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 – Université Henri Poincaré-Nancy 2, 615, rue du Jardin Botanique, BP 101, 54602 Villers les Nancy Cedex, France

Abstract. The incorporation of reasoning specialists (such as, e.g., decision procedures) in automated deduction systems (such as, e.g., paramodulation based theorem provers) is considered to be a key ingredient to significantly improve the success rate of the available theorem proving strategies. In this paper, we present **RDL** (**R**ewrite and **D**ecision procedure **L**aboratory), a deduction system whose main design goal is twofold: (*i*) to experiment with the "plug-and-play" incorporation of satisfibility procedures in rewriting and (*ii*) to study suitable mechanisms for widening the scope of the procedures. We describe the basic concepts underlying **RDL**'s interface and how the various reasoning modules cooperate to build the powerful deductive mechanism of the system. Finally, we report on some experimental results which confirm the flexibility and the effectiveness of **RDL**.

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, e.g., theorem provers) with the efficiency of specialized ones (such as, e.g., decision procedures). This is witnessed by the fact that many theorem provers developed for verification purposes (such as Acl2 [12], PVS [18], Simplify [9], STeP [15], and Tecton [11]) incorporate decision procedures for ubiquitous theories such as the theory of equality, (decidable fragments of) arithmetics, lists, and arrays. However—as pointed out in [7]—to obtain an effective integration is far from being a trivial task and the solutions adopted in the verification systems listed above are not completely satisfactory. This is mainly for two reasons.

- Some of the approaches apply to decision procedures for specific theories only and therefore are not immediately applicable to decision procedures for different theories.
- It is often the case that only a tiny portion of the proof obligations arising in practical applications fall exactly into the domain the specialized reasoners are designed to solve. As a consequence, the available decision procedures need to be supplemented with mechanisms capable of widening their scope.

To overcome such difficulties in [1, 2] we put forward a generalized form of contextual rewriting [23], called *Constraint Contextual Rewriting* (CCR for short), which allows the available decision procedure to access and manipulate the rewriting context. The main features of CCR are the following.

1. CCR is independent from the theory decided by the decision procedure and therefore its applicability is not restricted to decision procedures for certain theories. We use the notation CCR(X) (by analogy with the CLP(X) notation used to denote the Constraint Logic Programming paradigm [10]) to stress this fact.
2. CCR features a powerful mechanism capable of extending the information available to the decision procedure with facts encoding properties of symbols the decision procedure does not know anything about. This mechanism can greatly widen the scope of the decision procedure thereby dramatically improving its effectiveness.
3. CCR is sound, terminating, and it can be used as a powerful simplification rule to implement sophisticated forms of subsumption and tautology checking modulo a rich background theory.

In order to experiment with CCR we have built **RDL** (acronym for **R**ewrite and **D**ecision procedure **L**aboratory), an automatic theorem prover based on CCR. Thus by design **RDL** inherits all the nice properties of CCR. More in detail:

- **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 quantifier-free theory of equality (UTE), the quantifier-free theory of Presburger arithemtic over integers (UPAI), and the theory obtained as the combination of the previous two (UTEPAI).
- **RDL** implements instances of a *generic extension schema* for decision procedures [3] based on a *lemma speculation mechanism* that 'reduces' the satisfiability problem of a given theory to the satisfiability problem of one of its subtheory for which a satisfiability procedure is already available. The current version of the system provides instances of such a schema that enable the satisfiability procedures for UPAI to handle properties of user-defined functions as well as a fragment of the theory of arithmetic with multiplication.

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

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

Related systems

Both ACL2 [12] and its predecessor NQTHM [6] feature a tightly integrated decision procedure for UPAI into their simplifier. However the integration schema employed heavily depends both on the idiosyncrasies of the host systems as well

as on the specific theory decided by the decision procedure. As a result, the schema cannot be readily applied in different contexts and/or with decision procedures for different theories. Both *Simplify* [9] and PVS [18] feature a bunch of cooperating decision procedures, following the paradigm proposed in [17] and in [21]. 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, e.g., some fragments of arithmetics with multiplication. In both systems, the user is forced to supply appropriate lemmas encoding the properties of the interpreted function such as the 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 fragment of arithmetics with multiplication, it is not flexible enough to provide similar services for other theories.

