Rewrite and Decision Procedure Laboratory: Combining Rewriting, Satisfiability Checking, and Lemma Speculation

Alessandro Armando[1], Luca Compagna[1], and Silvio Ranise[2,⋆]

[1] DIST – Università degli Studi di Genova, Viale Causa 13 – 16145 Genova, Italia
[2] LORIA & INRIA – Université Henri Poincaré-Nancy 2,
615, rue du Jardin Botanique, BP 101, 54602 Villers les Nancy Cedex, France

1 Introduction

The lack of automated support is probably the main obstacle to the application of formal method techniques in the industrial setting. A possible solution to this problem is to combine the expressiveness of general purpose reasoners (such as theorem provers) with the efficiency of specialized ones (such as decision procedures). This is witnessed by the fact that many theorem provers developed for verification purposes (such as Acl2 [11], PVS [17], Simplify [9], STeP [14], and Tecton [10]) have integrated procedures for ubiquitous theories such as the theory of equality, decidable fragments of arithmetics, lists, and arrays. Unfortunately, designing an effective integration is far from being a trivial task and the solutions available in the verification systems listed above are not completely satisfactory for two main reasons. First, the schemes designed to incorporate decision procedure in larger systems are not flexible enough to allow developers to easily incorporate new procedures. Second, only a tiny portion of the proof obligations arising in many practical applications falls exactly into the domain the specialized reasoners are designed to solve. Thus, in many cases, available decision procedures are of little help if they are not combined with mechanisms for widening their scope.

1.1 RDL

RDL [1] is the acronym for **R**ewrite and **D**ecision procedure **L**aboratory. It provides an extension of rewriting to a powerful simplification mechanism exploiting satisfiability procedures for conjunctions of literals (i.e. reasoners specialized to solve the satisfiability problems for certain theories). The interplay between the satisfiability procedure and the other modules of **RDL** is parametric in the theory in which the procedure works. As a consequence, the reasoning activity implemented by the system can be easily extended by plugging-in new satisfiability

⋆ The last author is also partially supported by Università degli Studi di Genova under the program "Finanziamenti a progetti di singoli e/o giovani ricercatori (D.R. 226 del 25.10.2000)".

procedures. In turn, incorporated satisfiability procedures can be extended to handle larger classes of formulas by instantiating a generic lemma speculation mechanism. This mechanism is parametric in the satisfiability procedure being extended.

Our main motivation to develop **RDL** is to experiment with different combinations of rewriting and possibly extended satisfiability procedures. The focus of the system is on simplification of quantifier-free clauses and corresponding applications in the area of verification supported by automated theorem proving.

RDL features the following characteristics.

1. It is based on Constraint Contextual Rewriting (CCR) [2, 3]. CCR is a powerful simplification mechanism in which contextual rewriting [20] is complemented by a specialized reasoner, a procedure capable of establishing formula satisfiability w.r.t. a fixed theory of interest. The key feature of CCR is that the context of rewriting (i.e. the literals assumed true during rewriting) are manipulated and checked for consistency by the satisfiability procedure.
2. **RDL** is an open system which can be modularly extended with new satisfiability procedures provided these offer certain interface functionalities. As underlying theories currently available there are the universal theory of equality (UTE), the universal theory of linear arithmetic over integers (ULAI), and the theory obtained as the combination of the previous two (UTELAI).
3. **RDL** implements instances of a *generic extension schema* for decision procedures [4]. The key ingredient of such a schema is a *lemma speculation mechanism* which 'reduces' the satisfiability problem of a given theory to the satisfiability problem of one of its sub-theory for which a satisfiability procedure is already available. The current version of the system provides implementations of this schema which enable the satisfiability procedure for ULAI to handle properties of user-defined functions as well as a fragment of arithmetic with multiplication.

RDL is implemented in (SICStus) Prolog and it is freely available via the *Constraint Contextual Rewriting Project* home page at

http://www.mrg.dist.unige.it/ccr

1.2 Related Systems

In Acl2 [11] (as in its predecessor, nqthm [6]), a sophisticated schema to incorporate a decision procedure for linear arithmetic in the simplification activity is implemented. Unfortunately, the design of such a schema heavily depends on the particular characteristics of the host system [7]. As a result, it is not easy to incorporate new decision procedures. Both *Simplify* [9] and pvs [17] feature a bunch of cooperating decision procedures, following the paradigm proposed in [16] and in [19], respectively. In both systems, while it is easy to plug-in new procedures, an insufficient degree of automatization is provided for some classes of proof obligations which frequently arise in practical verification efforts such as some sub-theory of the theory of arithmetic with multiplication. In both systems, the user is forced to supply appropriate lemmas encoding the properties of the interpreted functions (e.g. multiplication). The version of STeP described in [5] implements a rational based version of the Fourier-Motzkin method, extended to handle multiplication by (partial) quantifier elimination and reasoning about the sign of multiplicands. Although STeP offers a high degree of automation for a significant sub-theory of arithmetic with multiplication, it is not flexible enough to provide similar degrees of automation for other theories.

Plan of the Paper

In Section 2, we describe how **RDL** solves a typical verification condition arising in the proof of termination of a function normalizing expressions. This serves the twofold purpose of introducing the concept of (theorem proving) problem solved by **RDL** and of giving a brief overview of the main reasoning activities implemented in the system. In Section 3, we describe how to specify a theorem proving problem to **RDL**. Then, in Section 3, we describe the reasoning activities implemented in the system and their interplay. Finally, in Section 5, we report an excerpt of the experimental results of the system on some typical problems and we compare **RDL** with other state-of-the-art validity checkers.

