SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers

Germain Faure, Robert Nieuwenhuis, Albert Oliveras,
and Enric Rodríguez-Carbonell*

Abstract. Many highly sophisticated tools exist for solving linear arithmetic optimization and feasibility problems. Here we analyze why it is difficult to use these tools inside systems for SAT Modulo Theories (SMT) for linear arithmetic: one needs support for disequalities, strict inequalities and, more importantly, for dealing with incorrect results due to the internal use of imprecise floating-point arithmetic. We explain how these problems can be overcome by means of result checking and error recovery policies.

Second, by means of carefully designed experiments with, among other tools, the newest version of ILOG CPLEX and our own new Barcelogic T-solver for arithmetic, we show that, interestingly, the cost of result checking is only a small fraction of the total T-solver time.

Third, we report on extensive experiments running exactly the same SMT search using CPLEX and Barcelogic as T-solvers, where CPLEX tends to be slower than Barcelogic. We analyze these at first sight surprising results, explaining why tools such as CPLEX are not very adequate (nor designed) for this kind of relatively small incremental problems.

Finally, we show how our result checking techniques can still be very useful in combination with inexact floating-point-based T-solvers designed for incremental SMT problems.

1 Introduction

The applicability of current SAT solvers to many areas in and outside computer science is nowadays well known. However, some practical problems are more naturally described and more efficiently solved in logics that are more expressive than propositional logic. For example, for reasoning about timed automata or about intervals in scheduling problems, a good choice is *difference logic*, where formulas contain atoms of the form $a - b \leq k$. Similarly, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the theory T of these data structures. In such applications, problems may consist of thousands of clauses like

$$p \quad \vee \quad \neg q \quad \vee \quad a = b - c \quad \vee \quad read(s, b - c) = d \quad \vee \quad a - c \leq 7$$

containing purely propositional atoms as well as atoms over (combined) theories. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory

* Tech. Univ. of Catalonia, Barcelona. All authors partially supported by Spanish Min. of Educ. and Science through the LogicTools-2 project, TIN2007-68093-C02-01.

H. Kleine Büning and X. Zhao (Eds.): SAT 2008, LNCS 4996, pp. 77–90, 2008.

T: given a formula F, determine whether F is T-satisfiable, i.e., whether there exists a model of T that is also a model of F. SMT has become an extremely active area of research and many SMT systems have been developed [DdM06a, dMB07, BBC+05, BT07, NO05a], as well as a library of benchmarks for SMT, called SMT-LIB [TR05].

The DPLL(T) approach to SMT couples a general DPLL(X) engine, in charge of enumerating propositional models of the formula, with a theory solver $Solver_T$, responsible for checking the consistency of these models over the theory T (e.g., if T is difference logic and the current boolean assignment contains $x - y \leq 0$, $y - z \leq 0$, and $x - z \geq 1$, then $Solver_T$ has to detect its T-inconsistency).

Here we consider SAT modulo the theories of Linear (Real or Integer) Arithmetic (LRA or LIA). So far, in SMT systems not much of the wide body of technology developed in the field of OR has been exploited. The reason for this is that the main application area of SMT is verification, which has some requirements that are not considered essential in OR: one needs to handle disequalities and strict inequalities, and, in order to guarantee correctness, employ arbitrary-precision arithmetic instead of floating-point arithmetic. The only work the authors are aware of the application of OR tools to SMT is [YM06], where nevertheless the issue of how incorrect answers from the solver should be handled was not addressed. Still, OR solvers *may* give wrong answers: for instance CPLEX 11 [ILO07], the newest version of ILOG CPLEX, returns that the following set of constraints (obtained from the industrial benchmark `clocksynchro_2clocks.main_invar.induct` from the SMT-LIB) is satisfiable:

$$\begin{aligned} -x - y + u &\leq 0 & \qquad -11z + v + 11t &\leq 0 \\ -u + z &\leq 0 & 11x - v &\leq -10^{-5} \\ -t + y &\leq 0 & x &\geq 10^{-5} \end{aligned}$$

However, it is unsatisfiable, as the reader can easily check by multiplying the first three constraints by 11 and adding up all constraints but the last bound (which is not in the conflict but is needed to get a wrong answer from CPLEX).

In this paper we further study the applicability of OR tools for developing theory solvers for LA. We show how imprecise floating-point-based simplex solvers can be used in combination with result checking and error recovery policies for handling solver failures.

