An Experimental Evaluation of Ground Decision Procedures [⋆]

(Submitted for Publication, January 25, 2003)

Leonardo de Moura and Harald Rueß

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{demoura, ruess}@csl.sri.com
phone +1 650 859-6136, fax +1 650 859-2844

Abstract. There is a large variety of algorithms for ground decision procedures, but their differences, in particular in terms of experimental performance, are not well studied. We develop maps of the behavior of ground decision procedures by comparing the performance of a variety of technologies on benchmark suites with differing characteristics. Based on these experimental results, we discuss relative strengths and shortcomings of different systems.

1 Introduction

Decision procedures are an enabling technology for a growing number of applications of automated deduction in hardware and software verification, planning, bounded model checking, and other search problems. In solving these problems, decision procedures are used to decide the validity of propositional constraints such as

$$z = f(x - y) \wedge x = z + y \rightarrow -y = -(x - f(f(z))) \ .$$

This formula is in the combination of linear arithmetic and an uninterpreted function symbol f. Since all variables are considered to be universally quantified, they can be treated as (Skolem) constants, and hence we say that the formula is *ground*.

Solving propositional formulas with thousands of variables and hundreds of thousands of literals, as required by many applications, is a challenging problem. Over the last couple of years, many different algorithms, heuristics, and tools have been developed in addressing this challenge. Most of these approaches either use extensions of binary decision diagrams [1], or an equisatisfiable reduction of propositional constraint formulas to propositional logic, which is solved using

[⋆] Funded by SRI International, by NSF Grants CCR-0082560, EIA-0224465, and CCR-0326540, DARPA/AFRL-WPAFB Contract F33615-01-C-1908, and NASA Contract B0906005.

a Davis-Putnam [2] search procedure. Their differences, especially in terms of experimental performance, are not well studied.

Ultimately, it is our goal to develop comprehensive maps of the behavior of ground decision procedures by comparing the performance of a variety of technologies on benchmark suites with differing characteristics, and the study in this paper is an essential first step towards this goal of comprehensive benchmarkmarking. We have been collecting a number of existing benchmarks for ground decision procedures and applied them to the CVC/CVCLite [3], ICS [4], UCLID [5], MathSAT [6], Simplify [7], and SVC [8] systems. This required developing translators from the various benchmark formats to the input languages of the systems under consideration. For the diversity of the algorithms underlying ground decision procedures, we only measure runtimes. We analyze the computational characteristics of each system on each of the benchmarks, and expose specific strengths and weaknesses of the systems under consideration.

Since such an endeavor should ultimately develop into a cooperative ongoing activity across the field, our main emphasis in conducting these experiments is not only on reproducibility but also on reusability and extendibility of our experimental setup. In particular, all benchmarks and all translators we developed for converting input formats, and all experimental results are publicly available at `http://www.csl.sri.com/users/demoura/gdb-benchmarks.html`.

This paper is structured as follows. In Section 2 we include a brief overview on the landscape of different methods for ground decision procedures, and in Section 3 we describe the decision procedures, benchmarks, and our experimental setup. Section 4 contains our experimental results by pairwise comparing the behavior of decision procedures on each benchmark set, and in Section 5 and 6 we summarize some general observations from these experiments and provide final conclusions.

2 Ground Decision Procedures

We consider *ground decision procedures* (GDP) for deciding propositional satisfiability for formulas with literals drawn from a given constraint theory $\mathcal{T}$. GDPs are usually obtained as extensions of decision procedures for the satisfiability problem of conjunctions of constraints in $\mathcal{T}$. For example, satisfiability for a conjunction of equations and disequations over terms in U is decidable in $O(n\log(n))$ using congruence closure [9]. Conjunctions of constraints in rational linear arithmetic are solvable in polynomial time, although many algorithms such as Simplex have exponential worst-case behavior. In the theory $\mathcal{D}$ of *difference logic*, arithmetic constraints are restricted to constraints of the form $x - y \leq c$ with c a constant. An $O(n^3)$ algorithm for the conjunction of such constraints is obtained by searching, using the Bellman-Ford algorithm, for negative-weight cycles in the graph with variables as nodes and an edge of weight c from x to y for each such constraints. For individual theories $\mathcal{T}_i$ with decidable satisfiability problems, the union of all $\mathcal{T}_i$'s is often decided using a Nelson-Oppen [10] or a Shostak-like [11,12] combination algorithm.

