This suggests that E may be better than CVC in deleting redundant data. The algorithm of [19], in essence, pre-processes the input problem with respect to the axioms in T_1, eliminates the occurrences of *store* by recurring to *partial equations*, and computes a congruence closure. Thus, there might not be a provision to eliminate redundant equations, as theorem provers do by contraction. For E, the presentation T_2 may represent more information than T_1 with pre-processing with respect to extensionality, so that the prover behaves better on $swap(N)$, even if

N	E-Auto	E-SOS	CVC-GRASP	CVC-Chaff
2	0.010	0.010	0.043	0.057
4	0.144	0.136	0.059	0.060
6	2.205	2.110	0.189	0.187
8	63.630	62.230	4.091	2.400
10	2069.700	2039.700	1297.000	95.780

Table 1: Behavior of E with presentation T_1 and CVC on $swap(N)$.

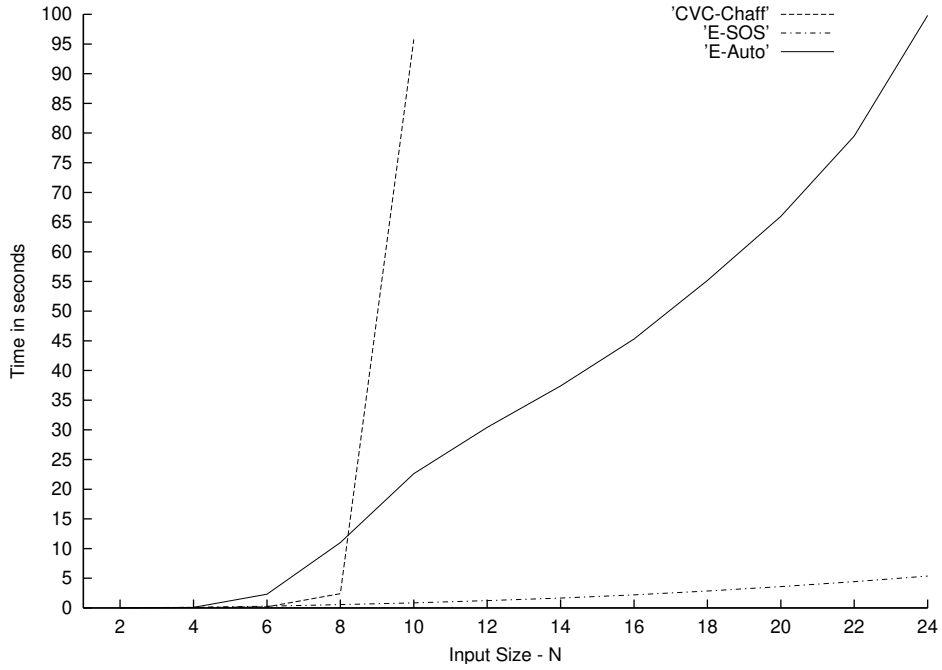

Figure 2: Behavior of E, with presentation T_2, and CVC on $swap(N)$, for N ranging from 2 to 24.

T_2 is not saturated and the prover is acting as a semi-decision procedure. We regard this behavior as evidence of the flexibility of an approach based on general-purpose deduction. Both E and CVC come with *proof checkers*. The E distribution features two tools, *e2pcl*, to extract a proof object from E's output, and *checkproof*, to check it by using another prover. While we did not use *flea*, the proof checker for CVC [20], we tried *e2pcl*, *checkproof* and Otter on sample outputs of E, and no error was detected.