Furthermore, we report on a large number of carefully designed experiments with commercial and non-commercial OR solvers, including CPLEX 11, and with several versions of our own new Barcelogic $Solver_T$ for LRA and LIA. These experiments show, among several other interesting results, that (i) result checking takes only a small fraction of the total OR solver time and (ii) OR solvers are not designed for the incremental feasibility problems that occur in SMT and are often outperformed in this context by our specialized exact T-solver.

This closes some research directions and opens other new ones. In particular, it seems that a good approach may be to combine result checking with floating-point implementations of our current SMT-style incremental solvers. Following

this idea we have implemented a prototype using floating-point arithmetic, which we have compared experimentally with CPLEX obtaining promising results.

This paper is structured as follows. We first give some background on SMT and DPLL(T) in Section 2. Section 3 studies which functionalities are offered and missing in OR solvers in order to be used as theory solvers. Then, Section 4 concentrates on how to use inexact solvers like CPLEX in DPLL(T). Next, Section 5 analyzes the performance of OR solvers when used as theory solvers. Finally, Section 6 presents preliminary results on the development of inexact solvers specifically designed for SMT, and we conclude in Section 7.

2 Background on SMT and DPLL(T)

In this section we give a quick overview of SMT and DPLL(T). We refer to [NOT06] for further details, extensions and references. The SMT problem consists of, given a ground first-order formula F and a theory T, deciding whether F is T-*satisfiable* (or T-*consistent*), i.e., whether there exists a model of T that is also a model of F. For that purpose, most state-of-the-art SMT solvers combine a boolean engine DPLL(X), very similar in nature to a SAT solver, with a theory solver $Solver_T$, thus producing a DPLL(T) system.

In the simplest version of such systems, the boolean engine initially considers each atom as a distinct propositional symbol. If the formula turns out to be propositionally unsatisfiable, it is T-unsatisfiable as well. Otherwise, DPLL(X) returns a propositional model M. This model, seen as a conjunction of literals, is then checked for T-consistency by $Solver_T$. If M is T-consistent then F is T-satisfiable; otherwise, in order to prevent M from later consideration, one can conjunct the negation of M (a disjunction of literals) to F and repeat the process until DPLL(X) finds a T-consistent model or returns unsatisfiable.

Example 1. Let F be $x \leq 2 \ \wedge \ (\neg(x + y = 1) \vee x \geq 3) \ \wedge \ x + y = 1$. In this case, DPLL($X$) will return the model $M = \{x \leq 2,\ x \geq 3,\ x + y = 1\}$ which will be detected T-inconsistent by $Solver_T$. After adding to F the clause $\neg(x \leq 2) \vee \neg(x \geq 3) \vee \neg(x + y = 1)$, the boolean engine will report the unsatisfiability of the formula.

In this simple setting all one needs from $Solver_T$ is the capability of checking the T-consistency of a conjunction of literals. However, for this approach to be efficient in practice several improvements need to be made. Here we list some of them, making special emphasis on the requirements they pose on $Solver_T$:

- The T-consistency of the assignment stored by DPLL(X) can be checked while it is being built, without delaying the check until a propositional model has been found (i.e., we are at a *leaf* of the search tree). This saves a large amount of useless work but requires $Solver_T$ to be incremental, that is, being faster in processing the addition of a single literal to a set of literals already found T-consistent than in reprocessing the whole set from scratch.

- When an assignment M is found T-inconsistent by $Solver_T$, one can ask DPLL(X) to backtrack to some point where the assignment was still T-consistent instead of restarting the search from scratch. This obviously forces $Solver_T$ to be able to support backtracking. Moreover, DPLL(X) needs to start its conflict analysis mechanism with an *inconsistency explanation* given by $Solver_T$, that is, a small subset of M that is also T-inconsistent (e.g, in Example 1, an inconsistency explanation is $\{x \leq 2,\ x \geq 3\}$).
- As a further optional refinement, if we want $Solver_T$ to play an active role in the search, instead of being used only to validate the search a posteriori, we can ask $Solver_T$ to detect unassigned input literals that are T-entailed by the current assignment M; that is, literals l such that $M \wedge T \models l$. This refinement, called *theory propagation*, allows DPLL(X) to assign them a truth value instead of having to guess an arbitrary value for them.