Plan of the paper

In Section 2, we show the usage of **RDL** on a typical verification effort: the proof of termination of a function that normalizes conditional 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. The kind of theorem proving problems dealt by **RDL** is then precisely defined in Section 3 and the reasoning activities implemented in the system are described in Section 4. In Section 5, we report on the application of the system on some typical problems and and we compare **RDL** with other state-of-the-art validity checkers. Finally, in Section 6, we draw some conclusions and we sketch the future work.

2 A Worked-out Example

Consider the problem of showing the termination of a function, *norm*, that normalizes conditional expressions as described in Chap. IV of [6]. The set of expressions being considered is the smallest set containing denumerably many propositional constants of the form $\mathsf{pl}(N)$ where N is an integer and such that if A, B, and C are conditional expressions, then also $\mathsf{if}(A, B, C)$ (read "if A then B else C") is. The normalization activity carried out by *norm* amounts to the exhaustive application of the following rewrite rule:

$$\mathsf{if}(\mathsf{if}(U, V, W), Y, Z) \longrightarrow \mathsf{if}(U, \mathsf{if}(V, Y, Z), \mathsf{if}(W, Y, Z))$$

For the termination of *norm* it suffices to show the existence of a measure function that decreases (according to a given well-founded ordering) at each function's recursive call. For example, *ms* (reported in [19]) 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 conditional expression A. Both the definition of ms and its property (1) can be stated in **RDL** by asserting the following Prolog facts:

```
fact(bmchIX,msbase,[],ms(pl(N))=1).
fact(bmchIX,msstep,[],ms(if(A,B,C))=ms(A)+ms(A)*ms(B)+ms(A)*ms(C)).
fact(bmchIX,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 obligations expressing the 'decreaseness' 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 variables ranging over conditional expressions. In order to tell **RDL** to consider (2) as a proof obligation it suffices to assert the following fact:

```
input(bmchIX,
      [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(bmchIX,
'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** which satisfiability procedure to use in order to check the validity of the formula:

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

where `aug_aff(eq_la)` tells **RDL** to use the satisfiability procedure for UTEPAI extended with lemma speculation tecniques that enable the use of the available `facts` (i.e. both the definition of ms and (1)) as well as some properties about multiplication. The tag `rpo` indicates that the ordering relation to be used during rewriting is a recursive path ordering.[1]

In order to run the system on the specified problem it suffices to type: `run(bmchIX)`. The output generated by **RDL** is given in Figure 1.[2] Lines 1–

[1] To simplify the presentation, we omit the procedence over function symbols needed to completely specify the ordering and we assume it to be such that all the rewriting steps described in the following are possible.

