The Role of Reformulation in the Automatic Design of Satisfiability Procedures

Jeffrey Van Baalen
Computer Science Department
University of Wyoming
jvb@moran.uwyo.edu

Abstract
Recently there has been increasing interest in the problem of knowledge compilation [Selman & Kautz 91]. This is the problem of identifying tractable techniques for determining the consequences of a knowledge base. We have developed and implemented a technique, called DRAT, that given a theory, i.e., a collection of first-order clauses, can often produce a type of decision procedure for that theory that can be used in the place of a general-purpose first-order theorem prover for determining the many of the consequences of that theory. Hence, DRAT does a type of knowledge compilation. Central to the DRAT technique is a type of reformulation in which a problem's clauses are restated in terms of different nonlogical symbols. The reformulation is isomorphic in the sense that it does not change the semantics of a problem.

INTRODUCTION
Recently there has been increasing interest in the problem of knowledge compilation [Selman & Kautz 91]. This is the problem of identifying tractable techniques for determining the consequences of a knowledge base. Most interesting knowledge bases are written in highly expressive languages for which the general problem of complete inference is intractable (e.g., at least NP-hard, usually undecidable). Even though the general inference problem in such a language is intractable, given a particular knowledge base, it is often possible to identify a tractable inference procedure that is complete for the inferences required in that knowledge base. We have developed and implemented a technique, called DRAT, that given a theory, i.e., a collection of first-order clauses, can often produce a type of decision procedure for that theory. This type of procedure is called a literal satisfiability procedure. Such a satisfiability procedure for a theory $T$ decides whether or not a conjunction of ground literals is satisfiable in $T$. A literal satisfiability procedure for a theory can be used in the place of a general-purpose first-order theorem prover for determining the many of the consequences of that theory. Hence, DRAT does a type of knowledge compilation.

Obviously, we are better off using a satisfiability procedure for determining the consequences of a theory than we are using a general-purpose theorem prover because the satisfiability procedure is guaranteed to halt. However, under what circumstances should we consider such a procedure tractable? A straightforward way to define tractability is polynomial-time worst-case complexity and for some theories DRAT can produce a satisfiability procedure that has this property. For many other theories, the satisfiability procedures produced are exponential in the worst case. Note that DRAT can determine whether a satisfiability procedure it produces has polynomial or exponential worst-case behavior. In either case, the procedures are usually much more efficient than a general theorem prover because the complexity of the theorem prover proving that a fact $F$ follows from a theory $T$ is a function of the sum of the size of $F \cup T$, while the complexity of the satisfiability procedure is a function of the size of $F$.

Even when DRAT cannot produce a literal satisfiability procedure for an entire theory it is often an improvement to use a procedure for a subset of an input theory because such a procedure can be interfaced with a general-purpose theorem prover in such a way that the procedure and the theorem prover work together to determine the consequences of the theory.

In practice, so long as a procedure can be found for a significant subset of the theory, the resulting inference systems are much more efficient than the theorem prover alone because many of the inferences that the theorem prover would have to do are done more efficiently by the satisfiability procedure.

Let $\Psi$ be the set of axioms of a problem and let $S$ be the satisfiability procedure that DRAT designs for $\Psi$, some subset of $\Psi$. The theorem prover restricts its manipulation of the statements in $\Psi'$, using $S$ instead whenever possible. This paper presents a formalization of DRAT and proves that it is complete, i.e., that for any first-order statement $\phi$, if $\Psi \models \phi$, $S$ combined with the theorem prover will prove $\phi$. We show that DRAT's reformulation greatly increases its effectiveness and that a solution to a reformulated version of a problem is
guaranteed to be a solution to the original problem.

We present only a brief description of the DRAT algorithm here. A detailed description of an implementation can be found in [VanBaalen89] or [VanBaalen92].

DRAT was inspired by human problem solving performance on analytical tasks of the type found on graduate level standardized admissions tests. An example problem is given in Figure 1.

Given: M, N, O, P, Q, R, and S are all members of the same family. N is married to P. S is a grandchild of Q. O is a niece of M. The mother of S is the only sister of M. R is Q's only child. M has no brothers. N is a grandfather of O.

Query: Who are the siblings of S?

Figure 1: The FAMILIES Analytical Reasoning Problem

We analyzed human problem-solving behavior on a number of these problems and found the prevalent use of diagrams to assist in problem solving. Figure 2 illustrates the typical diagrams people use to solve the problem in Figure 1.

```
"R is the only child of Q"  "S is a grandchild of Q"
(Divided rectangles represent couples; circles represent sets of children of the same couple: full circles are closed sets, broken circles are sets all of whose members may not be known; the directed arc represents the "children-of" function between couples and their sets of children.)
```

Figure 2: Two statements in a representation commonly used by people.

These diagrams were found to contain a common set of structures (across different people and different problems). The arcs in Figure 2 are an example of such a structure. They represent the 1-1 function between a married couple and their set of children. Each common structure was also found to have a standard set of procedures for manipulating it. For example, one procedure associated with the arcs in Figure 2 ensures that they behave like a 1-1 function. It reads roughly as, "If two objects are equal and they appear at the same end of two separate 1-1 function arcs with the same function symbol, the arcs and the objects at their other end can be composed." This procedure is among those used to compose the structures in Figure 2 to yield the diagram in Figure 3.

People use these diagrams to test the satisfiability of a particular collection of facts by creating the structures representing each fact and then composing them. The conjunction is satisfiable just in case no contradiction is signalled in the composition process.

DRAT has a library of procedures called schemes. These schemes model people's diagrammatic structures and their manipulations. Schemes were found to have a number of important properties which are described in this paper. Perhaps the most important of these properties is that each scheme turns out to be a satisfiability procedure. Another important property of schemes is that they can be used as building blocks to construct "larger" satisfiability procedures. DRAT uses this property to construct satisfiability procedures for input problems.

The implementation of DRAT includes the schemes found in analyzing the diagrams that people used on thirty analytical tasks. It has been tested on twelve of these problems stated in a sorted first-order logic. The problems vary in size from thirty to sixty sorted first-order statements. The performance of the theorem prover/satisfiability procedure combinations that DRAT produces for these problems was at least two orders of magnitude better than the performance of the theorem prover alone. For example, our general theorem prover took 988,442 resolutions—three hours and five minutes—to solve the problem shown in Figure 1. The satisfiability procedure that DRAT produced was able to solve the problem entirely without the theorem prover and did so in less than three seconds.