These improvements have allowed SMT solvers to be successfully used in a variety of applications. Many of them involve reasoning over the theory of *linear arithmetic* (LA), where atoms are of the form $a_1x_1 + \ldots + a_nx_n \bowtie b$, being the a_i's rational numbers, the x_i's integer or rational variables and $\bowtie$ one of the operators $=$, $\leq$, $<$, $>$, $\geq$ or $\neq$. An interesting fragment of linear arithmetic is the one of difference logic (DL), where atoms are of the form $x_1 - x_2 \bowtie b$. In SMT benchmarks most LA constraints are indeed DL and their consistency can be checked very efficiently by means of negative-cycle-detection algorithms. Hence, when checking the T-consistency of a set of LA constraints it is not uncommon to first apply a specialized DL solver to filter out the inconsistencies that arise only taking into account DL atoms. On the other hand, for dealing with general LA constraints all state-of-the-art theory solvers in SMT tools are based on the *simplex method*. For further reading see, e.g., [Sch87].

3 Using OR Solvers as Theory Solvers for LA

In this section we summarize what OR solvers provide and miss so as to be applied to DPLL(LA).

All linear programming (LP) packages developed in OR allow the user to test the satisfiability of a conjunction of linear equations and non-strict inequations. Very often there is no specific facility for this purpose, since all that needs to be done is to optimize the null function over the system of constraints of interest: all models of the formula are optimal with respect to this objective function.

Moreover, most of these systems implement the so-called *bounded* simplex method [Mar86], which handles bounds on variables in a more efficient way than in the textbook version [Sch87]. This is important in the SMT context, since typically a significant amount of the literals in a problem are bounds: on average over 30% in the SMT-LIB, and in some benchmarks beyond 50%.

Also important as regards efficiency, the majority of these packages provide an API that avoids expensive communication through files and system calls.

Another issue that is paramount for the application to DPLL(LA) is incrementality: fortunately, most often the interfaces of these tools provide facilities for adding and removing constraints and modifying bounds, among others.

However, when one is faced with an unsatisfiable conjunction of constraints, as far as the authors know only commercial LP tools (or demo versions with limited capabilities of these) provide a means for computing an irredundant explanation for the inconsistency. Moreover some of these, such as CPLEX 9.1, produce explanations for LRA but not for LIA; besides, for some pathological instances, the explanations given by CPLEX 9.1 are redundant, though they should not be according to the documentation. For example, for the following system of constraints:

$$x + y \leq 2 \ \wedge \ x \leq 1 \ \wedge \ x \geq 1 \ \wedge \ y \leq 2 \ \wedge \ y \geq 2$$

CPLEX 9.1 considers the conjunctions $x \leq 1 \wedge x \geq 1$ and $y \leq 2 \wedge y \geq 2$ as the equations $x = 1$ and $y = 2$ respectively, and returns $E = \{x + y \leq 2, x = 1, y = 2\}$ as an irredundant explanation, whereas $E' = \{x + y \leq 2, x \geq 1, y \geq 2\}$ is a proper subset of E that is also inconsistent. Fortunately CPLEX 11 fixes these problems and does produce truly irredundant explanations for both LRA and LIA.

On the other hand, to the knowledge of the authors what all LP packages lack is support for handling disequalities and strict inequalities. Basically this is due to two facts: (1) optimization problems with these constraints may not have optimal solutions, and (2) in LP data are not usually absolutely precise, e.g., because they are subject to measurement errors.

Another feature that most LP tools lack is precise arithmetic. For the sake of efficiency, typically an OR solver works with floating-point arithmetic, instead of arbitrary-precision rationals as done in SMT solvers. This is the reason why, as shown in Section 1, an OR solver may give a wrong answer, i.e., return "SAT" for an unsatisfiable problem or "UNSAT" for a satisfiable one, or also compute a wrong explanation of inconsistency. Moreover, the use of floating-point arithmetic entangles the risk of a sudden unexpected failure; this is one of the reasons why optimization routines in LP libraries return a status value that indicates whether an internal error has occurred.

For instance, all of the versions of CPLEX we have experimented with just support floating-point arithmetic. On the other hand, the non-commercial OR solver GLPK [Mak07] additionally provides the user with exact arbitrary-precision arithmetic. See Section 5 for the results of our experiments with this feature.