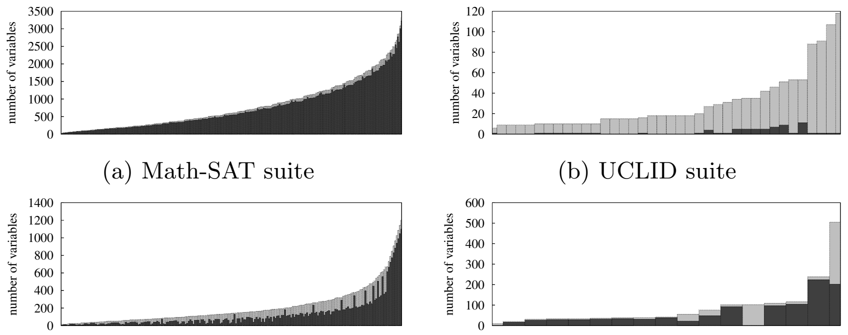
ICS	`ulimit -s 30000; ics` *`problem-name`*`.ics`
UCLID	`uclid` *`problem-name`*`.ucl sat 0 zchaff`
CVC	`cvc +sat <` *`problem-name`*`.cvc`
CVC Lite	`cvcl +sat fast <` *`problem-name`*`.cvc`
SVC	`svc` *`problem-name`*`.svc`
Simplify	`Simplify` *`problem-name`*`.smp`
Math-SAT	`mathsat_linux` *`problem-name`*`.ms -bj math -heuristic SatzHeur`

Table 1. Command line options used to execute GDPs.

Given a procedure for deciding satisfiability of conjunctions of constraints in $\mathcal{T}$ it is straightforward to decide propositional combinations of constraints in $\mathcal{T}$ by transforming the formula into disjunctive normal form, but this is often prohibitively expensive. Better alternatives are to extend binary decision diagrams to include constraints instead of variables (e.g. difference decision diagrams), or to reduce the propositional constraint problem to a purely propositional problem by encoding the semantics of constraints in terms of added propositional constraints (see, for example, Theorem 1 in [13]). Algorithms based on this latter approach are characterized by the eagerness or laziness with which constraints are added.

In *eager* approaches to constructing a GDP from a decision procedure for $\mathcal{T}$, propositional constraints formulas are transformed into equisatisfiable propositional formulas. In this way, Ackermann [14] obtains a GDP for the theory U by adding all possible instances of the congruence axiom and renaming uninterpreted subterms with fresh variables. In the worst case, the number of such axioms is proportional to the square of the length of the given formula. Other theories such as S-expressions or arrays can be encoded using the reductions given by Nelson and Oppen [10]. Variations of Ackermann's trick have been used, for example, by Shostak [15] for arithmetic reasoning in the presence of uninterpreted function symbols, and various reductions of the satisfiability problem of Boolean formulas over the theory of equality with uninterpreted function symbols to propositional SAT problems have recently been described [16], [17], [18]. In a similar vein, an eager reduction to propositional logic works for constraints in difference logic [19].

In contrast, *lazy* approaches introduce part of the semantics of constraints on demand [13], [20], [6], [21]. Let ϕ be the formula whose satisfiability is being checked, and let L be an injective map from fresh propositional variables to the atomic subformulas of ϕ such that $L^{-1}[\phi]$ is a propositional formula. We can use a propositional SAT solver to check that $L^{-1}[\phi]$ is satisfiable, but the resulting truth assignment, say $l_1 \wedge \ldots \wedge l_n$, might be spurious, that is $L[l_1 \wedge \ldots \wedge l_n]$ might not be ground-satisfiable. If that is the case, we can repeat the search with the added *lemma* clause $(\neg l_1 \vee \ldots \vee \neg l_n)$ and invoke the SAT solver on $(\neg l_1 \vee \ldots \vee \neg l_n) \wedge L^{-1}[\phi]$. This ensures that the next satisfying assignment returned is different from the previous assignment that was found to be ground-unsatisfiable. In such an *offline* integration, a SAT solver and a constraint solver can be used as black boxes. In contrast, in an *online* integration the search for satisfying