5 Discussion

We tested the usage of theorem-proving strategies as decision procedures, on synthetic benchmarks in the theory of arrays with extensionality. The results indicate that this investigation should continue, and we envision several directions, in experimentation, implementation, and theory. For experimentation, we intend to work with more synthetic benchmarks (e.g., in the theory of

extensional finite sets, also covered in [1]), real-world problems (e.g., completing those in [4], already successfully started in [1]), problems involving other theories (e.g., for other data-types [13]) and *combinations of theories*. We also plan to conduct more experiments to understand the role of *flattening* better. Flattening aids by inducing a fully shared representation of terms as *dags*, which has been traditionally considered an advantage of congruence closure, and by making terms *shallow*, while increasing the number of equations. The first factor might not play such a key role for E, however, since E represents terms internally as *perfectly shared terms* regardless of the input [15]. The second factor might, since shallow terms simplify *matching* and all *term indexing* operations. The impact of the additional equations should be negligible, since provers are designed to handle millions, and a prover that inter-reduces its input uses only then an equation reducing a complex term to a constant. Surprisingly, and unlike Otter, E does not inter-reduce its input before starting the search.

It would be interesting to try other provers, especially those that implement the Otter version of the given-clause loop, to see whether a broader application of *eager contraction* helps in these problems. In our experiments, the difference between automatic mode and user-selected search plan had a visible impact on asymptotic behavior. This, together with the need of reducing the time spent testing search plans, invites more work on the *automatic mode* of provers, and more attention to *search plan design*. We felt at times that architecture and presentation of contemporary provers overemphasize blind saturation at the expense of search control.

Directions for theoretical research include termination and complexity results for more decidable theories. Satisfiability of a conjunction of literals in the theory of arrays with extensionality is NP-complete [19]. The algorithm of [19] has worst-case time complexity $O(2^{n \log n})$, where n is the size of the set of literals. For the deduction-based approach, the upper bound on the number of clauses that can be generated from $T \cup S$, where T contains the first two axioms of the theory and S is a set of flat equational literals, pre-processed with respect to extensionality, is $O(2^{n^2})$ [1]. However, this analysis refers to saturation, and does not take any search plan into account. The *complexity of theorem-proving strategies*, defined as the combination of inference system and search plan, is still largely unexplored, primarily because the underlying problem is only semi-decidable in the general first-order case. Results such as those of [1] exclude infinite derivations, and open the way to studying the complexity of concrete theorem-proving strategies for specific decidable theories. For theories where termination of saturation may not be proved, one may investigate obtaining a decision procedure by *integrating* theorem proving and model building. Indeed, the perspective of *system integration* encompasses all these directions: since one of the motivations for studying decision procedures is to integrate them into proof assistants, using theorem-proving strategies as decision procedures goes in the direction of fostering the integration of proof assistants and theorem provers.

Acknowledgements We would like to thank Stephan Schulz, for making E available and answering our questions on his prover, and the anonymous referees for their comments.

References

[1] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, to appear, 2002.

[2] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, to appear, 2002.

[3] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In David McAllester, editor, *Proc. CADE-17*, volume 1831 of *LNAI*, pages 79–97. Springer, 2000.

[4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proc. CAV-6*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.

[5] Graham Collins and Donald Syme. A theory of finite maps. In *Proc. TPHOLs*, LNCS. Springer, 1995.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[7] D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. Technical report, DEC, 1996.

[8] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Eds. *Computer Aided Reasoning : ACL2 Case Studies*. Kluwer, 2000.

[9] William W. McCune. Otter 3.0 reference manual and guide. Technical Report 94/6, MCS Division, Argonne National Laboratory, 1994. See also the web page `http://www-unix.mcs.anl.gov/AR/otter/`.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 39th Design Automation Conf.*, 2001.

[11] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

[12] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Proc. CADE-11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[13] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. See also the web page `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/jcr/www/`.

[14] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *Proc. LICS-16*. IEEE, 2001.

[15] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 2002. See also the web page `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.

[16] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[17] João Marques Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[18] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a Cooperating Validity Checker. In Kim G. Larsen and Ed Brinksma, editors, *Proc. CAV-14*, volume to appear of *LNCS*. Springer, 2002. See also the web page `http://verify.stanford.edu/CVC/`.

[19] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proc. LICS-16*. IEEE, 2001.

[20] Aaron Stump and David L. Dill. Faster proof checking in the Edinburgh Logical Framework. In Andrei Voronkov, editor, *Proc. CADE-18*, volume to appear of *LNAI*. Springer, 2002.