[2] The output has been slightly edited in order to simplify the ensuing discussion.

```
 1  Problem: bmchIX
 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**

8 provide the user with a brief description 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 the time used by the system to perform the simplification (600 msec) and the total time (610 msec) that includes input-output. Lines 11 to 21 outline the simplification steps carried out by the system: `cl_simp` at line 12 represents the beginning of clause simplification which builds up the rewriting context (empty in this case, as the input clause is unit) and selects the literal to be simplified (`id` at line 13). Then, the rewriting activity starts. Both sides of (2) are simplified by means of the definition of ms (the sequence of `crew` at line 14) and the following literal is obtained:

$$\begin{aligned} ms(u) + ms(u) * ms(v) + ms(u) * ms(v) * ms(y) + ms(u) * ms(v) * ms(z) + \\ ms(u) * ms(w) + ms(u) * ms(w) * ms(y) + ms(u) * ms(w) * ms(z) < \\ ms(u) + ms(u) * ms(v) + ms(u) * ms(w) + ms(u) * ms(y) + \\ ms(u) * ms(v) * ms(y) + ms(u) * ms(w) * ms(y) + \\ ms(u) * ms(z) + ms(u) * ms(v) * ms(z) + ms(u) * ms(w) * ms(z). \end{aligned} \quad (3)$$

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

$$ms(u) * ms(y) + ms(u) * ms(z) > 0. \quad (4)$$

In order to prove the validity of (4), **RDL** checks the unsatisfiability of its negation (`cxt_entails_true` at line 16), i.e. of

$$ms(u) * ms(y) + ms(u) * ms(z) \leq 0. \tag{5}$$

To do this, it factorizes (5) to $ms(u) * (ms(y) + ms(z)) \leq 0$ and then exploits the following fact about the sign of multiplicands:

$$(ms(u) > 0 \wedge ms(y) + ms(z) > 0) \Longrightarrow ms(u) * (ms(y) + ms(z)) > 0 \tag{6}$$

(this is identified by `augment_affinize` at line 17). Fact (6) is automatically generated by the lemma speculation mechanism implemented in **RDL**. In order to make the conclusion of (6) 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, thereby getting $ms(u) > 0$, $ms(y) > 0$, and $ms(z) > 0$ (the three `augment_affinize` at lines 19–21). At this point, it is trivial to detect the unsatisfiability of the conjunction of (5) and the conclusion of (6).

3 Problem Specification in RDL

A problem provides a specification of the clause to be simplified, the satisfiability procedure to be used during simplification, and a set of facts assumed valid. Implicitely, it identifies which predicate and function symbols are interpreted for they are known to a satisfiability procedure (e.g. $<$ is interpreted as the usual 'less-than' relation over integers when the satisfiability procedure for UPAI is selected) or they are taken into account by a particular instance of the lemma speculation mechanism (e.g. the symbol $*$ is interpreted as the multiplication over integers when extending the satisfiability procedure for UPAI by means of the affinization technique). In practice, a problem is specified to **RDL** by asserting the following Prolog facts.

`description(`*TagPb*`,` *Author*`,` *Descr*`)`. The first argument, *TagPb*, is a tag uniquely identifying the problem (e.g. the Prolog constant `bmchIX` in Figure 1). The second argument, *Author*, is a string specifying the name of the author of the problem. The third argument, *Descr*, is a string containing an informal description of the problem.

`input(`*TagPb*`,` *Clause*`)`. The second argument, *Clause*, is a list of literals representing a clause. The fact states that *Clause* is the clause to be simplified in problem *TagPb*.

`fact(`*TagPb*`,` *TagFact*`,` *Conds*`,` *Concl*`)`. The fourth argument, *Concl*, is a literal and the third argument, *Conds*, is a list of literals. The fact states that the formula

$$\bigwedge_{C \in Conds} C \Longrightarrow Concl$$

is a valid fact in problem *TagPb* and that *TagFact* is a tag uniquely identifying it within the name space of the problem. Prolog variables occurring in *Conds* or in *Concl* are intended as (implicitly) universally quantified. For instance, the following fact