(a) Math-SAT suite (b) UCLID suite

(c) SAL suite (d) SEP suite

Fig. 1. Number of boolean (dark gray) and non-boolean (light gray) variables in each problem.

assignments of the SAT solver is synchronized with constructing a corresponding logical context of the theory-specific constraint solver. In this way, inconsistencies detected by the constraint solver may trigger backtracking in the search for satisfying propositional assignments. An effective *online* integration can not be obtained with a black-box decision procedures but requires the constraint solver to process constraints *incrementally* and to be *backtrackable* in that not only the a current logical context is maintained but also contexts corresponding to backtracking points in the search for satisfying assignments. The basic refinement loop in the *lazy* integration is usually accelerated by considering negations of minimal inconsistent sets of constraints or "good" over-approximations thereof. These so-called *explanations* are either obtained from an explicitly generated proof object or by tracking dependencies of facts generated during constraint solver runs.

3 Experimental Setup

We describe the setup of our experiments including the participating systems, the benchmarks and their main characteristics, and the organization of the experiments itself. This setup has been chosen before conducting any experiments.

3.1 Systems.

The systems participating in this study implement a wide range of different satisfiability techniques as described in Section 2. All these GDPs are freely available and distributed, and in each case we have been using the latest version (as of January 10, 2004). We provide short descriptions in alphabetical order.

The Cooperating Validity Checker (CVC 1.0a) [3] is a GDP for the combination of theories including linear real arithmetic $\mathcal{A}$, uninterpreted function symbols $\mathcal{U}$, functional arrays, and inductive datatypes. Propositional reasoning is obtained by means of a lazy, online integration of the zChaff SAT solver and a constraint solver based on a Shostak-like [11] combination, and explanations are obtained as axioms of proof trees. We also consider the successor system CVC Lite (version 1.0.1), whose architecture is similar to the one of CVC, but it uses a home-grown SAT solver for realizinig a tighter integration between SAT and constraint solving.

The Integrated Canonizer and Solver (ICS 2.0) [4] is a GDP for the combination of theories including linear real arithmetic $\mathcal{A}$, uninterpreted function symbols $\mathcal{U}$, functional arrays, S-expressions, products and coproducts, and bitvectors. It realizes a lazy, online integration of a non-clausal SAT solver with an incremental, backtrackable constraint engine based on a Shostak [11] combination. Explanations are generated and maintained using a simple tracking mechanism.

UCLID (version 1.0) is a GDP for the combination of difference logic and uninterpreted function symbols. It uses an eager transformation to SAT problems, which are solved using zChaff [5]. The use of other theories such as lambda expressions and arrays is restricted in order to eliminate them in preprocessing steps.

Math-SAT [6] is a GDP for linear arithmetic based on a black-box constraint solver, which is used to detect inconsistencies in constraints corresponding to partial Boolean assignments in an offline manner. The constraint engine uses a Bellman-Ford algorithm for difference logic constraints and a Simplex algorithm for more general constraints.

Simplify [7] is a GDP for linear arithmetic $\mathcal{A}$, uninterpreted functions $\mathcal{U}$, and arrays based on the Nelson-Oppen combination. Simplify uses home-grown SAT solving techniques, which do not incorporate many efficiency improvements found in modern SAT solvers. However, Simplify goes beyond the other systems considered here in that it includes heuristic extensions for quantifier reasoning, but this feature is not tested here.

Stanford Validity Checker (SVC 1.1) [8] decides propositional formulas with uninterpreted function symbols $\mathcal{U}$, rational linear arithmetic $\mathcal{A}$, arrays, and bitvectors. The combination of constraints is decided using a Shostak-like combination extended with binary decision diagrams.

3.2 Benchmark suites

We have included into this study freely distributed benchmark suites for GDPs with constraints in $\mathcal{A}$ and $\mathcal{U}$ and combinations thereof. These problems range from standard timed automata examples to equivalence checking for microprocessors and the study of fault-tolerant algorithms. For the time being we do not separate satisfiable and unsatisfiable instances. Since some of the benchmarks are distributed in clausal and other in non-clausal form, it is difficult to provide measures on the difficulty of these problems, but Figure 1 contains the number of variables for each benchmark problem. The dark gray bars represents the number of boolean and the light gray bars the number of non-boolean variables.