PRELIMINARIES

Each scheme is a tractable literal satisfiability procedure for a theory.

Definition 1 A theory is a set of statements in first-order predicate calculus with equality.

Definition 2 A literal satisfiability procedure for a theory \( T \) is a procedure that decides for any conjunction of ground literals \( \Sigma \) whether or not \( \Sigma \cup T \) is satisfiable.

Each scheme is tractable in the sense that, given any \( \Sigma \) containing \( n \) literals, the scheme for a theory \( T \) decides the satisfiability of \( \Sigma \cup T \) in time polynomial in \( n \).

Given a particular \( \Sigma \), in addition to determining literal satisfiability in some theories, each scheme computes \( \{ u = v \mid u, v \in C \land \Sigma \} \) where \( C \) is the set of constant symbols appearing in \( \Sigma \). As detailed in section, these equalities are communicated between schemes in a way that allows the combination
of schemes to determine satisfiability for the union of their theories.

One important result of this research is the particular library of schemes we have developed from the observation of human problem solving of analytical tasks. However, in the formal characterization that follows, we abstract away from the detail of the current scheme library, identifying the properties of schemes required for the completeness of DRAT.

This paper first takes a simplified view of what DRAT will accept as an input problem and also assumes that DRAT is only successful if it can produce a satisfiability procedure for an entire problem. In this setting, we prove that a combination of schemes is a satisfiability procedure for the union of the theories of the individual schemes. In section , the above restrictions are relaxed and it is shown how, in the more general setting, the procedures produced by DRAT are interfaced with a theorem prover.

DRAT requires that the formulas of schemes and the formulas of an input problem be converted to clauses, i.e., disjunctions of first-order literals. The remainder of the paper assumes that this has been done. However, the presentation will often use more intuitive forms for statements, when the conversion to clause form is straightforward.

The restricted definition of a problem taken first is:

**Definition 3 A problem is a triple \( < \Sigma, T_C, \Phi > \), where \( \Sigma \) and \( \Phi \) are sets of ground literals and \( T_C \) is a set of clauses each of which contains at least one variable. Such a triple is interpreted as a question about whether or not for all the ground literals \( \phi \in \Phi \), \( \Sigma \cup T_C \models \phi \).

Here is an example problem:

\[
\Sigma_1 = \{ \text{grandfather}(O, N), \text{married}(N, P), \text{grandchild}(S, Q), \text{niece}(O, M), M \neq N, N \neq O, \ldots \}
\]

\[
T_{C_1} = \{ \text{mother}(S, z) \Rightarrow \text{sister}(M, z), (\text{sisster}(M, z) \land \text{sister}(M, y)) \Rightarrow z = y, \text{child}(Q, z) \Rightarrow z = R, \ldots \}
\]

\[
\Phi_1 = \{ \text{sibling}(O, S), \text{child}(N, M) \}
\]

In addition to those axioms shown, \( \Sigma_1 \) also contains disequalities between all of the individual constants mentioned. \( T_{C_1} \) also contains definitions of concepts such as a grandchild and formulas defining general properties of the family relation domain such as symmetry of married.

Given a problem \( < \Sigma, T_C, \Phi > \), DRAT's objective is to design a literal satisfiability procedure for \( T_C \). This procedure is used to solve the problem for the particular \( \Sigma \) and \( \Phi \). To determine whether for some \( \phi \in \Phi \), \( \Sigma \cup T_C \models \phi \), the satisfiability procedure for \( T_C \) is used to decide whether or not \( \Sigma \cup T_C \cup \neg \phi \) is unsatisfiable. For example, DRAT tries to design a satisfiability procedure for \( T_{C_1} \). If successful, the procedure is used to decide whether "O" is a sibling of "S" and "M" is a child of "N" follow from \( \Sigma_1 \cup T_{C_1} \) by determining the satisfiability of \( \Sigma_1 \cup T_{C_1} \cup \neg \text{sibling}(O, S) \) and of \( \Sigma_1 \cup T_{C_1} \cup \neg \text{child}(N, M) \).

Obviously, we are better off using a satisfiability procedure for \( T_C \) to solve a problem \( < \Sigma, T_C, \Phi > \) than using a general theorem prover because the satisfiability procedure is guaranteed to halt. Perhaps less obvious is the fact that these procedures are usually much more efficient than a general theorem prover. The intuition behind this is that the complexity of the theorem prover solving the problem is a function of the size of the entire problem, while the complexity of the satisfiability procedure is a function of the size of \( \Sigma \cup \Phi \). As pointed out in section , this intuition is substantiated by the performance of the procedures that DRAT has designed.

**THE DRAT TECHNIQUE**

We will call the relation, function and individual constant symbols in a theory the nonlogical symbols of that theory. The nonlogical symbols of each scheme's theory are treated as parameters to be instantiated with the nonlogical symbols of \( T_C \). For example, the scheme \( T_{\text{symmetric}} \) whose theory is \( \{ R(x, y) \Rightarrow R(y, x) \} \) is parameterized by \( R \).

DRAT tries to find a set of scheme instances that can be combined to give a literal satisfiability procedure for \( T_C \). Consider a set of scheme instances. Call the union of the theories of each scheme instance \( T_I \). DRAT has succeeded in finding a satisfiability procedure when it finds a \( T_I \) that is logically equivalent to \( T_C \). The following is an abstract description of this process:

**instances \( \rightarrow \emptyset \)**
**\( T_I \rightarrow \emptyset \)**
**\( T_C \rightarrow T_C \)**
UNTIL empty(\( T_C \)) DO
**instance \rightarrow \text{choose-instance}(T_C)**
**IF null\( (\text{instance}) \) THEN \text{EXIT-WITH failure)**
**instances \rightarrow \text{union}(\text{instance}, \text{instances})\)**
**\( T_I \rightarrow \text{union}(\text{theory}(\text{instance}), T_I)\)**
**FOR EACH \( \phi \in T_C \)**
**WHEN \( T_I \models \phi \) DO \( T_C \rightarrow T_C \cup \neg \phi \)**
END FOR
END UNTIL

A set of scheme instances is built up incrementally and, simultaneously, the set of clauses in \( T_C \) is pared down. Each time \text{choose-instance} is invoked, it inspects \( T_C \) and chooses a scheme instance whose theory is entailed by \( T_C \). After the theory of \text{instance} is added to \( T_I \), DRAT removes clauses from \( T_C \) that are entailed by \( T_I \).