2 An Example

Consider the problem of showing the termination of the function to normalize conditional expressions in propositional logic as described in [6].

The expressions of the logic are built over propositional constants (denoted in the following with $\mathsf{pl}(N)$, where N is an integer) and the ternary connective “if A then B else C” (denoted in the following with $\mathsf{if}(A, B, C)$, where A, B, and C are variables ranging over the set of expressions of propositional logic). Informally, the function *norm* for normalizing conditional expressions (recursively) remove all the if’s occurring as the first argument of another if by pushing them into the other two arguments of the external if.

The argument in the proof of termination of *norm* is based on exhibiting a measure function that decreases (according to a given well-founded ordering) at each function’s recursive call. For example, *ms* (reported in [18]) is one such a function:

$$\begin{aligned} ms(\mathsf{pl}(N)) &= 1 \\ ms(\mathsf{if}(A, B, C)) &= ms(A) + ms(A) * ms(B) + ms(A) * ms(C) \end{aligned}$$

where $+$ and $*$ denote addition and multiplication over integers. It is easy to check that *ms* enjoys the following property:

$$ms(A) > 0 \tag{1}$$

for each expression A of the logic. The definition of *ms* and property (1) are stated in **RDL** by asserting the following Prolog facts:

```
  fact(bm,msbase,[],ms(pl(N))=1).
  fact(bm,msstep,[],ms(if(A,B,C))=ms(A)+ms(A)*ms(B)+ms(A)*ms(C)).
  fact(bm,msfact,[],ms(A)>0).
```

where `msbase`, `msstep`, and `msfact` are the unique identifiers of the facts in the system and `[]` indicates that the facts are unconditional.

One of the proof obligation expressing the decrease argument is

$$ms(\mathsf{if}(u,\mathsf{if}(v,y,z),\mathsf{if}(w,y,z))) < ms(\mathsf{if}(\mathsf{if}(u,v,w),y,z)), \tag{2}$$

where $<$ denotes the 'less-than' relation over integers and u, v, w, y, and z are expressions of the logic. In **RDL** we can specify the clause (in this case a unit clause) to be checked for validity as follows:

```
input(bm,
     [ms(if(u, if(v,y,z), if(w,y,z))) < ms(if(if(u,v,w), y, z))]).
```

There are still two missing ingredients to complete the specification of our problem to **RDL**. First, we need to provide an informal description of the problem under consideration:

```
description(bm,
'Silvio Ranise',
'Problem taken from the paper
  "Proving Termination of Normalization Functions
   for Conditional Expressions"
 by L C Paulson.').
```

Second, we need to tell **RDL** what satisfiability procedure to use in order to check the validity of the formula:

```
 expected_output(bm, aug_aff(eq_la), rpo, [true]).
```

where `aug_aff(eq_la)` tells **RDL** to use the satisfiability procedure for UTELAI extended with lemma speculation techniques which allow to use the definition of *ms* and its property as well as some properties about multiplication; `rpo` is the recursive path ordering that is going to be used by the system for rewriting[1].

Now, we are in the position to run the system on the specified problem by simply typing

```
 run(bm).
```

RDL's output is reported in Figure 1[2]. Lines 1–8 provide the user with a brief summary of the problem that **RDL** is going to simplify. Line 9 gives the formula obtained by the simplification process implemented by the system. Line 22 shows

[1] To simplify the presentation, we omit the precedence over function symbols needed to completely specify the ordering and, in the following, we assume that such a precedence has been defined so that the rewriting steps described below are possible.

[2] The original output of the system has been slightly edited in order to simplify the discussion that follows.

```
 1 Problem: bm
 2 Reasoning Specialist: combination of the theory of ground
 3                       equality and Linear Arithmetic with
 4                       a combination of augmentation and
 5                       affinization enabled.
 6 Ordering: Recursive Path Ordering.
 7 Input Formula: [ms(if(u,if(v,y,z),if(w,y,z)))<ms(if(if(u,v,w),y,z))]
 8 Expected Formula: [true]
 9 Simplified Formula: [true]
10 Status: ok!
11 Reduction:
12 cl_simp:
13   [id,
14     crew>(crew>(crew>(crew>(crew>(crew>(crew>
15     (normal>
16      cxt_entails_true:[
17       augment_affinize:
18        [crew:[
19         augment_affinize,
20          cs_extend]>
21           (augment_affinize>augment_affinize)]])))))))]
22 Time (Elapsed-Theorem Proving): 610-600 msec
```

Fig. 1. Sample output of **RDL**.

the time used by the system to perform the simplification (600 msec) and the total time (610 msec), i.e. the time to perform simplification as well as the other instructions such as input-output. Lines 11 to 21 describe the simplification steps undertook by the system. First of all, **RDL** initializes the simplification of the clause (2) (`cl_simp` at 12). This consists of building up the context of simplification and of selecting a literal to be simplified (`id` at 13); in this case, the simplification context is empty and the literal being simplified is the only literal in the clause. Then, the simplification process can start.