Finally, no LP package supports theory propagation. This is natural, since in the context of OR this notion does not make any sense. Although the importance of theory propagation has been acknowledged elsewhere for LA and other theories [NO05b, DdM06b], one of the initial hypotheses of this research was that, given the huge amount of work done in the area of OR over the years, the performance of LP tools would be so outstanding that this limitation would be compensated for. Further, as seen in Section 2, theory propagation is not a

necessary part of the core interface with DPLL(X), but an optimization on this interface.

4 How to Deal with Inexact Solvers in DPLL(T)

As discussed in Section 3, there are two issues that must be addressed so as to employ an inexact OR solver as a LA-solver:

(1) In the SMT context, constraints may be not only equalities and non-strict inequalities but also the negation of these, i.e., disequalities and strict inequalities; the OR solver must be able to handle them all.
(2) Due to imprecise arithmetic, the answers given by the OR solver may be wrong, and thus must be checked; moreover, there must be a policy for recovering from the possible errors and resuming the search.

In this section it is shown how this gap can be filled. As far as (1) is concerned, the problem of handling disequalities can be reduced to that of strict inequalities, since one can preprocess the input formula by splitting equalities into conjunctions of non-strict inequalities and disequalities into disjunctions of strict inequalities, which works very well in practice [DdM06b]. Now, given that the issue of correctness of the OR solver needs to be addressed anyway, a possibility is to strengthen strict inequalities by subtracting a small value ϵ; i.e., a constraint of the form $c^T x < d$ is transformed into $c^T x \leq d - \epsilon$ (for instance, in our experiments we used $\epsilon = 10^{-5}$). Thus, the problem (1) of handling strict constraints has been reduced to (2), that of correctness of the inexact solver.

Now, regarding (2), a general solution for using inexact T-solvers in DPLL(T) (not necessarily OR solvers when T is LA) is to check results by means of an exact T-solver only when it is strictly necessary to ensure correctness. That is, (i) whenever the inexact solver returns "UNSAT", checking that the explanation for the conflict is indeed inconsistent; and (ii) whenever the inexact solver returns "SAT" (or an internal error occurs) *at a leaf*, checking that the assignment is indeed consistent with the theory. A corresponding error recovery policy can be easily described: in case (i), if the explanation is wrong, the exact solver is called again over the partial assignment, and the search is resumed using the result of this exact consistency check; similarly, in case (ii) the result of the check with the exact solver is employed to continue the search. Both result checking and error recovery policies are summarized in Algorithm 1.

Notice that the most expensive calls to the exact solver are those where the consistency of the whole partial assignment is checked. Under the hypotheses that the inexact solver will produce almost no wrong explanations of inconsistency and that internal errors will be infrequent too, these calls will be basically due to the uncommon event of the boolean search getting to a leaf of the tree. So it is reasonable to imagine that the cost of these calls will not be noticeable.

As regards the calls to the exact solver with inconsistency explanations, these will be much more frequent, since typically every few decisions a conflict arises.

Algorithm 1. Consistency check and error recovery policies

```
if in a leaf then
    if there is an internal error or inexact solver returns "SAT" then
        check consistency of partial assignment with exact solver;
        resume search using the result given by exact solver;
    else //inexact solver returns ''UNSAT''
        check consistency of inconsistency explanation with exact solver;
        if exact solver returns "SAT" then
            check consistency of partial assignment with exact solver;
            resume search using the result given by exact solver;
        else //exact solver returns ''UNSAT''
            resume search using inconsistency explanation for conflict analysis;
else //in an internal node
    if there is an internal error or inexact solver returns "SAT" then
        continue search as if partial assignment were theory consistent;
    else //inexact solver returns ''UNSAT''
        check consistency of inconsistency explanation with exact solver;
        if exact solver returns "SAT" then
            continue search as if partial assignment were theory consistent;
        else //exact solver returns ''UNSAT''
            resume search using inconsistency explanation for conflict analysis;
```

Fortunately, in general the number of literals in an explanation is below a few tens, and therefore these calls are often cheap.

In order to empirically assess the cost of result checking, we have carried out the following experiment: for all benchmarks in the LRA and LIA divisions (501 and 203 problems, respectively) of the SMT-LIB [TR05], we have run our SMT tool using CPLEX 11 as a LA-solver and implementing the result checking and error recovery policies presented in Algorithm 1 with our exact Barcelogic LA-solver. In order to avoid noise, no difference-logic pre-filtering has been applied. In this and in the rest of experiments in this paper, the machine used was a PC with an Intel(R) Xeon(TM) CPU 3.80GHz processor running Linux Debian 4.1.1. The timeout was set to 15 minutes.