DRAT uses the following procedure for computing satisfiability in \( T_I \) to determine the \( \phi \in T_C \) that follow from \( T_I \). For each clause \( \phi \), it creates \( \phi' \) by substituting a new individual constant for each unique variable in \( \phi \). If the satisfiability procedure for \( T_I \) reports that \( \neg \phi' \cup T_I \) is unsatisfiable, \( T_I \models \phi \).

If the algorithm is exited with \( T_C \) empty, DRAT has succeeded in finding a \( T_I \) that is equivalent to \( T_C \). To

---

163
see this, note that T_C ∪ T_I = T_C is an invariant of the
procedure. Adding theory(instance) to T_I does not violate
the condition because T_C = theory(instance). Removing
from T_C clauses φ such that T_I ̸≡ φ also does
not violate the condition.

If the algorithm is exited because choose-instance
returns nil, it has failed to find a T_I that is equivalent
to T_C.

Note that this algorithm is nondeterministic be-
cause, in general, on a call to choose-instance, there
are several instances from which to choose. The DRAT
implementation searches for an appropriate collection
of scheme instances. This search is reduced consider-
ably by the fact that scheme instances in T_I may not
share nonlogical symbols. As discussed in section, this
restriction is required to allow schemes to be
combined by the method described below. More detail on how
the DRAT implementation controls this search can be
found in [VanBaalen92].

A PROCEDURE FOR COMBINING
SCHEMES

Since T_I is the theory of a set of scheme instances, so
long as these instances do not share nonlogical symbol-
s, DRAT has a satisfiability procedure for T_I. This
procedure is the combination of schemes used to create T_I.
DRAT's combination technique is the same technique as
reported by Nelson & Oppen in [Nelson&Oppen79] and
a more detailed description than what follows can be
found there.

Let L(T) be the set of nonlogical symbols appearing
in the clauses of T. We will often refer to L(T) as the
language of T. Consider two scheme instances, T_1 and
T_2, where L(T_1) is disjoint from L(T_2), and consider a
conjunction of literals Σ in L(T_1 ∪ T_2). The procedure
for deciding the satisfiability of Σ ̸∈ T_1 ∪ T_2
begins by splitting Σ into two conjunctions of literals: Σ_1, with
literals in L(T_1) and Σ_2, with literals in L(T_2) such that
the conjunction of literals in Σ_1 and Σ_2 is satisfiable
just in case Σ is.

When a literal in Σ contains nonlogical symbols from
L(T_1 ∪ T_2), remove each subterm whose function
symbol is not in the language of the head symbol of the
term. A subterm is removed by substituting a new
constant symbol for that subterm in the literal and
conjoining an equality between the term and the new
symbol with the proper Σ_i. For example, suppose R
is in L(T_1). f is in L(T_2) and Σ contains the literal
R(f(a)). The embedded terms is in the wrong language,
so it is removed. This is done by substituting a new
constant, say b, for f(a) in R(f(a)) to obtain R(b) and
conjoining b = f(a) with Σ_2.

For each literal in Σ, this technique is applied repeated-
ly to the right most function symbol in the wrong
language until the literal no longer contains symbols
in the wrong language. Then the literal is conjoined
with the appropriate Σ_i. For instance, R(b) from the
example above contains no symbols in the wrong lan-
guage so it is conjoined with Σ_1.

Next the scheme for T_I is used to determine the satis-
fiability of Σ_1 ∪ T_1. Recall that in so doing, this scheme
also computes the set of equalities between constants
in Σ_1 that follow from Σ_1 ∪ T_1. Call this set E_1. The
scheme for T_2 is used to determine the satisfiability of
Σ_2 ∪ T_2 ∪ E_1. If it is satisfiable, E_2, the set of equalities
that follow from Σ_2 ∪ T_2 ∪ E_1, is propagated back to
T_1, i.e., T_I is used to compute Σ_1 ∪ T_1 ∪ E_2.

This propagation of equalities continues until one of
the schemes reports "unsatisfiable" or until no new eq-
ualities are computed. Note that since there are at
most n - 1 nonredundant equalities between n con-
stant symbols, this process will terminate. Unless the
scheme for T_I or T_2 reports "unsatisfiable," the procedure
for the combination returns "satisfiable."

A complication to this equality propagation pro-
cedure is that given a set of ground literals, many
tractable schemes imply disjunctions of equalities be-
 tween constants without implying any of the dis-
juncts alone, a property called nonconvexity in
[Nelson&Oppen79]. An example of a convex scheme
is one that determines satisfiability for the theory of
equality with uninterpreted function symbols. An
example of a nonconvex scheme is one for the theory
of sets. To see this, note that \{a, b\} = \{c, d\} implies
a = cV a = d, but does not imply either equality alone.

A scheme associated with a nonconvex theory must
calculate disjunctions of equalities between constants
that follow from a given conjunction of ground liter-
als. The equality propagation procedure is extend-
ed to handle such schemes by case splitting when a
nonconvex scheme produces a disjunction. When one
of the component schemes produces the disjunction
c_1 \equiv d_1 \land \cdots \land c_n \equiv d_n, the combined satisfiability
procedure is applied recursively to the conjunc-
tions Σ_1 ∪ Σ_2 ∪ \{c_1 = d_1\}, \ldots, Σ_1 ∪ Σ_2 ∪ \{c_n = d_n\}. If any
of these is satisfiable, "satisfiable" is returned, otherwise
"unsatisfiable" is returned.

As a simple example of this procedure, consider two
schemes: E for the theory of equality with uninter-
preted function symbols and S for the theory of finite sets.
Now consider whether

Σ = \{f(a) = \{b, g\} \land f(c) = \{d, e\} \land a = c \land
\}
g \neq d \land g \neq d \land \land b \neq d \land b \neq e

is satisfiable. First Σ is split into

Σ_1 = \{a = c \land g \neq d \land g \neq e \land b \neq d \land b \neq e \land
\}
f(a) = c \land f(c) = c_2
\}
Σ_2 = \{b, g\} \land c_2 = \{d, e\}

E is run on Σ_1 and determines that c_1 = c_2. S is
run on Σ_2 ∪ \{c_1 = c_2\} which produces the disjunction
b = d \lor d = e. The procedure is now invoked recursive-
ly for Σ_1 ∪ Σ_2 ∪ \{b = d\} and Σ_1 ∪ Σ_2 ∪ \{b = e\}. In both
calls, Σ_2 produces the disjunction g = d \lor g = e which
is unsatisfiable. Therefore, both calls return "unsatis-
fiable," hence Σ ̸∈ E ∪ S is unsatisfiable.