RDL rewrites the l.h.s. and the r.h.s. of (2) with the definition of ms (the sequence of `crew` at 14) and it obtains the following literal:

$$\begin{array}{l} ms(u) + ms(u) * ms(v) + ms(u) * ms(v) * ms(y) + \\ ms(u) * ms(v) * ms(z) + ms(u) * ms(w) + \\ ms(u) * ms(w) * ms(y) + ms(u) * ms(w) * ms(z) < \\ ms(u) + ms(u) * ms(v) + ms(u) * ms(w) + \\ ms(u) * ms(y) + ms(u) * ms(v) * ms(y) + ms(u) * ms(w) * ms(y) + \\ ms(u) * ms(z) + ms(u) * ms(v) * ms(z) + ms(u) * ms(w) * ms(z) \end{array} \tag{3}$$

RDL then performs all the possible cancellations in (3) (`normal` at line 15) and it obtains:

$$ms(u) * ms(y) + ms(u) * ms(z) < 0. \tag{4}$$

In order to prove the validity of (2), **RDL** checks the unsatisfiability of its negation (`cxt_entails_true` at line 16). To do this, it factorizes (4) to $ms(u) * (ms(y) + ms(z)) < 0$ and then it considers the following instance of a trivial property about the sign of multiplicands:

$$(ms(u) > 0 \wedge ms(y) + ms(z) > 0) \Longrightarrow ms(u) * (ms(y) + ms(z)) > 0 \qquad (5)$$

(this is identified by `augment_affinize` at line 17). In order to make the conclusion of (5) available to the system (`cs_extend` at line 20), it is necessary to relieve its hypotheses (`crew` at line 19). This is easy since **RDL** readily instantiates (1) three times, namely to $ms(u) > 0$, $ms(y) > 0$, and to $ms(z) > 0$ (the three `augment_affinize` at lines 19–21). At this point, it is trivial to detect the unsatisfiability of (4) and the conclusion of (5).

3 Specifying a Problem to RDL

The basic concept underlying **RDL**'s user interface is that of *problem*. Intuitively, a problem provides a specification of the clause to be simplified as well as the satisfiability procedure to be used during simplification and the facts that the user wants to assume valid. Formally, a problem determines a (first-order) language and a (first-order) theory. In particular, a problem specifies which predicate and function symbols are interpreted since either they are known to the satisfiability procedures or they are taken into account by the lemma speculation mechanism. The specification of a problem in **RDL** involves a number of information that must be specified by asserting certain facts (in Prolog parlance) to the system.

`description(TagPb, Author, Descr).` The first argument `TagPb` is the unique label of the problem the user wishes to solve. In **RDL**, each problem must be uniquely identified by a Prolog term, e.g. the Prolog constant `bm` in Figure 1. The other two arguments of the predicate are Prolog strings. In particular, `Author` specifies the name of the author of the problem and `Descr` gives an informal description of the problem.

`input(TagPb, Clause).` The first argument `TagPb` is the unique identifier of the problem. The second argument `Clause` specifies the (ground) clause to be simplified. **RDL** represents clauses as Prolog lists of literals. First-order literals are represented by ground Prolog literals.

`fact(TagPb, TagFact, Conds, Concl).` The first argument `TagPb` is the unique label of the problem. The second argument `TagFact` is the unique label of the fact within the name space of the problem. The last two arguments specify the hypotheses and the conclusion of a conditional fact. In particular, `Conds` is a list of literals and `Concl` is a single literal. In this case, **RDL** represents first order literals as Prolog literals. In particular, first order variables are represented by Prolog variables which can occur only in this position of the specification of a problem. As an example, consider the following Prolog fact:

```
fact(pb,label,[g(X)>0, f(Y,c)=g(Z)],h(X,Y)=Z).
```

It encodes the following formula:

$$\forall\, x\, y\, z\, ((g(x) > 0 \wedge f(y, c) = g(z)) \Longrightarrow h(x, y) = z),$$

where x, y, and z are variables (represented by the Prolog variables `X`, `Y`, and `Z` respectively), c is a constant, g is a unary function symbol, f and h are binary function symbol, $=$ is the equality symbol, and $>$ is an infix binary predicate. The formula specified by `fact` is assumed valid during the simplification activity.

`expected_output(TagPb, RS, Ord, Clause)`. The first argument `TagPb` is the unique label of the problem. The second argument `RS` specify the (possibly extended) satisfiability procedure to be used in order to support the simplification activity. In the actual implementation of **RDL**, `RS` can be one of the following Prolog term:

– `eq` identifies a satisfiability procedure for UTE. The implementation of this procedure is based on the congruence closure algorithm described in [19];
– `la` identifies a satisfiability procedure for ULAI based on the version of Fourier-Motzkin algorithm described in [7];
– `eq_la` identifies a combination of the above two satisfiability procedures based on Nelson and Oppen's combination paradigm [16];
– `aug(SatProc)` (where `SatProc` is either `eq`, `la`, or `eq_la`) identifies the extension of the satisfiability procedure `SatProc` by means of the augmentation mechanism [2, 3], i.e. the capability of making available to the satisfiability procedure selected instances of available lemmas (specified by `fact`);
– `aff(SatProc)` (where `SatProc` is either `la` or `eq_la`) identifies the extension of the satisfiability procedure `SatProc` by means of the affinization mechanism [4], i.e. the capability of making available selected properties of multiplication;
– `aug_aff(SatProc)` (where `SatProc` is either `la` or `eq_la`) identifies a combination of the two extension mechanisms outlined above [4].