In the graph in Figure 1 each dot represents an SMT instance of LRA. The horizontal axis shows the time spent in CPLEX (consistency checking, inconsistency explanation generation and backtracking); the vertical axis represents the time taken by result checking. Besides, the line $y = x/10$ is drawn as a reference.

As can be seen from the graph, for most problems in LRA the time taken by result checking is in general at most 10% of the time spent in CPLEX. In those instances for which result checking is significantly more expensive than that, this is due to either (1) the length of the inconsistency explanations (for some examples in the `TM` family, several hundreds of literals) or (2) the amount of errors produced by CPLEX. However, in more than 75% of the benchmarks, errors occur in at most 2% of the consistency checks.

The graph in Figure 2 is similar to that in Figure 1, but for benchmarks from LIA. In this case it is also clear that, in general, the cost of result checking is at most 10% of the time spent by CPLEX.

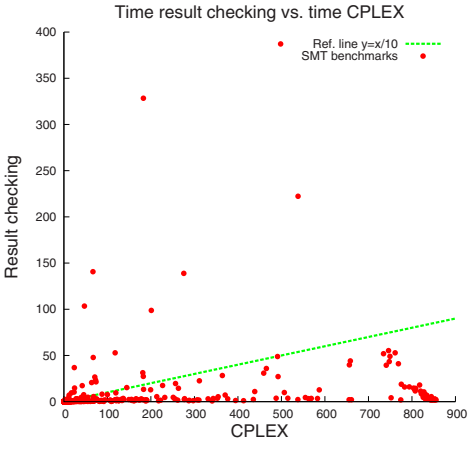

Fig. 1. Result checking evaluation for LRA

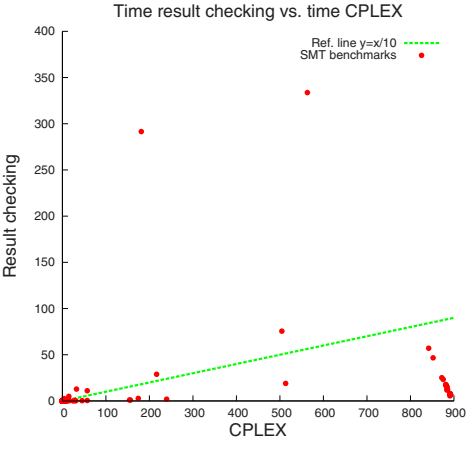

Fig. 2. Result checking evaluation for LIA

5 Performance of OR Solvers as *T*-Solvers

In this section we experimentally evaluate the performance of inexact OR solvers against that of specialized exact LA-solvers designed for SMT.

To this end, we have carried out the following experiment: guiding the search with our exact Barcelogic LA-solver, we have also run in parallel CPLEX 11 and compared the timings of the two tools, counting consistency checks [1], incon-

[1] There is a difference in the way consistency checks were performed with each tool. For our LA-solver, pending constraints were asserted one at a time; for CPLEX, all pending constraints were asserted at the same time. The reason for this is that, if CPLEX was asked to deal with constraints one at a time, it was much slower.

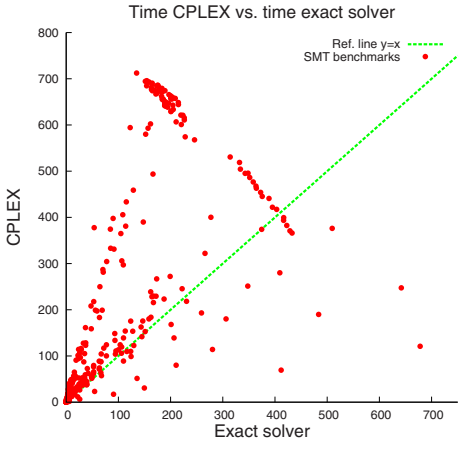

Fig. 3. Comparison between our exact solver and CPLEX in LRA

sistency explanation generation and backtracking. To make a fair comparison, apart from exploring the search space in the same way, neither difference-logic pre-filtering nor theory propagation have been applied.