```
fact(pb1,t3,[g(X)>0, f(Y,c)=g(Z)],h(X,Y)=Z).
```

states that the formula

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

is identified by the tag `t3` in problem `pb1` and it is to be considered as valid in `pb1`.

`expected_output(`*TagPb*, *RS*, *Ord*, *Clause*`)`. This fact specifies that the results of simplifying the clause associated with *TagPb* with reasoning specialist *RS* and ordering *Ord* is expected to be *Clause*.[3] A *reasoning specialist* is any of the available satisfiability procedures possibly extended with one of the available lemma speculation mechanisms. In the current implementation of **RDL**, *RS* can take one of the following values:

`eq` identifies a satisfiability procedure for UTE. The implementation of this procedure is based on the congruence closure algorithm described in [21];

`la` identifies a satisfiability procedure for UPAI 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 [17];

`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 [1, 2], i.e. the capability of making available to the satisfiability procedure selected instances of the 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 [3], i.e. the capability of making available some properties about multiplication;

`aug_aff(`*SatProc*`)` (where *SatProc* is either `la` or `eq_la`) identifies the extension of the satisfiability procedure *SatProc* by means a combination of the two extension mechanisms outlined above [3].

Two orderings for rewriting are currently supported in **RDL**: the Knuth-Bendix ordering [13] (identified by the tag `kbo`) and the recursive-path ordering [8] (identified by the tag `rpo`). For `kbo`, we need to specify the weight of each symbol by means of the predicate `symbol_weight(`*TagPb*, `_`, *F*, *N*`)`, where *F* is a function (or predicate) symbol and *N* is a positive natural number (i.e.

[3] The last argument, *Clause*, is not strictly required (it can be left unspecified by using a Prolog variable) and it is mainly used for testing **RDL** against the corpus of problems shipped with the system.

the weight of the symbol). For both orderings, the precedence relation over function (and predicate) symbols can be specified by means of the predicate `ord_gt(`$TagPb, \_, F, G$`)` meaning that symbol F is greater than symbol G in the precedence relation.

4 The reasoning activities of RDL

RDL features a tight integration of three reasoning activities: *satisfiability checking*, *contextual rewriting*, and *lemma speculation*. The sophisticated interaction between these reasoning activities is the key to the effectiveness of the resulting simplification.

4.1 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 which are assumed true during rewriting. The constraint store is built by interning the literals in the rewriting context. For efficiency, the data structure used to implement the constraint store depends on the theory T_c.

Here we provide an abstract characterization of the functionalities provided by a satisfiability procedure in terms of the following set of relations involving the constraint store.

`cs_init`(C) denotes the empty constraint store and hence models the activity of initializing the constraint store.

`cs_unsat`(C) denotes the set of T_c-unsatisfiable constraint stores whose T_c-unsatisfiability can be checked by means of a computationally inexpensive syntactic check.

$P :: C \xrightarrow[\mathsf{cs-extend}]{} C'$ models the activity of extending the constraint store C with a set of literals P yielding a new constraint store C'.

$C :: p \xrightarrow[\mathsf{cs-normal}]{} p'$ models the activity of computing a normal representation of an expression p w.r.t. the information stored in the constraint store C.

Example 1 (A Satisfiability Procedure for UPAI). Consider the first-order language consisting of the numerals $\ldots, -2, -1, 0, 1, 2, \ldots$, variables, the function symbol $+$, and the (infix) binary predicate symbols $<$, $\leq$, $=$, $\geq$, and $>$. The intended structure of this language (whose theory is UPAI) interprets numerals as integers,[4] 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 UPAI and n-ary function symbols (other than $+$) interpreted as arbitrary functions from n-tuples of integers to integers.

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

The Fourier-Motzkin elimination method [14] 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 adapted to obtain a proof procedure for T_c along the lines of [20].

We assume that $<$, $=$, $\geq$, and $>$ (in the language of UPAI) 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

$$c_1 \cdot m_1 + \cdots + c_n \cdot m_n \leq c \tag{7}$$