We place one additional requirement on schemes

164
to make the equality propagation procedure practical. Schemes must be incremental. This means that a scheme must be able to save its "state" when a conjunction of literals is satisfiable and it must be able to use the saved state to determine the satisfiability of larger conjunctions at incremental cost.

**REFORMULATION**

The DRAT technique as described in section is severely limited by the way in which a problem is stated. Often, it is much more successful with an equivalent formulation of the problem stated in terms of a different collection of nonlogical symbols. For instance, recall the problem about family relations given in section. It was stated in terms of the binary relation child. It turns out that, given the current scheme library, the DRAT implementation is much more successful when the problem is stated in terms of parents, a function from an individual to his or her set of parents. One reason this formulation is better is that the library contains a scheme for a theory of fixed sized sets. DRAT discovers an instance of this scheme that allows it to remove several general clauses from the problem including one that limits the size of parent sets to two.

In an effort to circumvent this sensitivity to a problem's formulation, DRAT is able to reformulate a problem in terms of new nonlogical symbols without changing the "meaning" of the problem. Choose instance is often able to find scheme instances in reformulated problems where it was unable to do so in the initial formulations. DRAT's reformulation technique is modeled after the reformulation that people do in solving analytical tasks. For an example of this refer again to the problem and diagrams given in section. In the diagrams appear concepts such as "married couples" and "sets of children of the same couple." These concepts are not present in the initial problem formulation — the problem has been reformulated.

DRAT does a particular kind of reformulation called isomorphic reformulation in [Korf80]. We formalize isomorphic reformulation as a relation between theories.

**Definition 4** A reformulation map $R^*_{\mathcal{L}_1, \mathcal{L}_2}$ between two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is a function from clauses in $\mathcal{L}_1$ to sets of clauses in $\mathcal{L}_2$.

**Definition 5** A theory $T_2$ is an isomorphic reformulation of a theory $T_1$ just in case there exists a reformulation map $R^*_{\mathcal{L}_1, \mathcal{L}_2}$ such that $T_1 \models \phi \iff T_2 \models R^*_{\mathcal{L}_1, \mathcal{L}_2}(\phi)$, for every clause $\phi$ in $\mathcal{L}(T_1)$.

If $T_2$ is an isomorphic reformulation of $T_1$, any question we have about what clauses are entailed by $T_1$ can be answered by theorem proving in $T_2$. Given the question, "does $T_1 \models \phi"? we use $R^*$ to translate $\phi$ into $\mathcal{L}(T_2)$ and then attempt to prove that $T_2 \models R^*(\phi)$.

As a simple example of isomorphic reformulation, consider the following two theories:

$$T_1 = \{ R(x, z), \\ R(x, y) \Rightarrow R(y, z), \\ R(x, y) \land R(y, z) \Rightarrow R(x, z) \}$$

$$T_2 = \{ \begin{align*} x \in R\text{-class}(y) & \Rightarrow y \in R\text{-class}(x) \\ x \in R\text{-class}(y) & \land y \in R\text{-class}(z) \Rightarrow z \in R\text{-class}(x) \end{align*} \}$$

$T_2$ is an isomorphic reformulation of $T_1$. To show this, we exhibit an appropriate $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$. First, we introduce the function $\gamma$ with $\gamma(R(x, y)) = x \in R\text{-class}(y)$ and $\gamma(-R(x, y)) = x \notin R\text{-class}(y)$.

Given the literals $\phi_1, \ldots, \phi_n$, $n \geq 1$:

$$R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(\phi_1 \lor \cdots \lor \phi_n) = \{ \gamma(\phi_1) \lor \cdots \lor \gamma(\phi_n) \}.$$ 

Now $T_2 \equiv R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(T_1)$, using the obvious extension of $R^*$ to sets of clauses. Therefore, $T_1 \models \phi \iff T_2 \models R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(\phi)$. To see this, note that we can take any resolution proof of $T_1 \vdash \phi$ and uniformly apply $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$ to the clauses in each step of the proof to obtain a proof of $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(T_1) \vdash R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(\phi)$. We can also define $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$ similarly to $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$ and use it to transform any proof $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(T_1) \vdash R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}(\phi)$ into a proof of $T_1 \vdash \phi$.

**ADDING REFORMULATION TO DRAT**

One strategy for finding a satisfiability procedure for a theory $T_1$ is to identify a theory $T_2$ with the following properties: (1) a satisfiability procedure is known for $T_2$, (2) we can find a reformulation map $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$, demonstrating that $T_2$ is an isomorphic reformulation of $T_1$ and (3) $R^*_{\mathcal{L}(T_1), \mathcal{L}(T_2)}$ is a computable function.

The actual DRAT technique is an extension of the algorithm discussed in section to apply the above strategy. This extension enables DRAT to generate theories that are isomorphic reformulations of $T_2$ while searching for a set of scheme instances that is a satisfiability procedure for $T_2$. DRAT has a library of reformulation rules, each of which is a reformulation map. These rules are applied to an input theory $T_2$ to construct theories that are isomorphic reformulations of $T_2$. The extended algorithm searches for scheme instances in these isomorphic reformulations as well as in the original $T_2$.

Roughly, each reformulation rule is viewed as an axiom schema that can be instantiated with nonlogical symbols and used as a rewrite rule to reformulate a theory. To understand this view, consider the following axiom schema in which $R$ is a parameter:
This states that for any binary relation, there is a projection function \( F_R \) that is a mapping from individuals to sets of individuals such that \( F_R(y) = \{ x \mid R(x, y) \} \).

DRAT can apply the above reformulation rule to binary relations in \( T_C \). When the rule is applied to \( R \) in \( T_C \), the new function symbol \( F_R \) is introduced and \( T_C \) is reformulated in terms of \( F_R \). For instance, if this rule is applied to \( \text{child} \) in the family relationship problem given earlier, it will introduce a function that we will call \( \text{parents} \), from an individual to his or her set of parents. DRAT uses the formula introducing \( \text{parents} \), i.e., \( \text{child}(x, y) \iff x \in \text{parents}(y) \), to reformulate the problem, rewriting all occurrences of \( \text{child}(x, y) \) to \( x \in \text{parents}(y) \).