The graph in Figure 3 shows the results of this experiment for LRA. Each dot represents an SMT instance. The horizontal axis is the time taken by our own exact LA-solver; the vertical axis is the time spent by CPLEX. Besides, the line $y = x$ is drawn as a reference.

Contrary to our initially expected results, when used as a theory solver in the DPLL(T) framework, CPLEX 11 tends to perform worse or not significantly better than our LRA solver. This is mainly due to consistency checks, and also inconsistency explanation generation, which are usually more expensive with CPLEX 11. The same experiments have also been carried out with other inexact OR solvers, namely CPLEX 9.1 and GLPK 4.25, the newest version of the GNU Linear Programming Kit, with similar outcome (although CPLEX 11 performs better than CPLEX 9.1, which is in turn better than GLPK). We have worked on several hypotheses in order to explain these results:

The default CPLEX parameter values are not adequate for SMT. The experiments above have been carried out using the default values for the parameters of CPLEX, so one could argue that these values are not the most appropriate for SMT problems. For this reason we have experimented changing several of the parameters that, according to CPLEX documentation, have most impact on the performance: simplex method (primal, dual, barrier), pricing strategy (standard, steepest edge, devex, ...), refactorization frequency, and several preprocessing options. No significant improvements have been achieved on the results obtained with the default values of the parameters.

The basis is refactored at each constraint addition/deletion. CPLEX allows writing a log file with information about the progress of the computation. From these log files it can be seen that refactorizations are not performed

systematically each time constraints are added or removed, but more spacedly. Also, as mentioned above, we did not significantly enhance the results by modifying the refactorization frequency.

CPLEX is using a Phase I procedure that adds many new auxiliary variables and/or rows to the problem at each consistency check. As far as the authors could infer from the documentation, the Phase I primal algorithm implemented in CPLEX is based on [Mar86], where no extra rows or variables are added to the problem. Moreover, if the dual simplex method is employed, since the objective function is null any basis is trivially feasible, and thus all work is done in Phase II, where no auxiliary rows or variables are added either; still, we did not improve timings by using the dual simplex method, as said above.

CPLEX is not designed for being used as a $Solver_T$ in DPLL(T), nor for the kind of problems that arise in SMT. This is the most plausible explanation for the results obtained in this experiment, since the way CPLEX is commonly used in OR is remarkably different from that in this paper for SMT.

First of all, CPLEX is aimed at linear programs with up to millions of variables and constraints, whereas consistency checks from SMT involve few thousands of constraints over few hundreds of variables. Thus, using CPLEX for solving these problems may be an overkill.

Secondly, when in OR a linear program is solved, typically the user carries out some sensitivity analysis; in order to reuse computations in further reoptimizations, CPLEX provides the facilities not only for adding/removing constraints and changing bounds, but also changing coefficients of the objective function and the whole constraint matrix. However, efficiency in adding/removing constraints and changing bounds is not as determinant as in DPLL(T), where thousands of problems need to be solved incrementally for a single benchmark. As a result of this, CPLEX does not outperform our exact LA-solver in an incremental setting, whereas when solving large static problems it is better by orders of magnitude.

As regards inconsistency explanations, the typical scenario in OR is the following one: when dealing with big linear programs it may be tedious to detect errors in the data, for instance when the problem turns out to be infeasible whereas it should not; CPLEX offers functionalities for computing conflicting sets of constraints in order to help the user to diagnose where the error could be. Thus, in the context for which CPLEX has been designed, the computation of explanations of inconsistency is not critical, unlike in DPLL(T). In fact, while we were experimenting with a previous version of CPLEX, CPLEX 9.1, the bottleneck for many problems in LRA (namely, the `sc` and `TM` families) was precisely the generation of these explanations. CPLEX 11 is more efficient than its predecessor when computing inconsistency explanations, but there are still instances for which it does not perform very well.

Finally, CPLEX provides the user with finely tuned technology for optimizing hard problems, among others sophisticate pricing strategies, several optimization algorithms, advanced basis methods, etc. On the other hand, from the optimization point of view, the linear programs arising from SMT problems are easy and

can be usually solved with few iterations of the simplex algorithm. Again, using CPLEX in this context may be excessive.