where $n \geq 0$ (if $n = 0$, then (7) stands for $0 \leq c$), $c, c_1, ..., c_n$ are relatively prime integers (called *coefficient*s), $m_1, ..., m_n$ are (first-order) terms (called *multiplicand*s) whose top-most function symbols are different from $+$ s.t. $m_{i+1} \prec m_i$ (where $\prec$ is the ordering used for rewriting, see Section 4.2 for details), 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 set of inequalities of the form (7) 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 signalling 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 (7) both having m as their heaviest multiplicand, k (k') is the coefficient of m in ι (ι', resp.), k and k' are of opposite sign, and $elim(\iota, \iota')$ is 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 $P :: C \xrightarrow[\mathsf{cs-extend}]{} C'$. First of all, put the literals in P into inequalities of the form (7) 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 such that for any ι_1 and ι_2 in C' we have $elim(\iota_1, \iota_2) \in C'$ (if $elim$ is defined).

The Fourier-Motzkin algorithm can be extended to extract equalities entailed by the constraint store as soon as we observe that all the inequalities contributing to the derivation of an inequality of the form $0 \leq 0$ can be turned into equalities (see [14] for details). The equalities so obtained are indeed entailed by the constraint store and it is always possible to orient them into rewrite rules since they are ground and the ordering $\prec$ is assumed to be total on ground terms (again, see Section 4.2 for details). This observation offers an obvious implementation of $C :: p \xrightarrow[\mathsf{cs-normal}]{} p'$. $\triangle$

4.2 Constraint Contextual Rewriting

The activity of constraint contextual rewriting is modeled by the ternary relation $C :: p \underset{\text{ccr}}{\longrightarrow} p'$. If $C :: p \underset{\text{ccr}}{\longrightarrow} p'$ then we say that p' is the result of constraint contextually rewriting p in context C. There are three possible ways of constraint contextually rewriting an expression in a given context: checking for entailment, ordered rewriting, and normalization.

Checking for entailment. The literal p can be rewritten to $true$ in a given context C if p is entailed by C. In order to check whether a literal p is entailed by the constraint store C, we can check the unsatisfiability of $C \wedge \neg p$. In **RDL**, this can be easly done by invoking `cs_unsat` on the constraint store C' resulting from the extension of the current constraint store C with the negation of the literal p currently being rewritten. Similarly, we can rewrite to $false$ the literal p if its negation is entailed by the context C. This kind of reasoning is formalized by the following two inference rules, named `cxt_entails_true` and `cxt_entails_false` (read from left to right):

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

Ordered rewriting. Given a literal $p[l\sigma]$, **RDL**'s rewriter returns $p[r\sigma]$ if the following condition is satisfied; there exists a (possibily conditional) rewrite rule of the form $h_1 \wedge \cdots \wedge h_n \Longrightarrow l = r$ in the set of facts assumed valid by **RDL** and a ground substitution σ such that each $h_i\sigma$ (for $i = 1, ..., n$) can be simplified to $true$ by recursively invoking the activity of constraint contextual rewriting, and the ground literals $h_1\sigma, ..., h_n\sigma$, and $p[r\sigma]$ are smaller than $p[l\sigma]$ according to a well-founded ordering $\prec$ which is total on ground terms. This activity can be formalized by the following inference rule, named `crew`:

$$\frac{C :: h_1\sigma \underset{\text{ccr}}{\longrightarrow} true \;\; \cdots \;\; C :: h_n\sigma \underset{\text{ccr}}{\longrightarrow} true}{C :: p[l\sigma]_u \underset{\text{ccr}}{\longrightarrow} p[r\sigma]_u}$$

Normalization. The third form of constraint contextual rewriting is obtained by exploiting the normalization capability of the satisfiability procedure:

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

provided that $p' \prec p$, or—in other words—normalization should be compatible with ordered rewriting.

4.3 Lemma speculation

By lemma speculation we mean the activity of finding out and 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, lemma speculation amounts to inspecting the context C and then to return a set of ground lemmas S entailed by C. This activity is modelled by the 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).

The current version of **RDL** implements the following sophisticated forms of lemma speculation (see [3] for a detailed description): augmentation, affinization, and a combination of them.

Augmentation. Augmentation extends the information available to the satisfiability procedure with selected instances of lemmas encoding properties of symbols the satisfiability 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 UPAI to handle formulae whose satisfiability depends on properties of multiplication. (An example of such a property is that the result of the multiplicaton of two positive integers is positive.)