We notice that each satisfiability (implicitly) declare a set of interpreted predicate symbols by means of the predicate `pred_sym(TagPb, PredSpec)`. The first argument `TagPb` is the unique identifier of the problem. The second argument `PredSpec` is a Prolog term of the form `p`(_, _, ..., _), where `p` is a predicate symbol together with its arity, namely (_, _, ..., _). For example, the predicate $<$ is automatically declared as interpreted by asserting `pred_sym(_, _<_)`.. This is done each time the satisfiability procedure for ULAI is used during the simplification process, i.e. the constant `la` is given as the second argument of `expected_output`.

The third argument `Ord` is the ordering to be used while rewriting. Two possible ordering can be used in **RDL**: the Knuth-Bendix ordering [12] (identified by the Prolog constant `kbo`) and the recursive-path ordering [8] (identified by the Prolog constant `rpo`). For `kbo`, we need to specify the weight of the symbol by means of the predicate `symbol_weight(TagPb, FSym, N)`, where `TagPb` is the unique label of the problem, `Fsym` is a function (or predicate) symbol, and `N` is a positive natural number (i.e. the weight of the symbol). Furthermore, for both ordering we can specify a precedence relation over function (and predicate) symbols by means of the predicate `ord_gt(TagPb,F1,F2)`, where `TagPb` is the unique label of the problem, `F1` and `F2` are function (or predicate) symbols s.t. `F1` is bigger than `F2` in the precedence relation used to define either `kbo` or `rpo`.

Finally, the last argument `Clause` of `expected_output` specifies the clause which the user thinks **RDL** is going to produce as the result of the simplification activity. This last parameter is not strictly required (it can be left unspecified by using a Prolog variable) and it is mainly used for validating the corpus of problems shipped with the system.

4 The Reasoning Activities of RDL

RDL features a tight integration of three reasoning activities: *contextual rewriting*, *satisfiability checking*, and *lemma speculation*. The sophisticated interactions between these reasoning activities are the key ingredients of the effectiveness of **RDL**'s simplification mechanism. In the following, the various functionalities will be modeled by means of relations whereas the interplay between the various functionalities is specified by an inductive definition by using a set of inference rules.

4.1 Contextual Rewriting

In **RDL**, the rewriter implements (a variant of) contextual rewriting [20]. It manipulates two data structures. The former is the set of literals which can be assumed true during the rewriting activity; the conjunction of these literals is called the (*rewriting*) *context*. The second data structure is a set of conditional rules which are added to **RDL**'s database of valid facts by asserting Prolog facts of the form

```
 fact(pb, name, [h1, ..., hn], l=r).
```

where `h1`, ..., `hn`, and `l=r` are **RDL**'s representation of some first-order literals, which are denoted in the following with $h_1, ..., h_n$, and $l = r$ (respectively).

Now, we are in the position to give a high-level description of the rewriting algorithm implemented in **RDL**. In the following, we assume that $\prec$ is a well-founded ordering on ground terms which allows for a suitable mechanization in **RDL**.

Without loss of generality, assume $r\sigma \prec l\sigma$ for a ground substitution σ. Otherwise, swap l with r (if $l\sigma$ is different from $r\sigma$). Given a literal $p[l\sigma]$, **RDL**'s rewriter returns $p[r\sigma]$ if the following two conditions are satisfied: (i) the ground literals $h_1\sigma, ..., h_n\sigma$, and $p[r\sigma]$ are smaller than $p[l\sigma]$ according to $\prec$; and (ii) each $h_i\sigma$ (for $i = 1, ..., n$) can be simplified to *true* by recursively invoking the activity of contextual rewriting. Checking for the entailment of an instantiated condition can be done in three ways. The first is to recursively invoke the **RDL**'s rewriter on the literal under consideration with the aim of rewriting it to *true*. This informal description can be formalized by means of the following inference rule named `crew`:

$$\frac{C :: h_1\sigma \xrightarrow[\mathsf{ccr}]{} true \quad \cdots \quad C :: h_n\sigma \xrightarrow[\mathsf{ccr}]{} true}{C :: p[l\sigma]_u \xrightarrow[\mathsf{ccr}]{} p[r\sigma]_u}$$

where $h_1 \wedge \cdots \wedge h_n \implies l = r$ is one of the available (possibly conditional) rewrite rules and $C :: p \xrightarrow[\mathsf{ccr}]{} p'$ is a sequent which denotes the mechanization of the one-step contextual rewriting relation, i.e. it takes a literal p in context C and returns the literal p'.

4.2 Satisfiability Checking

In **RDL**, a satisfiability procedure for a given (first-order) theory T_c works on a data structure (called *constraint store*) representing a conjunction of ground literals to be assumed true during **RDL**'s simplification activity. The constraint store is built by interning the literals in the rewriting context. As it will be discussed below, the particular data structure used to implement the constraint store depends on the theory T_c.

There are four functionalities manipulating the constraint store: `cs_init`(C), `cs_unsat`(C), $P :: C \xrightarrow[\mathsf{cs\_extend}]{} C'$, and $C :: p \xrightarrow[\mathsf{cs\_normal}]{} p'$, where C and C' are constraint stores, p and p' are ground literals, and P is a set of ground literals.