This example reformulation rule can be applied to any binary relation in any theory. More generally, DRAT's reformulation rules are conditional on properties of nonlogical symbols in a theory. A property of a nonlogical symbol is simply a first-order statement mentioning that symbol. Before giving the general form of reformulation rules, we introduce the function \( \text{rf-symbols}(T) \), the set of relation and function symbols of \( T \). The \( \text{rf-symbols}(T) \) does not contain the symbols = or \( \in \), even if they are mentioned in \( T \). These are treated as special (logical) symbols in the reformulation process.

The general form of reformulation rules is given in the following definition.

**Definition 6** A triple \(< P, Q, \Theta \iff \Psi >\) is a reformulation rule when it meets the following restrictions: (1) \( P \) and \( Q \) are conjunctions of clauses (both of which may be empty). (2) \( \Theta \) and \( \Psi \) are conjunctions of literals. (3) \( \text{rf-symbols}(P) \subseteq \text{rf-symbols}(\Theta) \) and \( \text{rf-symbols}(Q) \subseteq \text{rf-symbols}(\Psi) \). (4) \( \text{rf-symbols}(\Theta) \) is disjoint from \( \text{rf-symbols}(\Psi) \). (5) \( \Theta \) and \( \Psi \) have the same variables.

Rules are symmetric in the sense that their biconditionals can be used to introduce new symbols in "either direction." When the parameters in \( \Theta \) are instantiated with symbols in a theory \( T \), the rule is used to reformulate \( T \) in terms of the new symbols in \( \Psi \). The conjunction of clauses \( P \) is the condition that must be true of a theory for the reformulation rule to be used to rewrite \( \Theta \) as \( \Psi \). When the parameters in \( \Psi \) are instantiated with the symbols in \( T \), the rule is used to reformulate \( T \) in terms of the new symbols in \( \Theta \). In this case, \( Q \) is the condition that must be true for the rule to be used.

Here is an example of a conditional reformulation rule:
\[
< \{ z \in F(y) \Rightarrow F(y) = \{ x \} \}, \]
\[
[ x \in F(y) \iff x \neq \bot \land x = F'(y) ] >. \]

This rule can be applied to any theory \( T \) containing a function \( F \) whose range elements are sets of size one, i.e., \( P = \{ x \in F(y) \Rightarrow F(y) = \{ x \} \} \). When applied, the rule reformulates \( T \) in terms of a function \( F' \) such that \( F'(y) = \{ x \} \) just in case \( x \in F(y) \). \( Q \) is empty in this rule because the rule can always be applied in the other direction.

The following is an abstract description of the DRAT algorithm extended to do reformulation:

```
i.e., \( P = \{ x \in F(y) \Rightarrow F(y) = \{ x \} \} \). When applied, the rule reformulates \( T \) in terms of a function \( F' \) such that \( F'(y) = \{ x \} \) just in case \( x \in F(y) \). \( Q \) is empty in this rule because the rule can always be applied in the other direction.

The following is an abstract description of the DRAT algorithm extended to do reformulation:

```
```
Again we have suppressed the issues of search by giving a nondeterministic procedure. The search conducted by the extended algorithm is over a much larger space than the search conducted by the simple algorithm described in section 2. The DRAT implementation with reformulation must compare alternative problem formulations. Fortunately, we have found some effective heuristics for controlling the search. See [VanBaalen89] or [VanBaalen91] for details.

The procedure instantiate instantiates a rule with respect to the nonlogical symbols in symbols to produce an instantiated-rule. \( R \) is the reformulation procedure. We describe this procedure for the case where a rule of the form

\[
\langle P, Q, \theta_1 \land \cdots \land \theta_n \Rightarrow \psi >
\]

is used to rewrite occurrences of \( \theta_1 \land \cdots \land \theta_n \), the from conjunct, to occurrences of \( \psi \), the to conjunct. The procedure for applying the rule in the other direction is obtained by reversing the biconditional and replacing references to \( P \) by references to \( Q \).

Each set of unit clauses in \( T_c \), of the form \( \{(\theta_1)\sigma, \ldots, (\theta_n)\sigma\} \), where \( \sigma \) is a substitution for the variables in \( \theta_1 \), is rewritten as the set of unit clauses \( (\psi)\sigma \). Each clause containing the literals \( (\neg \theta_1)\sigma, \ldots, (\neg \theta_n)\sigma \) is rewritten to contain \( (\neg \psi)\sigma \). After all possible occurrences are rewritten, the clauses in \( Q \) are added to the rewritten theory.

We call a rewriting produced by \( R \) complete when it removes all of the nonlogical symbols appearing in the from conjunct. \( R \) may or may not produce a complete rewriting. For example, given a right hand side of the form \( R(f(x)) \), rewriting will only be complete when \( R \) and \( f \) appear in a theory only in patterns of this form. If the rewriting process is not complete, \( R \) adds the instantiated \( \Theta \Rightarrow \psi \) to the rewritten theory.

As an example of applying \( R \), consider again the rule

\[
\langle [x \in F(y) \Rightarrow F(y) = \{x\}], [x \in F'(y) \Rightarrow z \neq \bot \land z = F'(y)] >.
\]

As noted, the condition \( P \) must follow from a theory to reformulate \( F \) as \( F' \) in that theory. Since the condition \( Q \) is empty, there are no clauses to add to the resulting theory. If the rewriting is not complete, \( [z \neq \bot \land z = F'(y) \Rightarrow z \in F(y)] \) is added to the rewritten theory. Since there is no condition \( Q \), this rule can always be used, in the other direction, to reformulate \( F' \) as \( F \). In this case, \( P \) is added to the rewritten theory. Again, the biconditional may need to be added to the rewritten theory.

To ensure that the extended DRAT algorithm generates only isomorphic reformulations, each reformulation rule must be shown to generate only isomorphic reformulations. To guarantee this, we require that, when instantiated, each reformulation rule be an extending definition.

Definition 7 A reformulation rule \( \langle P, Q, \Theta \Rightarrow \psi > \) is an extending definition if for all theories \( T \) the following conditions hold:

1. Whenever the \( rf\text{-}symbols(\Theta) \subseteq rf\text{-}symbols(T) \).

2. Whenever the \( rf\text{-}symbols(\psi) \subseteq rf\text{-}symbols(T) \).

\( rf\text{-}symbols(\psi) \) is disjoint from \( rf\text{-}symbols(T) \) and \( T \models P \), then every model of \( T \) can be expanded to a model of \( T \cup \{ \Theta \Rightarrow \psi \} \).

Section shows that for any reformulation rule rule, \( \lambda(t) \langle rule, t \rangle \) is a computable function and so long as rule is an extending definition, that whenever a theory \( T \) entails the appropriate condition of rule, \( R(\text{rule}, \Theta \rangle \) is an isomorphic reformulation of \( T \).

The \( R^* \) produced by DRAT on the problem \( \langle \Sigma, T_C, \Phi > \) is the composition of reformulation maps used by the algorithm to reformulate \( T_C \). Since each reformulation map generates an isomorphic reformulation, \( R^*(T_C) \) is an isomorphic reformulation of \( T_C \). Since each step is computable, \( R^* \) is a computable function.

Finally we point out that, since \( \psi \) and \( \Theta \) in the reformulation rule \( \langle P, Q, \Theta \Rightarrow \psi > \) are required to have the same variables, \( R^*(\Sigma) \) and \( R^*(\Phi) \) will always be ground. However, even though \( \Sigma \) and \( \Phi \) are conjunctions of ground literals, \( R^*(\Sigma) \) and \( R^*(\Phi) \) may not be. To see this, suppose that \( \Sigma \) contains the literal \( \neg \phi \) and \( R^*(\phi) \) is a conjunction. Then \( \neg R^*(\phi) \) will be a disjunction.

Section shows that when DRAT uses reformulation in designing a satisfiability procedure for a problem \( \langle \Sigma, T_C, \Phi, \Theta \rangle \) and \( R^*(\Sigma) \) is a conjunction of literals, the problem can be solved by solving \( \langle R^*(\Sigma), R^*(T_C), R^*(\Phi) > \). The fact that a satisfiability procedure for a reformulation of a problem requires \( R^*(\Sigma) \) to be a conjunction of literals is not a significant difficulty in the more general setting discussed in section in which satisfiability procedures are used in conjunction with a theorem prover.

AN EXAMPLE

In practice, we have found that adding reformulation to DRAT increases its effectiveness considerably. We illustrate this with a relatively simple example excerpted from the DRAT implementation design of a satisfiability procedure for the example problem given in section 3. We illustrate the implementation's behavior on the set \( T \) of clauses:

\[
\neg \text{married}(x, z), \text{married}(x, y) \Rightarrow \text{married}(y, x), \text{married}(y, z) \Rightarrow \neg \text{married}(x, z), \text{married}(y, z) \land \neg \text{married}(x, z) \Rightarrow y = z.
\]

There are three schemes in DRAT's library that are relevant to the example. The scheme \( F \) for the theory of partial 1-1 functions with parameters \( F \) and \( F' \), which are inverse functions, and theory \( (F') = \{x \in F'(y) \land x \neq \bot \Rightarrow y = F'(z) \land y \neq \bot\} \). The scheme \( S_2 \) for the theory of sets of size two with \( S \) as a parameter and theory \( (S_2) = \{z_1 \in S \land z_2 \in S \land z_1 \neq z_2 \Rightarrow S = \}

\]
The relevant reformulation rules are:
\[ r_1 := \langle R(x, y) \leftrightarrow y \in R(x) \rangle \]
\[ r_2 := \langle x \in F(y) \Rightarrow F(y) = \{ x \} \rangle \]
\[ r_3 := \langle (z \neq z \wedge y \neq z \Rightarrow (z = F(y) \Rightarrow y = F(x)) \rangle \]

As is typical in the implementation, these rules are normally used only in one direction. As noted in section, \( r_1 \) reformulates a binary relation in a theory as a function \( F_R \) onto sets: \( F_R(x) = \{ y \mid R(x, y) \} \). Also as noted in section, when applied to a theory containing a function \( F \) whose range elements are sets of size one, \( r_2 \) introduces a function \( F' \) such that \( F'(y) = x \) just in case \( x \in F(y) \). The rule \( r_3 \) reformulates an \( F \) that is its own inverse as a function \( F' \), mapping an individual into sets of size two such that \( F'(x) = (x, \{ x \}) \).