The Math-SAT benchmark suite (`http://dit.unitn.it/~rseba/Mathsat.html`) is composed of timed automata verification problems. The problems are distributed only in clausal normal form and arithmetic is restricted to coefficients in $\{-1, 0, 1\}$. This benchmark format also includes extra-logical *search hints* for the Math-SAT system. This suite comprises 280 problems, 159 of which are in the difference logic fragment. The size of the ASCII representation of these problems ranges from 4Kb to 26Mb. As can be seen in Figure 1(a) most of the variables are boolean.[1]

Benchmark suite	Sat	Unsat	Unsolved
Math-Sat	37	224	19
UCLID	0	36	2
SAL	21	167	29
SEP	9	8	0

Table 2. Classification.

The UCLID benchmark suite (`http://www-2.cs.cmu.edu/~uclid`) is derived from processor and cache coherence protocol verifications. This suite is distributed in the SVC input format. In particular, propositional structures are non-clausal and constraints include uninterpreted functions $\mathcal{U}$ and difference constraints $\mathcal{D}$. Altogether there are 38 problems, the size of the ASCII representation ranges from 4Kb to 450Kb, and the majority of the literals are non-boolean (Figure 1(b)).

The SAL benchmark suite (`http://www.csl.sri.com/users/demoura/gdb-benchmarks.html`) is derived from bounded model checking of timed automata and linear hybrid systems, and from test-case generation for embedded controllers. The problems are represented in non-clausal form, and constraints are in full linear arithmetic. This suite contains 217 problems, 110 of which are in the difference logic fragment, the size of the ASCII representation of these problems ranges from 1Kb to 300Kb. Most of the boolean variables are used to encode control flow (Figure 1(c)).

[1] We suspect that many boolean variables have been introduced through conversion to CNF.

		ICS	UCLID	CVC	CVC Lite	SVC	Simplify	Math-SAT
Math-SAT	timeout	0	3	0	19	50	91	52
suite	aborts	22	21	58	61	39	0	0
UCLID	timeout	4	2	0	9	5	24	n/a
suite	aborts	4	0	14	9	0	0	n/a
SAL	timeout	1	36	1	115	87	99	n/a
suite	aborts	29	4	64	7	4	4	n/a
SEP	timeout	0	0	0	1	1	0	n/a
suite	aborts	0	1	1	1	0	0	n/a

Table 3. Number of timeouts and aborts for each system.

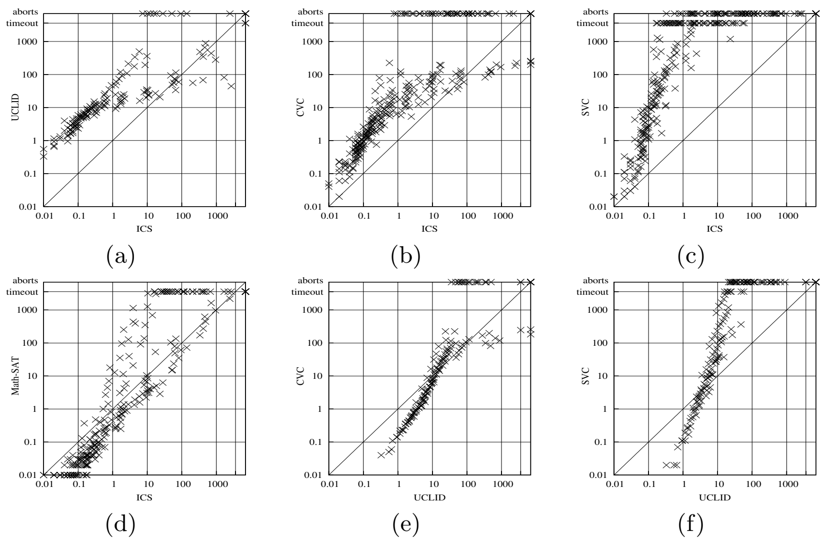


Fig. 2. Runtimes (in seconds) on Math-SAT benchmarks.

The SEP benchmark suite (http://iew3.technion.ac.il/~ofers/smtlib-local/index.html) is derived from symbolic simulation of hardware designs, timed automata systems, and scheduling problems. The problems are represented in non-clausal form, and constraints in difference logic. This suite includes only 17 problems, the size of the ASCII representation of these problems ranges from 1.5Kb to 450Kb.