The crucial step for the success of augmentation is the selection of suitable instances of the available facts. This is an instance of the more general problem of choosing suitable lemmas for guiding a generic prover to a successful proof. Unfortunately, for such a problem no general satisfactory solution does exist. In **RDL**, we solved the problem by implementing the heuristics of finding instances of the conclusions of the available (conditional) lemmas which promote further computations (e.g. further Fourier-Motzkin elimination steps in the case of the procedure for UPAI) 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 the ordering $\prec$) literal in C.

It is worth pointing out that augmentation critically depends on the shape of the avaialable lemmas and the algorithm implemented by the satisfiability procedure. If a suitable set of lemmas is defined, then augmentation dramatically widens the scope of the 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. 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 $x * y \leq -1$ 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 us 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 [16]. The generated (conditional) lemmas are used as for augmentation.

Combining 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. Augmentation and affinization can be combined in a straighforward way by considering the function symbols occurring in the constraint store C, i.e. the top-most function symbol of a maximal (according to $\prec$) literal in C triggers the invocation of either affinization or augmentation.

Using the available lemmas. We still need to specify how the facts resulting from the lemma speculation activity are used by **RDL**. To understand how this is done, recall that the lemma speculation activity is capable of generating a set of ground lemmas S entailed by the actual context C (cf. $C \mapsto \langle C', S \rangle$). The main obstacle to using the facts in S is that often they are conditional. Hence, their hypotheses must preliminary be relieved before adding their conclusion to the satisfiability procedure. This problem is solved by trying to (constraint contextually) rewrite each hypothesis to *true*. This 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:

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

where $(g_1 \wedge \cdots \wedge g_n) \Longrightarrow (c_1 \wedge \cdots c_m)$ is a ground fact in S (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, e.g., TPTP [22] can be (partially) tackled by **RDL**, we think that **RDL**'s performances should be assessed on proof obligations from real verification efforts. To do this, we are building a corpus of proof obligations extracted from the literature 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 5 reports a selection of the results of our computer experiments. PROBLEM lists the available lemmas[5] (if any) and the formula to be simplified. The symbol $\vdash$ denotes 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 available decision procedure extended with the available facts). The last column records the time (expressed in msec) used by **RDL** to solve a problem.[6]

Table 1. Experimental Results with **RDL**

#	PROBLEM	Time
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

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 lemmas. The rewriter implemented in **RDL** is a key feature to successfully solve problem 2 since ordered rewriting allows to handle non-orientable rewrite rules such as commutativity (i.e. $A * B = B * A$).

RDL solves problem 3 with a satisfiability procedure for UPAI extended with augmentation. In fact, the available lemma is applied once its instantiated condition, namely $x > 0$, is relieved by the satisfiability procedure (it is straightforward to check the inconsistency of the conjunction of $x > 0$ and the literal $x \leq 0$ in the context).

[5] Capitalized letters denote implicitly universally quantified variables.

[6] 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.

RDL solves problems 4, 5, 6,[7] and 7 with a procedure for UPAI and extended with the agumentation 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.
Problem 8 is solved by **RDL** with the combination of the satisfiability procedures for UPAI and UTE.
RDL solves problems 9 and 10 with the procedure for UPAI, 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.
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.

5.1 Comparison with other state-of-the-art systems

As a matter of fact, the online version of STeP fails to solve all of the problems reported in Table 5. However, most of them are successfully solved by the improved version of STeP described in [5]. This version of STeP 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 procedure and an incomplete matching algorithm which is capable of instantiating 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, e.g., $0 < I \Longrightarrow J \leq I * J$ in problem 6).
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.