In order to look further into the cost of consistency checks in OR solvers, we experimented with the open-source OR solver GLPK 4.25 [2], which supports both floating-point and arbitrary-precision arithmetic. As regards inexact arithmetic, as mentioned above the results of the experiments were similar to those with CPLEX 11, although the performance of GLPK was worse than that of CPLEX. The execution profiles showed that about half of the time in GLPK was spent on factorizing the basis and the rest on (re)initializing data structures and simplex iterations, but did not reveal any deeper insights. Regarding exact arithmetic, GLPK performed two orders of magnitude worse than our exact LA-solver.

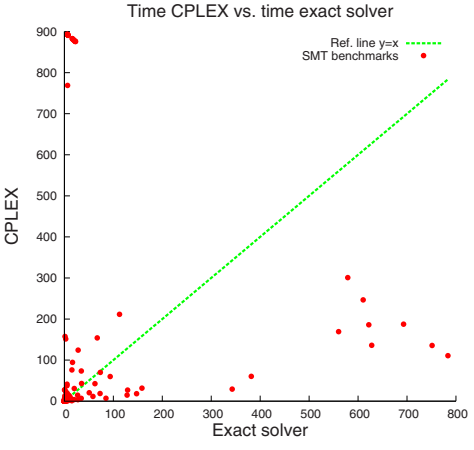

Fig. 4. Comparison between our exact solver and CPLEX in LIA

Finally, in Figure 4 we show the results of the experiment with CPLEX 11 and our exact solver on all LIA benchmarks from the SMT-LIB. As can be seen from the graph, CPLEX does perform better in general than our LIA solver (notice, however, that in some instances CPLEX is much slower; this is because for these particular LIA benchmarks it spends a huge amount of time computing explanations of inconsistency). This outcome is explained by the simplicity of our LIA solver, whose heuristics for branch & bound and cut generation have not been finely tuned. Nevertheless, given that the underlying engine for solving integer problems is a solver for reals, and given the above results for LRA, it seems reasonable to think that this difference between CPLEX and our LIA solver can be reduced if our search mechanism for integer solutions is improved.

[2] GLPK does not provide facilities for computing irredundant explanations of inconsistency, and so it was just used to check the consistency of partial assignments.

6 New Prospects: An Inexact Solver Designed for DPLL(T)

In this section we report on work in progress towards the use of inexact LA-solvers specifically designed for SMT as opposed to solvers developed in OR, based on the results obtained in the previous section.

Namely, in Section 5 our experiments in LRA have revealed that state-of-the-art OR solvers such as CPLEX 11 and GLPK 4.25, when applied in the DPLL(T) framework, are not competitive with specialized tools. Though in principle this is a negative result, in fact it suggests a new line for research: to combine result checking techniques with implementations of our current SMT incremental solvers using floating-point instead of arbitrary-precision arithmetic.

To assess the viability of this idea, we have run in parallel CPLEX 11 and an implementation of our LA-solver with floating-point numbers, using result checking and the error recovery policies described in Section 4. Our inexact LA-solver is currently a first-stage prototype that has been implemented basically by replacing exact rational variables by `double` variables, but without fine tuning for handling precision errors. As expected, using floating-point numbers in code designed for exact arithmetic may sometimes cause invariant violation and thus runtime errors and non-terminating behavior in the solver. For this reason, the experiment described here does not include all benchmarks from the LRA division of the SMT-LIB, but just those for which these errors did not occur. Interestingly enough, just 15% of the benchmarks were discarded; these are mainly the most difficult ones in the `clock_synchro`, `sc` and `tta_startup` families.

The outcome of this experiment is shown in the graph in Figure 5. Again, each dot represents an SMT benchmark. The horizontal axis is the time taken by our inexact LA-solver prototype; the vertical axis is the time spent by CPLEX. Besides, the lines $y = x$ and $y = 5x$ are drawn as a reference.

As can be seen from the graph, the results are promising. For most instances, CPLEX 11 spends at least five times as much time as our inexact solver. Taking into account the results obtained in the previous section, there is thus a potential gain in employing inexact LA-solvers implemented with floating-point arithmetic over exact LA-solvers implemented with arbitrary-precision numbers. [3]

However, two problems need to be addressed. First, although result checking was not an issue when using CPLEX, the situation is different with our prototype, for which the cost of ensuring correctness starts to become significant over the total time spent in theory reasoning. Therefore, result checking becomes eligible for optimization. A possibility in this direction is as follows. In all of the experiments reported here, result checking is implemented by having an auxiliary checking solver that each time it is called asserts all required constraints, and once the answer is returned it is emptied; this could be enhanced, for instance,

[3] A more precise experiment would have been to compare our inexact solver with the exact one. However this was not possible: our implementation employs static objects, which prevents us from having two solvers running simultaneously without resorting to system calls.