We developed various translators from the input format used in each benchmark suite to the input format accepted by each GDP described on the previous section, but Math-SAT. We did not implement a translator to the Math-SAT format because it only accepts formulas in the CNF format, and a translation to CNF would destroy the structural information contained in the original formulas. In addition, no specifics are provided for generating hints for Math-SAT. In developing these translators, we have been careful to preserve the structural information, and we used all the information available to use available language

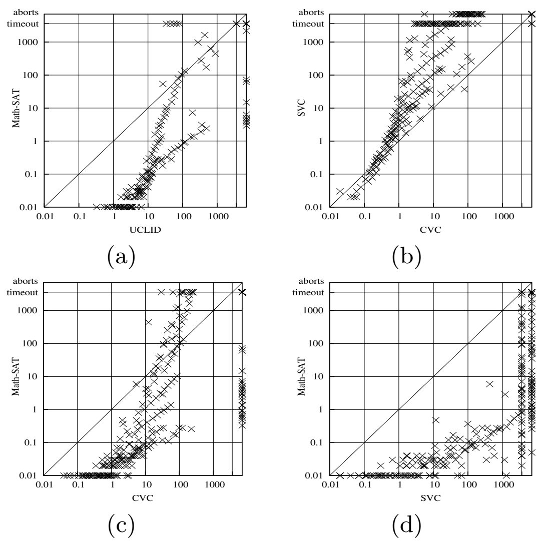


Fig. 3. Runtimes (in seconds) on the Math-SAT benchmarks (cont.)

features useful for the underlying search mechanisms.[2] The Math-SAT *search hints*, however, are ignored in the translation, since this extra-logical information is meaningless for all the other tools.

3.3 Setup

All experiments were performed on machines with 1GHz Pentium III processor 256Kb of cache and 512Mb of RAM, running Red Hat Linux 7.3. Although these machines are now outdated, we were able to set aside 4 machines with identical configuration for our experiments, thereby avoiding error-prone adjustment of performance numbers. We considered an instance of a GDP to *timeout* if it took more than 3600 secs. Each GDP was constrained to 450Mb of RAM for the data area, and 40Mb of RAM for the stack area. We say a GDP *aborts* when it runs out of memory or crashes for other reasons. With these timing and memory constraints running all benchmarks suites requires more than 30 CPU days.

For the diversity of the input languages and the algorithms underlying the GDPs, we do not include any machine-independent and implementation-independent measures. Instead, we restrict ourselves to reporting the user time of each process as reported by Unix. In this way, we are measuring only the systems as implemented, and observations from these experiments about the underlying algorithms can only be made rather indirectly.

[2] We have also been contemplating a malicious approach for producing worst-possible translations, but decided against it in such an early stage of benchmarking.

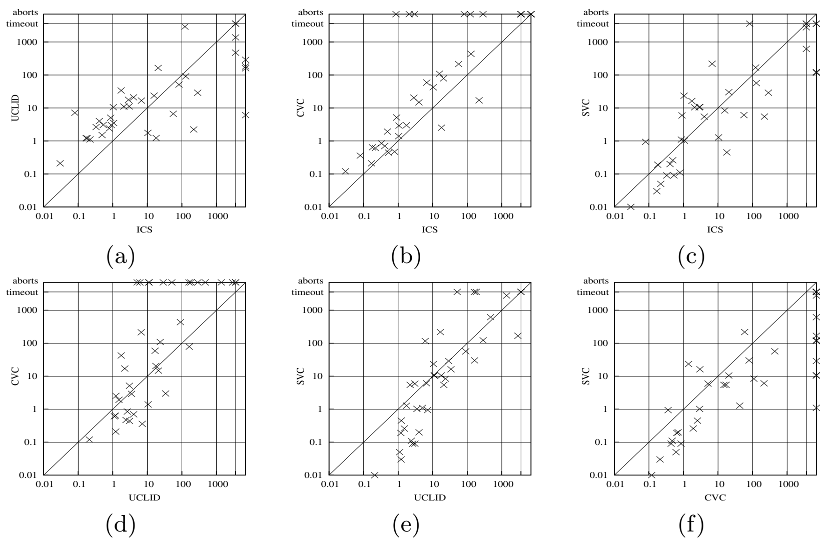


Fig. 4. Runtimes (in seconds) on the UCLID benchmarks.