Given the schemes above, DRAT is unable to design a satisfiability procedure for \( T \) without reformulation. In an effort to design a satisfiability procedure for all of \( T \), the DRAT implementation repeatedly reformulates the problem, finally producing a formulation in terms of a function that we will call couple, mapping an individual to the married couple of which he or she is a member.

**DRAT uses rule \( r_1 \)** to reformulate \( T \) in terms of a function that we will call spouses, a mapping from an individual to the set of his or her spouses. \( R(r_1, T) \) is
\[ \begin{align*}
\forall x \in \text{spouses}(y) & \Rightarrow y \in \text{spouses}(x) \\
\forall x \in \text{spouses}(y) \wedge y \in \text{spouses}(z) & \Rightarrow z \notin \text{spouses}(x) \\
\forall y \in \text{spouses}(x) \wedge z \notin \text{spouses}(y) & \Rightarrow y = z
\end{align*} \]

**DRAT uses rule \( r_2 \)** to reformulate \( R(r_1, T) \) in terms of a partial function that we will call spouse, a mapping from an individual to his or her spouse.
\[ R(r_2, R(r_1, T)) \]
\[ \begin{align*}
x \neq \text{spouse}(x) & \lor x = z \\
x = \text{spouse}(y) \wedge x \neq y & \Rightarrow y = \text{spouse}(z) \wedge y \neq z \\
x = \text{spouse}(y) \wedge x \neq y & \Rightarrow y = \text{spouse}(x) \wedge y \neq z \\
& \Rightarrow x \neq \text{spouse}(z) \lor z = y \\
y = \text{spouse}(x) \wedge y \neq z & \Rightarrow z = \text{spouse}(x) \wedge z \neq y \\
& \Rightarrow y = z
\end{align*} \]

**DRAT uses rule \( r_3 \)** to reformulate the above theory in terms of the function couple. The result is
\[ \begin{align*}
couple(x) & \neq \{ x, z \} \lor z = x \\
couple(x) = \{ z, y \} \wedge x \neq y & \Rightarrow \\
couple(y) = \{ x, z \} \wedge y \neq z \\
couple(y) = \{ y, z \} \wedge y \neq z & \Rightarrow \\
couple(z) \neq \{ x, z \} \wedge z = x \\
couple(y) = \{ z, y \} \wedge y \neq z \\
couple(z) = \{ z, x \} \wedge z \neq y & \Rightarrow \end{align*} \]

All of the clauses in this set follow from the combination of \( S_2 \) and an instance of \( E \) containing the uninterpreted function symbol couple. Thus, through the use of reformulation, DRAT succeeds in designing a satisfiability procedure for the theory \( T \). Without reformulation it is unable to design a procedure for any subset of \( T \).

**STEPS TOWARDS THE COMPLETENESS OF DRAT**

This section proves two results towards the completeness of DRAT. First, we show that DRAT designs satisfiability procedures. If DRAT successfully designs a procedure for some set of axioms \( T_C \), then that procedure can be used to decide the problem \( < E, T_c, \Sigma, \Phi > \) for any conjunctions of ground literals \( \Sigma \) and \( \Phi \). Second, we consider the addition of reformulation to DRAT and show that a satisfiability procedure for \( R^*(\Sigma) \) can be used as a satisfiability procedure for \( T_C \) so long as \( R^*(\Sigma) \) is a conjunction of literals. These results are necessary preliminaries for the proof of completeness in section.

**DRAT DESIGNS SATISFIABILITY PROCEDURES**

Before proceeding to prove that DRAT designs satisfiability procedures, we recall properties of schemes presented thus far and discuss some additional required properties.

Recall that a scheme for a theory \( T \) is a procedure that decides the satisfiability of \( \Sigma \cup T \), where \( \Sigma \) is a conjunction of ground literals. Given a particular \( \Sigma \), each scheme also computes the set of equalities between constants in \( \Sigma \) that follow from \( \Sigma \cup T \). If \( T \) is nonconvex, its scheme also computes disjunctions of equalities between constants in \( \Sigma \) that follow from \( \Sigma \cup T \).

We call a first-order theory whose formulas contain no existential quantifiers a quantifier-free theory. An additional requirement on schemes is that their theories be quantifier-free. As a practical matter, this is not a serious restriction beyond restricting schemes to be tractable. See [Oppen80] for further discussion of this point.

The theories of schemes are also required to have infinite models. The equality propagation technique may not work if a theory has only finite models because, given a set of constant symbols larger than the set of individuals in the model's domain, such a theory implies the disjunction of equalities between those constant symbols. Theories with infinite models do not imply disjunctions of equalities between variables. Therefore, given a theory \( T \) with infinite models, such disjunctions can only follow from \( T \cup \Sigma, \Sigma \) for some \( \Sigma \) whose satisfiability is being decided. Any disjunctions of equalities between constants that follow must involve only constants mentioned in \( \Sigma \). This restriction to theories with infinite models does not appear to be significant. To date, we have not found any schemes that we could not include because they violated this restriction.
The theorem proved below is similar to the theorem in [Nelson & Oppen 79]. It differs in the addition of the requirement that each scheme's theory have infinite models. The theorem appearing in [Nelson & Oppen 79] is incorrectly stated. The reason a different proof is included here is that the proof given in [Nelson & Oppen 79] is incorrect. We also include our proof because the technique is much more direct and serves as a foundation for research in progress to extend our results.

**Theorem 1** Let $T_1$ and $T_2$ be theories with no common nonlogical symbols. If there are schemes for $T_1$ and $T_2$, there is a scheme for $T_1 \cup T_2$.

**Proof:** We prove that the procedure described in section 3 for combining two schemes is a scheme for $T_1 \cup T_2$. If the scheme for $T_1$ or $T_2$ reports "unsatisfiability," clearly $\Sigma_1 \cup \Sigma_2 \cup T_1 \cup T_2$ is unsatisfiable and, since $\Sigma_1 \cup \Sigma_2$ and $\Sigma$ are cosatisfiable, $\Sigma \cup T_1 \cup T_2$ is unsatisfiable. We must show that if the procedure of section 3 reports "satisfiable," $\Sigma \cup T_1 \cup T_2$ is satisfiable. This is done by showing how to construct a model of $\Sigma \cup T_1 \cup T_2$ when the procedure reports "satisfiable."

Let $C = \{c_0, \ldots, c_n\}$ be the set of constant symbols appearing in $\Sigma_1$ or $\Sigma_2$. Let $E$ be the set of equalities propagated by the procedure of section 3. As we will see, when the procedure halts, $E$ contains all the $c_i = c_j$ such that $c_i, c_j \in C \wedge c_i = c_j \notin E$. Since the schemes for $T_1$ and $T_2$ reported "satisfiable," there are models of $\Sigma_1 \cup T_1 \cup E$ and $\Sigma_2 \cup T_2 \cup E$. Let $M_1$ and $M_2$ be models of $\Sigma_1 \cup T_1 \cup E$ and $\Sigma_2 \cup T_2 \cup E$ respectively that agree on the interpretation of the equalities in $E$. We show how to construct a model $M = \Sigma \cup T_1 \cup T_2$ from $M_1$ and $M_2$.