The functionalities `cs_init` and `cs_unsat` manipulate a constraint store and are invoked by **RDL**'s rewriter so to synchronize the content of the rewriting context and of the constraint store. More precisely, `cs_init`(C) sets C to the "empty" constraint store and `cs_unsat`(C) is a boolean function recognizing a subset of unsatisfiable (in T_c) constraint stores whose unsatisfiability can be checked by means of a computationally inexpensive (syntactic) check.

The remaining two interface functionalities (i.e. $P :: C \xrightarrow[\mathsf{cs\_extend}]{} C'$ and $C :: p \xrightarrow[\mathsf{cs\_normal}]{} p'$) must satisfy some requirements in order to allow the "plug-and-play" incorporation of new satisfiability procedures in the system. As said above, the constraint store is the result of interning the literals in the rewriting context by invoking the functionality

$$P :: C \xrightarrow[\mathsf{cs\_extend}]{} C'$$

which denotes the computation performed in order to intern the literals in the input set P, (possibly) deriving new literals entailed by the conjunction of the literals in P and C, and adding them to the actual state C of the procedure so to obtain the new state C'. The last functionality

$$C :: p \xrightarrow[\mathsf{cs\_normal}]{} p'$$

provided by the satisfiability procedure computes a normal representation p' of a given literal p w.r.t. the theory T_c and the literals stored in the constraint store C. In order to simplify the integration with the **RDL**'s rewriter, we require that (i) the literal p' returned by this functionality is entailed by T_c and the conjunction of literals in C, and that (ii) p' is smaller (according to the rewriting ordering $\prec$) than p.

Example 1. The satisfiability procedure for ULAI in RDL. Consider the first-order language consisting of the numerals $..., -2, -1, 0, 1, 2, ...$, variables, the function symbol $+$, the (infix) binary predicate symbols $<$, $\leq$, $=$, $\geq$, and $>$, and the usual logical connectives. The intended structure of this language (whose theory is ULAI) interprets numerals as integers[3], variables range over integers, $+$ is interpreted as addition, $<$, $\leq$, $\geq$, and $>$ are interpreted as the usual ordering relations, and $=$ is interpreted as the identity relation.

Let T_c be the first-order theory containing ULAI and n-ary function symbols (other than $+$) interpreted as arbitrary functions from n-tuples of integers to integers.

The Fourier-Motzkin elimination method [13] is based on the idea of eliminating one variable at a time in the hope of obtaining a 'trivially' unsatisfiable inequality such as, e.g., $0 \leq -1$. It can be straightforwardly adapted to obtain a proof procedure for T_c.

Although the exponential worst-case complexity seems to discourage its usage for checking the unsatisfiability of conjunctions of inequalities, the Fourier-Motzkin method can be made usable in practice (as observed e.g. in [7]) by using the simple trick of choosing the variable to eliminate according to a given ordering.

We assume that $<$, $=$, $\geq$, and $>$ (in the language of ULAI) are preliminary eliminated in favor of $\leq$ (e.g. $x < 0$ can be rewritten to $x \leq -1$ by exploiting the integral property of integers). The inequalities in the constraint store are put into the following (normal) form

$$c_1 \cdot m_1 + \cdots + c_n \cdot m_n \leq c \tag{6}$$

where $n \geq 0$ (if $n = 0$, then (6) stands for $0 \leq c$), $c, c_1, ..., c_n$ are relatively prime integers (called *coefficients*), $m_1, ..., m_n$ are (first-order) terms (called *multiplicands*) whose top-most function symbols are different from $+$ s.t. $m_{i+1} \prec m_i$ (where $\prec$ is the ordering used for rewriting which is required to be total over ground terms), and $c_i \cdot m_i$ $(i = 1, ..., n)$ abbreviates the term $m_i + \cdots + m_i$ in which m_i occurs c_i times.

A constraint store is a data base of inequalities of the form (6) indexed by the key multiplicands. More precisely, each key multiplicand points to two lists of inequalities: one contains inequalities where the coefficient of the key multiplicand is positive whereas the other contains inequalities where the coefficient of the key multiplicand is negative. If we derive an inequality of the form $0 \leq c$, where c is a negative integer, we stop the exhaustive elimination of variables and we set a flag signaling the unsatisfiability of the constraint store. The functionality `cs_init`(C) is defined so to set up the empty data base of inequalities and `cs_unsat`(C) returns true when the flag of the unsatisfiability of the constraint store is true.

Let ι and ι' be two inequalities of the form (6) both having m as their heaviest multiplicand, k (k') be the coefficient of m in ι (ι', resp.), k and k' be of opposite

[3] In the following, to simplify the discussion, we will use the term 'integer' in place of 'numeral'.

sign, and $elim(\iota, \iota')$ be the normal form of the linear combination of ι and ι' not containing m. Now, we are in the position to describe an implementation of the functionality $P :: C \xrightarrow[\text{cs\_extend}]{} C'$. First of all, put the literals of P into inequalities of the form (6) and insert them into C at appropriate positions. Then, close the resulting data base under the operation $elim$ so to obtain C', i.e. C' is that for any ι_1, ι_2 in C' we have that $elim(\iota_1, \iota_2)$ is in C' (if $elim$ is defined).