With the notable exception of CVC Lite, all GDPs under consideration were obtained in binary format from their respective web sites. The CVC Lite binary was obtained using g++ 3.2.1 and configured in optimized mode. Table 1 contains the command line options used to execute each GDP. [3]

4 Experimental Results

Table 2 shows the number of satisfiable and unsatisfiable problems[4] for each benchmark, and a problem has been classified "Unsolved" when none of the GDPs could solve it within the time and memory requirements. The scatter graphs in

– Figures 2 and 3 include the results of running CVC, ICS, UCLID, MathSAT, and SVC on the Math-SAT benchmarks,
– Figure 4 contains the runtimes of CVC, ICS, SVC, UCLID on the UCLID benchmarks, and
– Figure 5) reports on our results of running CVC, ICS, SVC, and UCLID on the SAL benchmarks

[3] In [20] the depth first search heuristic (option `-dfs`) is reported to produce the best overall results for CVC, but we did not use it because this flag causes CVC to produce many incorrect results.

[4] For validity checkers, satisfiable and unsatisfiable should be read as invalid and valid instances respectively.

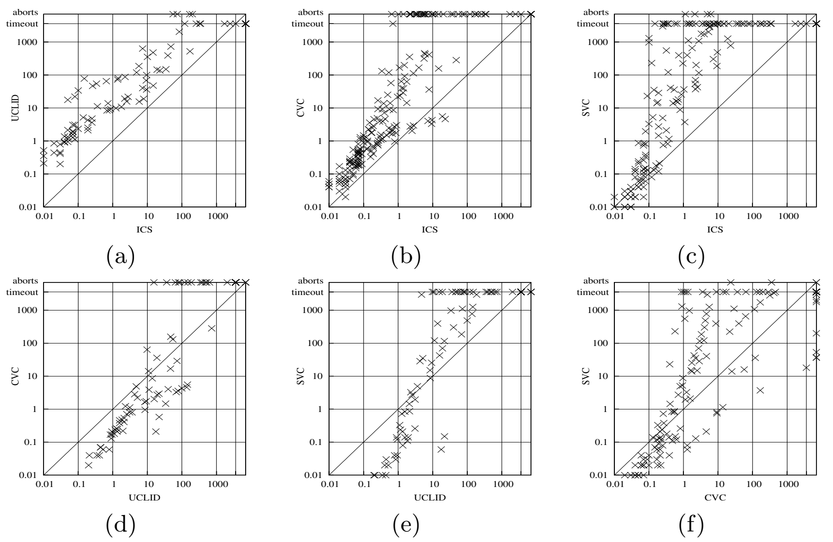


Fig. 5. Runtimes (in seconds) on the SAL benchmarks.

using the experimental setup as described in Section 3. Points above (below) the diagonal correspond to examples where the system on the x (y) axis is faster than the other; points one division above are an order of magnitude faster; a scatter that is shallower (steeper) than the diagonal indicates the performance of the system on the x (y) axis deteriorates relative to the other as problem size increases; points on the right (top) edge indicate the x (y) axis system timed out or aborted. Multiplicities of dots for timeouts and aborts are resolved in Table 3.

For lack of space, the plots for Simplify and CVC Lite are not included here. Simplify performed poorly in all benchmarks except SEP, and does not seem to be competitive with newer GDP implementations. In the case of CVC Lite, its predecessor system CVC demonstrated over-all superior behavior. Also, the SEP suite did not prove to distinguish well between various systems, as all but one problem could be solved by all systems in a fraction of a second. All these omitted plots are available at `http://www.csl.sri.com/users/demoura/gdb-benchmarks.html`.

Figures 2 and 3 compare ICS, UCLID, CVC, SVC and Math-SAT on the Math-SAT suite. The plots comparing UCLID contain only the problems in the difference logic fragment supported by UCLID. The results show that the overall performance of ICS is better than those of UCLID, CVC and SVC on most problems of this suite.