PVS is capable of handling all the problems in Table 5 but only with a substantial amount of user interaction. This is so, despite the fact that a mechanism to access and instantiate a data base of lemmas (containing a large number of facts about multiplication) is available in the system.[8] Our affinization technique provides the desired degree of automation with a relatively modest implementation effort.
Finally, ACL2 as well as its predecessor NQTHM are capable of solving all the problems involving augmentation with a high degree of automation. However,

[7] Here and in problem 9, $ms^n(C)$ abbreviates $ms^{n-1}(C) * ms(C)$ for $n \geq 2$, and $ms^1(C)$ stands for $ms(C)$.

[8] Recently, to alleviate this situation, a package for manipulating arithmetic expressions (available at http://shemesh.larc.nasa.giv/people/bld/manip.html) have been developed for PVS. We plan to compare **RDL** and the new extension to PVS in the near future.

they fail on the problems requiring affinization desipte the availability of a large data base of facts about multiplication in the distribution of the systems. This is so because multiplication can create a lot of slightly different variants of essentially the "same" formula. This prevents the succesfull instantiation of the available lemmas since augmentation critically depends on the shape of the available lemmas as well as the formula to be proved. Our affinization technique is more robust in this respect since the lemmas about multiplication are speculated by considering the formula currently being proved.

6 Conclusions and Future Work

We have presented **RDL**, a reasoning system especially designed to experiment with the flexible incorporation of satisfibility procedures in a sophisticated form of contextual rewriting and to study lemma speculation mechanisms for widening the scope of applicability of the available procedures. We have also reported on some experimental results which confirm the effectiveness of **RDL** as well as its greater flexibility w.r.t. other state-of-the-art reasoning systems.

We envisage two possible lines of future work. First, we notice that the task of implementing effective satisfiability procedures is non trivial and error-prone. Furthemore, about 40% of **RDL**'s code is devoted to the implementation of the reasoning specialists. In [4], a methodology to build satisfiability procedures based on a superposition calculus is proposed. It would be interesting to study how to incorporate such a methodology in our system to partially automate the task of bulding satisfiability procedures. Second, we have already observed that **RDL** can be used as a powerful reasoning sub-module in generic deduction systems such as, e.g., paramodulation based theorem provers. The problem of building a theory in paramodulation calculi has been widely studied for the case of theories axiomatized by finite sets of equations. It would be interesting to study how to embed **RDL**'s specialized reasoning activity in a paramodulation theorem prover since the theory handled by the available satisfiability procedure needs not be purely equational or even finitely axiomatizable.

References

1. A. Armando and S. Ranise. Constraint Contextual Rewriting. In *Proc. of the 2nd Intl. Workshop on First Order Theorem Proving (FTP'98)*, 1998.
2. 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)*, 2000.
3. 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).
4. A. Armando, S. Ranise, and M. Rusinowitch. Uniform Derivation of Superposition Based Decision Procedures. In *Proc. of the Computer Science in Logic (CSL'01)*. LNCS 2142, Springer-Verlag, 2001.

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. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.
11. D. Kapur, D.R. Musser, and X. Nie. An Overview of the Tecton Proof System. *Theoretical Computer Science*, Vol. 133, October 1994.
12. 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.
13. 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.
14. J.-L. Lassez and M.J. Maher. On Fourier's Algorithm's for Linear Arithmetic Constraints. *J. of Automated Reasoning*, 9:373–379, 1992.
15. Z. Manna and the STeP Group. STeP: The Stanford Temporal Prover. Technical Report CS-TR-94-1518, Stanford University, June 1994.
16. 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.
17. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. Technical Report STAN-CS-78-652, Stanford Computer Science Department, April 1978.
18. 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.
19. L. C Paulson. Proving termination of normalization functions for conditional expressions. *J. of Automated Reasoning*, pages 63–74, 1986.
20. R. S. Shostak. An Efficient Decision Procedure for Arithmetic with Function Symbols. *J. of the ACM*, 2(26):351–360, April 1979.
21. R.E. Shostak. Deciding Combination of Theories. *J. of the ACM*, 31(1):1–12, 1984.
22. G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *12th International Conference on Automated Deduction*, number 814 in LNAI, pages 252–266. Springer-Verlag, 1994.
23. H. Zhang. Contextual Rewriting in Automated Reasoning. *Fundamenta Informaticae*, 24(1/2):107–123, 1995.