Finally, we notice that it is possible to extend the Fourier-Motzkin algorithm in order to derive equalities entailed by inequalities [13] (stored in the constraint store). A set of entailed equalities is created as soon as an inequality of the form $0 \leq 0$ is derived; all the inequalities which contributed to create this inequality are turned into equalities (see, e.g. [13] for more details). It is natural to exploit these equalities to simplify the literal which is currently rewritten. This observation offers an immediate implementation of $C :: p \xrightarrow[\text{cs\_norm}]{} p'$. In fact, if we extend the data base C of inequalities to store also the entailed equalities, we can use them as rewrite rules since we are always capable of orienting them; the entailed equalities are ground and the ordering $\prec$ is assumed to be total on ground terms.

Another example of implementation of $C :: p \xrightarrow[\text{cs\_norm}]{} p'$ is given by the algebraic manipulation required to perform the cancellation in (3) to derive (4) in Section 2.

4.3 Combining Rewriting and Satisfiability Checking

Now, we are in the position to describe how the functionalities provided by the satisfiability procedure are exploited by the rewriting activity of **RDL**.

A given literal can be rewritten to $true$ in a given context if it is entailed by the context. In turn, the check for entailment of a literal l by a conjunction of literals C can be performed by checking the unsatisfiability of $C \wedge \neg l$. In **RDL**, this can be easily done by invoking `cs_unsat` on the constraint store obtained by adding the negation of the literal being rewritten. Similarly, we can rewrite to $false$ a literal if its negation is entailed by the context. This kind of reasoning can be formalized by the following two inference rules, named `cxt_entails_true` and `cxt_entails_false` (read from left to right):

$$\frac{\{\neg p\} :: C \xrightarrow[\text{cs\_extend}]{} C'}{C :: p \xrightarrow[\text{ccr}]{} true} \text{ if } \texttt{cs\_unsat}(C') \qquad \frac{\{p\} :: C \xrightarrow[\text{cs\_extend}]{} C'}{C :: p \xrightarrow[\text{ccr}]{} false} \text{ if } \texttt{cs\_unsat}(C')$$

In **RDL**, there is another mechanism of rewriting realized by simply invoking the functionality for normalization provided by the satisfiability procedure. This can be formalized by the following inference rule, named `normal`:

$$\frac{C :: p \xrightarrow[\text{cs\_normal}]{} p'}{C :: p \xrightarrow[\text{ccr}]{} p'}$$

We notice that the inference rules `cxt_entails_true`, `cxt_entails_false`, and `normal` extends the definition of the relation $C :: p \xrightarrow[\text{ccr}]{} p'$ modeling the activity of contextual rewriting as introduced in Section 4.1.

4.4 Lemma Speculation

Three instances of the lemma speculation mechanism described in [4] are implemented in **RDL**. All the instances share the goal of feeding the satisfiability procedure with new facts about function symbols which are otherwise uninterpreted in the theory in which the satisfiability procedure works. More precisely, they inspect the context C and return a set of ground facts entailed by C. The lemma speculation activity of computing a set S of ground facts given a constraint store C can be modeled by the following relation:

$$C \mapsto \langle C', S \rangle,$$

where C' is a constraint store which differs from C in the fact that some literals are marked as already used (this is useful to avoid infinite looping by reconsidering infinitely often the same literals for deriving new facts).

Augmentation. It extends the information available to the satisfiability procedure with selected instances of lemmas encoding properties of symbols the decision procedure does not know anything about. For example, by devising a suitable set of lemmas about multiplication, it is possible to enable a procedure for ULAI to handle formulas whose satisfiability depends on properties of multiplication (e.g. multiplying two positive integers we obtain a positive integer).

The crucial step for the success of augmentation is the selection of suitable instances of the available formulas. This is an instance of the more general problem of choosing suitable instances of lemmas for guiding a generic prover to a successful proof. Unfortunately, for such a problem no general satisfactory solution does exist. In **RDL**, for our particular instance, we implemented the heuristics of finding instances of the conclusions of the available conditional lemmas promoting further computations (e.g. further Fourier-Motzkin elimination steps in the case of the procedure for ULAI) when added to the current state of the satisfiability procedure.

A further problem is the presence of extra variables in the hypotheses (w.r.t. the conclusion) of lemmas. **RDL** avoids this problem by requiring that the conclusion contains all the variables occurring in the lemma and that all the variables get instantiated by matching the conclusion of the lemma against the largest (according to $\prec$) literal in C.

We notice that augmentation critically depends on the shape of the available lemmas and the algorithm implemented by the satisfiability procedure. If a suitable set of lemmas is defined, then augmentation dramatically widens the scope of a satisfiability procedure. Unfortunately, devising such a suitable set is a time consuming activity. This problem can be solved in some important special cases such as some fragments of arithmetics with multiplication.

Affinization. In the actual version of **RDL**, affinization implements the 'on-the-fly' generation of lemmas about multiplication over integers. We emphasize that the user is no more required to provide suitable lemmas about properties of multiplication since instances of some classes of properties are automatically generated.