With the exception of Math-SAT (which was applied only to its own benchmark set), every other system failed on at least several problems—mainly due to exhaustion of the memory limit. Also, the Math-SAT problems proved to be a non-trivial test on parsers as SVC's parser crashed on several bigger problems (see Table 1). The performance of SVC is affected by the size of the problems and the CNF format used (Figure 3(d)) on this suite, since its search heuristics are heavily dependent on the propositional structure of the formula. On the other hand, the performance of the non-clausal SAT solver of ICS is not quite as heavily impacted by this special format. In fact, ICS is the only GDP which solves all problems solved by Math-SAT (Figure 2(d)), and it also solved several problems not solved by Math-SAT, even though *search hints* were used by Math-SAT. UCLID performs better than CVC and SVC on most of the bigger problems (Figures 2(e) and 2(f)).

Figure 4 compares ICS, UCLID, CVC, and SVC on the UCLID benchmarks. What is surprising here is that SVC is competitive with UCLID on UCLID's own benchmarks (Figure 4(e)). Also, the overall performance of SVC is superior to its predecessor CVC system (Figure 4(f)). ICS does not perform particularly well on this benchmark set in that it exhausts the given memory limit for many of the larger examples.

Figure 5 compares ICS, UCLID, CVC, and SVC on the SAL benchmarks. ICS performs better than UCLID on all examples (Figure 5(a)), and its overall performance is better than CVC (Figure 5(b)). ICS, UCLID and CVC fail on several problems due lack of memory. Although the overall performance of CVC seems better than UCLID, the latter managed to solve several problems where CVC failed (Figure 5(d)).

5 Observations and Research Questions

As has been mentioned in Section 3, the results produced in our experiments measure mainly implementations of GDPs. Nevertheless, we try to formulate some more general observations about algorithmic strengths and weaknesses from these experiments, which should be helpful in studying and improving GDPs.

Insufficient constraint propagation in lazy integrations. The eager UCLID system usually outperforms lazy systems such as ICS, CVC, and Math-SAT on problems which require extensive constraint propagation. This seems to be due to the fact that the underlying lazy integration algorithms only propagate inconsistencies detected by the constraint solver, but they do not propagate constraints implied by the current logical context. Suppose a hypothetical problem which contains the atoms $\{x = y, y = z, x = z\}$, and during the search the atoms $x = y$ and $y = z$ are assigned to true, then the atom $x = z$ could be assigned to true by constraint propagation (transitivity), but none of the existing lazy provers propagates this inference. In contrast, these kinds of propagations are performed in eager integrations, since the unit-clause rule of Davis-Putnam like SAT solvers assumes the job of propagating constraints.

Arithmetical constraints in the eager approach. On arithmetic-intensive problems, UCLID usually performs worse than other GDPs with dedicated arithmetic constraint solvers. In particular, UCLID's performance is heavily affected by the size of constants. One of the approaches used by UCLID to reduce difference logic to propositional logic [19] constructs a graph where each node represents an integer variable, and a weighted edge represents a difference constraint. Starting from this graph, a clause is created for each negative cycle. We believe several irrelevant clauses are generated in this process. We say a clause is irrelevant when it cannot be falsified during the search due to, for instance, the presence of other constraints. In several examples, UCLID consumed all memory in the translation to propositional logic. For instance, the problem `abz5-900` was easily solved by ICS and Simplify using less than 40Mb, but UCLID consumed all memory during the translation to propositional logic.

Performance vs. expressiveness. The version of UCLID we have been using is restricted to separation logic, but other systems with support for full arithmetic seem to be competitive even when problems are restricted to this domain. The goal should be a fully general system that reliably does the special cases (restricted theories, bounded instances) at least as well as specialized systems.

Blind search problem in lazy solvers. The problems in the UCLID suite are mainly composed of non-boolean variables (Figure 1), so almost all atoms are constraints, and the lazy constraint solvers introduce a fresh boolean variable for each distinct constraint. This process produces an underconstrained formula which contains several propositional variables which occurs only once. Thus, the begin of the search is completely chaotic, and arbitrary, since from the point of view of the SAT solving engine any assignment will satisfy the apparently easy and underconstrained formula. We say the SAT engine starts an almost *blind search*, where the successful search heuristics developed by the SAT community are hardly useful. Our hypothesis is corroborated by the Math-SAT and SAL suites, where several boolean variables are used to encode the control flow and finite domains. In this case, although the SAT engine does not know the hidden relationships between the freshly added boolean variables, it is guided by the control flow exposed by the boolean variables. This hypothesis also helps explaining why SVC performs better than ICS and CVC on the UCLID suite, simply because SVC uses specialized heuristics based on the structure of the formula.