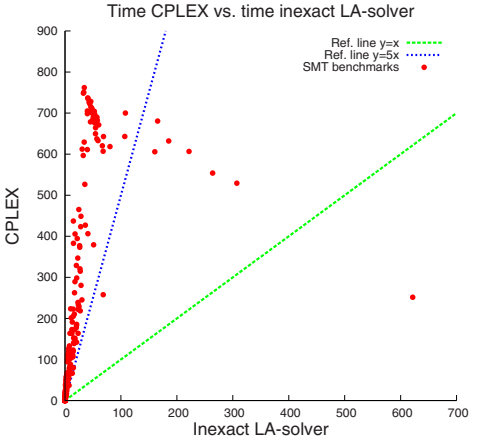

Fig. 5. Comparison between our inexact solver and CPLEX in LRA

by having two auxiliary solvers, one for checking explanations of inconsistency and another one for checking consistency of partial assignments, and working incrementally with the second one. Moreover, so far the inexact solver does not communicate any internal information to the auxiliary checking solver: still, the latter could use some data from the former to speed up the consistency check, for example which is the feasible basis or which are the multipliers of the inconsistency certificate. The second problem that has to be solved is that, even though result checking guarantees that on normal termination the answer given by the SMT tool will be correct, runtime errors or non-termination are clearly undesirable. It remains to be seen how these situations can be avoided without much computational effort.

On the other hand, unlike with OR solvers, this approach has the advantage that theory propagation could be applied by properly extending the result checking and error recovery policies. This would take the best of the two worlds: first, the efficiency of floating-point arithmetic; and second, the possibility to convey theory information to the boolean engine.

7 Conclusions

The main contributions of this paper can be summarized as follows. First, we have explained how OR tools can be used as theory solvers for SMT by means of result checking techniques and error recovery policies. Second, we have shown that the cost of the result checking techniques is only a small fraction of the time spent in the OR solver. Third, by means of exhaustive experiments we have shown that OR tools tend to be slower than exact solvers specifically designed for the DPLL(T) framework, and thus are not adequate in the context of SMT. Finally, based on empirical results we outline a new direction of research for obtaining efficient theory solvers, which consists in combining inexact

floating-point-based implementations of solvers designed for DPLL(T) with result checking and error recovery policies.

Acknowledgments. The authors would like to thank J. Cortadella, J. Carmona and J. Larrosa for technical support with CPLEX. We are also grateful to L. de Moura, P. Stuckey and the anonymous referees for insightful comments.

References

[BBC+05] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: The MathSAT 3 System. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 315–321. Springer, Heidelberg (2005)

[BT07] Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)

[DdM06a] Dutertre, B., de Moura, L.: The YICES SMT Solver. Technical report, SRI International (2006), Available at `http://yices.csl.sri.com`

[DdM06b] Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

[dMB07] de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. Technical report, Microsoft Research, Redmon (2007), Available at `http://research.microsoft.com/projects/z3`

[ILO07] ILOG. ILOG CPLEX v.11 (2007), `http://www.ilog.com/products/cplex`

[Mak07] Makhorin, A.: GLPK 4.25 (GNU Linear Programming Kit) (2007), Available at `http://www.gnu.org/software/glpk/`

[Mar86] Maros, I.: A general Phase-I method in linear programming. European Journal of Operational Research 23(1), 64–77 (1986)

[NO05a] Nieuwenhuis, R., Oliveras, A.: Decision Procedures for SAT, SAT Modulo Theories and Beyond. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 23–46. Springer, Heidelberg (2005)

[NO05b] Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)

[NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM, JACM 53(6), 937–977 (2006)

[Sch87] Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1987)

[TR05] Tinelli, C., Ranise, S.: SMT-LIB: The Satisfiability Modulo Theories Library (2005), `http://goedel.cs.uiowa.edu/smtlib/`

[YM06] Yu, Y., Malik, S.: Lemma Learning in SMT on Linear Constraints. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 142–155. Springer, Heidelberg (2006)