To understand how affinization works, consider the non-linear inequality $x * y \leq -1$ (where x and y range over integers). By resorting to its geometrical interpretation, it is easy to verify that (over integers) $x * y \leq -1$ is equivalent to $(x \geq 1 \wedge y \leq -1) \vee (x \leq -1 \wedge y \geq 1)$. To avoid case splitting, we observe that the semi-planes represented by $x \geq 1$ and $x \leq -1$ as those represented by $y \leq -1$ and $y \geq 1$ are non-intersecting. This allows to derive the following four lemmas: $x \geq 1 \Longrightarrow y \leq -1$, $x \leq -1 \Longrightarrow y \geq 1$, $y \geq 1 \Longrightarrow x \leq -1$, and $y \leq -1 \Longrightarrow x \geq 1$. This process can be generalized to non-linear inequalities which can be put in the form $x * y \leq k$ (where k is an integer) by factorization [15]. The generated (conditional) lemmas are used as for augmentation.

A Combination of Augmentation and Affinization. On the one hand affinization can be seen as a significant improvement over augmentation since it does not require any user intervention. On the other hand it fails to apply when inequalities cannot be transformed into a form suitable for affinization. **RDL** combines augmentation and affinization by considering the function symbols occurring in the constraint store C, i.e. the top-most function symbol of the largest (according to $\prec$) literal in C triggers the invocation of either affinization or augmentation.

4.5 Combining Rewriting and Lemma Speculation

The main obstacle to using the facts resulting from the lemma speculation activity is that such facts are conditional. Hence, we should preliminary relieve their hypotheses in order to be entitled to make their conclusions available to the satisfiability procedure. In **RDL**, we solve this problem by rewriting each hypothesis to *true* (if possible) by invoking the rewriter. (Notice that this implies that the rewriter and the satisfiability procedure are mutually recursive.) The above reasoning activity can be formalized by the following inference rule, named either `augment`, `affinize`, or `augment_affinize` depending on which lemma speculation mechanism is specified:

$$\frac{C' :: g_1 \xrightarrow[\mathsf{ccr}]{} true \quad \cdots \quad C' :: g_n \xrightarrow[\mathsf{ccr}]{} true \quad \{c_1, ..., c_m\} :: C \xrightarrow[\mathsf{cs\_extend}]{} C'}{P :: C \xrightarrow[\mathsf{cs\_extend}]{} C'} \text{ if } C \mapsto \langle C', S \rangle$$

where $(g_1 \wedge \cdots \wedge g_n) \Longrightarrow (c_1 \wedge \cdots \wedge c_m)$ is in S and it is ground (for $n \geq 1$ and $m \geq 1$).

5 Experiments

RDL must be judged w.r.t. its effectiveness in simplifying (and possibly checking the validity of) proof obligations arising in practical verification efforts where decision procedures play a crucial role. Although standard benchmarks for theorem provers such as TPTP can be (partially) tackled by **RDL**, we prefer to evaluate **RDL**'s performances on proof obligations extracted from real verification efforts. To do this, we are building a corpus of proof obligations taken from the literature

Table 1. Experimental Results.

#	Problem	**RDL**
1	$f(A) = f(B) \Longrightarrow (r(g(A,B),A) = A) \vdash$ $r(g(y,z),x) = x \vee \neg(g(x,y) = g(y,z)) \vee \neg(y = x)$	26
2	$A * B = B * A, (\neg(C = 0)) \Longrightarrow (rem(C * D, C) = 0) \vdash$ $rem(y * z, x) = 0 \vee \neg(x * y = z * y) \vee x = 0$	109
3	$(A > 0) \Longrightarrow (rem(A * B, A) = 0) \vdash\ rem(x * y, x) = 0 \vee x \leq 0$	12
4	$min(A) \leq max(A) \vdash$ $\neg(k \geq 0) \vee \neg(l \geq 0) \vee \neg(l \leq min(b)) \vee \neg(0 < k) \vee l < max(b) + k$	12
5	$(memb(A,B)) \Longrightarrow (len(del(A,B)) < len(B)) \vdash$ $\neg(w \geq 0) \vee \neg(k \geq 0) \vee \neg(z \geq 0) \vee \neg(v \geq 0) \vee \neg(memb(z,b))$ $\vee \neg(w + len(b) \leq k) \vee w + len(del(z,b)) < k + v$	17
6	$(0 < A) \Longrightarrow (B \leq A * B), 0 < ms(C) \vdash$ $ms(c) + ms(d)^2 + ms(b)^2 < ms(c) + ms(b)^2 + 2ms(d)^2 * ms(b) + ms(d)^4$	72
7	$A \geq 4 \Longrightarrow (A^2 \leq 2^A) \vdash \neg(c \geq 4) \vee \neg(b \leq c^2) \vee \neg(2^c < b)$	14
8	$(max(A,B) = A) \Longrightarrow (min(A,B) = B), (p(C)) \Longrightarrow (f(C) \leq g(C)) \vdash$ $\neg(p(x)) \vee \neg(z \leq f(max(x,y))) \vee \neg(0 < min(x,y)) \vee \neg(x \leq max(x,y)) \vee$ $\neg(max(x,y) \leq x) \vee z < g(x) + y$	114
9	$0 < ms(C) \vdash$ $ms(c) + ms(d)^2 + ms(b)^2 < ms(c) + ms(b)^2 + 2ms(d)^2 * ms(b) + ms(d)^4$	63
10	$\vdash\ x \geq 0 \Longrightarrow x^2 - x + 1 \neq 0$	40

and from the examples available for similar systems. The problems in our corpus are representative of various verification scenarios and are considered difficult for current state-of-the-art verification systems.