Memory Usage. Math-SAT and Simplify are the systems using the least amount of memory. In contrast, CVC often aborts by running out of memory instead of running up its time limits. A similar phenomenon can be observed with ICS. We have traced this deficiency back to imprecise generation of explanations for pruning Boolean assignments. For instance, *path compression* is an optimization commonly used on congruence closure algorithms, however it produces less precise *explanations*. Better tradeoffs between the accuracy of the generated explanations and the cost for computing them are needed. Another problem in systems such as ICS and CVC is the maintenance of information to perform backtracking. Math-SAT is a non-backtrackable system, so it does not suffer from this problem, at the cost of having to restart from scratch every time an inconsistency is detected.

Loss of Structural Information. When developing the translators for these experiments, we noticed that the performance of most solvers heavily depends on the way problems are encoded. For instance, the repetitive application of the transformation $F[t] \implies F[x] \wedge x = t$ with t a term occurring in F, and x a fresh variable, transforms many easy problems into very hard problems for UCLID, CVC and CVC Lite.

6 Conclusions

Our performance study demonstrates that recently developed translation-based GDPs—eager or lazy—well advance the state-of-the-art. However, this study also exposes some specific weaknesses for each of the GDP tools under consideration. Handling of arithmetic needs to be improved for eager systems, whereas lazy systems can be considerably improved by a tighter integration of constraint propagation, specialized search heuristics, and the generation of more precise explanations. A main impediment for future improvements in the field of GDPs, however, is not necessarily a lack of new algorithms in the field, but rather the limited availability of meaningful benchmarks. Ideally, these benchmarks are distributed in a form close to the problem description (e.g. not necessarily in clausal normal form). A collaborative effort across the field of GDPs is needed here.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions in Computers **8** (1986) 677–691
2. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
3. Stump, A., Barrett, C.W., Dill, D.L.: CVC: a cooperating validity checker. In: Proc. of CAV'02. Volume 2404 of LNCS. (2002)
4. Filliâtre, J.C., Owre, S., Rueß, H., Shankar, N.: ICS: Integrated Canonization and Solving. In: Proc. of CAV'01. Volume 2102 of LNCS. (2001)
5. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Proc. of CAV'02. Volume 2404 of LNCS. (2002)
6. Audemard, G., Bertoli, P., Cimatti, A., Kornilowicz, A., Sebastiani, R.: A SAT based approach for solving formulas over boolean and linear mathematical propositions. In: Proc. of CADE'02. (2002)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
8. Barrett, C., Dill, D., Levitt, J.: Validity checking for combinations of theories with equality. LNCS **1166** (1996) 187–201
9. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpressions problem. Journal of the ACM **27** (1980) 758–771

10. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
11. Shostak, R.E.: Deciding combinations of theories. Journal of the ACM **31** (1984) 1–12
12. Shankar, N., Rueß, H.: Combining Shostak theories. In: Proc. of RTA'02. Volume 2378 of LNCS. (2002)
13. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Proc. of CADE'02. Volume 2392 of LNCS. (2002)
14. Ackermann, W.: Solvable cases of the decision problem. Studies in Logic and the Foundation of Mathematics (1954)
15. Shostak, R.E.: Deciding linear inequalities by computing loop residues. Journal of the ACM **28** (1981) 769–779
16. Goel, A., Sajid, K., Zhou, H., Aziz, A.: BDD based procedures for a theory of equality with uninterpreted functions. LNCS **1427** (1998) 244–255
17. Pnueli, A., Rodeh, Y., Shtrichman, O., Siegel, M.: Deciding equality formulas by small domains instantiations. LNCS **1633** (1999) 455–469
18. Bryant, R.E., German, S., Velev, M.N.: Exploiting positive equality in a logic of equality with uninterpreted functions. LNCS **1633** (1999) 470–482
19. Strichman, O., Seshia, S.A., Bryant, R.E.: Reducing linear inequalities to propositional formulas. In: Proc. of CAV'02. Volume 2404 of LNCS. (2002)
20. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Proc. of CAV'02. Volume 2404 of LNCS. (2002)
21. Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Proc. of CAV'03. Volume 2725 of LNCS. (2003)