Table 1 reports a selection of the results of our computer experiments. Problem lists the available lemmas[4] (if any) and the formula to be decided. $\vdash$ is the binary relation characterizing the deductive capability of **RDL** (we have that $\vdash$ is contained in $\models_T$, where T is the theory decided by the decision procedure extended with the available facts). The last column records the time (expressed in msec) used by **RDL** to solve a problem[5].

RDL solves problems 1 and 2 with the procedure for UTE. In the former, the procedure is used to derive equalities entailed by the context which are used as rewrite rules and enable the use of the available lemma. The ordered rewriting engine implemented by **RDL** is a key feature to successfully solve problem 2 since this form of rewriting allows to handle usually non-orientable rewrite rules such as $A * B = B * A$. **RDL** solves problem 3 with a satisfiability procedure for ULAI extended with augmentation. Infact, the available lemma is applied once its instantiated condition, namely $x > 0$, is relieved by the decision procedure (it is straightforward to check the inconsistency of $x > 0$ and the literal $x \leq 0$ in the

[4] Capitalized letters denote implicitly universally quantified variables.

[5] Benchmarks run on a 600 MHz Pentium III running Linux. **RDL** is implemented in Prolog and it was compiled using SICStus Prolog, version 3.8.

context). **RDL** solves problems 4, 5, 6, and 7 with a procedure for ULAI extended with the augmentation mechanism. In particular, the formula of problem 6 is a non-linear formula whose validity is successfully established by **RDL** in a similar way of the example in Section 2. **RDL** solves problem 8 with the combination of procedures for ULAI and UTE. **RDL** solves problems 9 and 10 with the procedure for ULAI, extended with both augmentation and affinization. The lemma about multiplication (i.e. $0 < I \Longrightarrow J \leq I * J$) is supplied in problem 6 but it is not in problem 9. Only the combination of augmentation and affinization can solve problem 9. Finally, problem 10 shows the importance of the context in which proof obligations are proved (since **RDL** does not case-split). In fact, without $x \geq 0$ `augment` and `affinize` would not be able to solve problem 10.

As a matter of fact, the online version of STeP fails to solve all of the problems reported in Table 1. However, most of them are successfully solved by the improved version of STeP described in [5]. This version solves problems 9 and 10 by resorting to a partial method for quantifier elimination (see [5] for details). Instead, our affinization mechanism is capable of proving the formula with simpler mathematical techniques. The comparison is somewhat difficult since the method used by STeP works over the rationals and our affinization technique only works over integers. *Simplify* successfully solves problems 1 to 8 thanks to a Nelson-Oppen combination of decision procedures and an incomplete matching algorithm which is capable of instantiating (valid) universally quantified clauses. However, it does not solve problems 9 and 10 since it is unable to handle non-linear facts without user-supplied lemmas (such as $0 < I \Longrightarrow J \leq I * J$ in problem 6). Finally, SVC fails to solve all the problems involving augmentation and affinization since it does not provide a mechanism to take into account facts which partially interpret user-defined function symbols.

References

1. A. Armando, L. Compagna, and S. Ranise. System Description: RDL—Rewrite and Decision procedure Laboratory. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR2001)*, pages 663–669. LNAI 2083, Springer-Verlag, 2001.
2. A. Armando and S. Ranise. Constraint Contextual Rewriting. In *Proc. of the 2nd Intl. Workshop on First Order Theorem Proving (FTP'98)*, 1998.
3. A. Armando and S. Ranise. Termination of Constraint Contextual Rewriting. In *Proc. of the 3rd Intl. W. on Frontiers of Comb. Sys.'s (FroCos'2000)*, volume 1794, pages 47–61. Springer-Verlag, 2000.
4. A. Armando and S. Ranise. A Practical Extension Mechanism for Decision Procedures: the Case Study of Universal Presburger Arithmetic. *Journal of Universal Computer Science*, 7(2):124–140, February 2001. Special Issue on Tools for System Design and Verification (FM-TOOLS'2000).
5. N. S. Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Computer Science Department, Stanford University, 1998.
6. R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
7. R.S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.

8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Hand. of Theoretical Comp. Science*, pages 243–320. 1990.
9. D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. Technical report, DEC, 1996.
10. D. Kapur, D.R. Musser, and X. Nie. An Overview of the Tecton Proof System. *Theoretical Computer Science*, Vol. 133, October 1994.
11. M. Kaufmann and J S. Moore. Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Soft. Eng.*, 23(4):203–213, 1997.
12. D. E. Knuth and P. E. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, Oxford, 1970. Pergamon Press.
13. J.-L. Lassez and M.J. Maher. On Fourier's Algorithm's for Linear Arithmetic Constraints. *J. of Automated Reasoning*, 9:373–379, 1992.
14. Z. Manna and the STeP Group. STeP: The Stanford Temporal Prover. Technical Report CS-TR-94-1518, Stanford University, June 1994.
15. V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. Technical Report CS-TR-3109.1, Dept. of Computer Science, University of Maryland, 1994.
16. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. Technical Report STAN-CS-78-652, Stanford Computer Science Department, April 1978.
17. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
18. L. C Paulson. Proving termination of normalization functions for conditional expressions. *J. of Automated Reasoning*, pages 63–74, 1986.
19. R.E. Shostak. Deciding Combination of Theories. *J. of the ACM*, 31(1):1–12, 1984.
20. H. Zhang. Contextual Rewriting in Automated Reasoning. *Fundamenta Informaticae*, 24(1/2):107–123, 1995.