Before giving this construction, we show that it is possible to pick an $M_1$ and an $M_2$ that agree on $E$. First note that if $E$ is empty, all $M_1$ and $M_2$ agree. Now suppose that $E$ is not empty. In this case, there exists an $M_1$ and an $M_2$ that do not satisfy any equality in $E$. For suppose to the contrary. In particular, suppose that every $M_i$ satisfies some equality in $E$. If $E$ contains exactly one equality, $c_1 = c_2$, $\Sigma_1 \cup T_1 \cup E \models c_1 = c_2$, and $E$ contains exactly one equality. If $E$ contains more than one equality, $\Sigma_1 \cup T_1 \cup E$ entails the disjunction of equalities in $E$. But then $\Sigma_1 \cup T_1 \cup E$ is nonconvex which is impossible because, instead of returning satisfiable, the algorithm in section 3 would have case split in this situation. This same argument can be made for $M_2$ and, hence, there exists an $M_2$ that does not satisfy any of the equalities in $E$. Thus, we can choose an $M_1$ and $M_2$ that agree on the interpretation of the equalities in $E$.

Note that since $M_1$ and $M_2$ agree on the interpretation of the equalities in $E$ and in $E$, they agree on the interpretation of every equality between constants in $C$.

Let $M_1 = \{D_1, R_1, F_1, C_1\}$, where $D_1$ is the domain of $M_1$, $R_1$ is the interpretation of relation symbols of $M_1$ in $D_1$, $F_1$ is the interpretation of the functions symbols of $M_1$ and $C_1$ is the interpretation of individual constant symbols in $M_1$. Similarly, let $M_2 = \{D_2, R_2, F_2, C_2\}$.

We now construct $M$ by merging $M_1$ and $M_2$ as follows. The domain of $M$ is $D_1 \cup D_2$, where $D_2$ is the domain of $M_2'$. A modified version of $M_2$. $M_2'$ is obtained by replacing individuals in $D_2$ by individuals in $D_1$ when they are designated by the same constant symbol. For all constant symbols $c \in C$, replace every occurrence of $C_2(c)$ in $D_2$ by $C_1(c)$, i.e., $C_2(c) = C_1(c)$ when $c$ is a shared constant symbol and $C_2(c) = C_2(c)$ otherwise. For all $R$ in the domain of $D_2$, let $R_2(R)$ be the set $R_2(R)$ modified by the above replacement procedure. Similarly, let $F_2'$ be the new interpretation of the function symbols of $M_2$. $M_2' = \{D_2', R_2', F_2', C_2'\}$.

$M_2$ and $M_2'$ are isomorphic structures because $M_1$ and $M_2$ agree on the interpretation of every equality between constants in $C$. If $M_1$ and $M_2$ did not agree, then $M_2$ and $M_2'$ would not be isomorphic. For suppose that $M_1 \models c = c_1$ but $M_2' \models c = c_2$. Then the two constant symbols designate the same individual in $D_2$ and different individuals in $D_2$, and hence, $M_2'$ is not isomorphic to $M_2$.

To finish the construction of $M$, we take $M = \{D_1 \cup D_2, R_1 \cup R_2, F_1 \cup F_2', C_1 \cup C_2\}$. Since $M_1 \models \Sigma \cup T_1$ and $M_2' \models \Sigma \cup T_2$, $M \models \Sigma \cup \Sigma_1 \cup \Sigma_2 \cup T_1 \cup T_2$. Since $\Sigma_1 \cup \Sigma_2$ and $\Sigma$ are cosatisfiable, $M \models \Sigma \cup T_1 \cup T_2$ and the proof of the theorem is complete. \(\Box\)

The fact that DRAT designs satisfiability procedures is a direct consequence of theorem 1. Since the result of combining two schemes is a scheme, any number of schemes can be combined by this method.

**DRAT DOES ISOMORPHIC REFORMULATION**

This section includes the proofs of two properties of DRAT's reformulation procedure $R$. These results are sufficient to show how a satisfiability procedure generated by DRAT for some reformulated theory can be used to solve the original problem.

**Lemma 1** If a reformulation rule (rule) is an extending definition in $T$ of the form $\langle P, Q, \Theta \models \psi \rangle$ and $T \models \psi$, then $R(\text{rule}, T)$ is an isomorphic reformulation of $T$.

**Proof:** The condition that must be met is that if $T \models \psi$, $T \models \phi \Rightarrow R(\text{rule}, T) \models R(\text{rule}, \phi)$. For any clause $\phi \in L(T)$. We prove the equivalent fact that if $T \models P$,
SAT(T \cup \{ \phi \}) \equiv SAT(R(\text{rule}, T) \cup \neg R(\text{rule}, \phi)),
\text{where SAT}(T) \text{ means that } T \text{ is satisfiable.}

[\Rightarrow] \text{If SAT}(T \cup \{ \phi \}), SAT(T \cup \{ \theta \} \cup \{ \neg \phi \})
\text{because, by the definition of extending definition, every model of } T \text{ can be extended to a model of } T \cup \{ \theta \} \cup \{ \neg \phi \}.
\text{Therefore, there exists a model of } T \cup \{ \theta \} \cup \{ \neg \phi \}.
\text{But } T \cup \{ \theta \} \cup \{ \neg \phi \} = R(\text{rule}, T) \cup \neg R(\text{rule}, \phi).
\text{Hence every model of } T \cup \{ \theta \} \cup \{ \neg \phi \} \text{ is a model of } R(\text{rule}, T) \cup \neg R(\text{rule}, \phi).
\text{Since there exists a model of } T \cup \{ \theta \} \cup \{ \neg \phi \}, \text{there exists a model of } R(\text{rule}, T) \cup \neg R(\text{rule}, \phi) \text{ hence, it is satisfiable.}

[\Leftarrow] \text{The proof in this direction is similar, with the added step of showing that every model of } R(\text{rule}, T) \cup \neg R(\text{rule}, \phi) \text{ can be extended to a model of } R(\text{rule}, T) \cup \{ \theta \} \cup \{ \neg \phi \} \text{ as follows. For each literal } \text{rule to rewrite occurrence of } \theta \text{ to occurrences of } \neg \phi.
\text{By the definition of } R, \text{the clauses of } Q \text{ will appear in } R(\text{rule}, T) \text{ and hence } R(\text{rule}, T) \models Q. \text{ Therefore, every model of } R(\text{rule}, T) \text{ can be extended to a model of } R(\text{rule}, T) \cup \{ \theta \} \cup \{ \neg \phi \} \text{ satisfies } \{ \theta \} \cup \{ \neg \phi \} \text{ so is } T \cup \{ \theta \} \cup \{ \neg \phi \}.

\text{Lemma 2 For any reformulation rule } (\text{rule}), \text{the function } \lambda(t).R(\text{rule}, t) \text{ is computable.}
\text{Proof: Suppose the biconditional of } \text{rule} = \theta \Rightarrow \psi \text{ and } R \text{ applies } \text{rule to rewrite occurrence of } \psi \text{ to occurrences of } \theta \text{ in } T \text{ as described in section } S. \text{ Since } \text{rf-symbols}(\theta) \text{ are disjoint from } \text{rf-symbols}(T), \text{ a rewrite step can never introduce a pattern of literals to which } \text{rule} \text{ can be applied a second time. The rewrite is applied repeatedly until one of the following events occurs: (1) all of the symbols in } \text{rf-symbols}(\psi) \text{ are removed from } T \text{ or (2) no new occurrences of } \psi \text{ can be found, even though symbols in } \text{rf-symbols}(\psi) \text{ are still present. In either case, repeated application of the rewrite rule terminates. Hence, } \lambda(t).R(\text{rule}, t) \text{ is computable.}

\text{The two preceding lemmas are sufficient to show that a satisfiability procedure for } R^*(\Sigma) \text{ can be used to solve the problem } < \Sigma, T^c, \Phi >, \text{ so long as } R^*(\Sigma) \text{ is a conjunction of ground literals. Assuming that } R^*(\Sigma) \text{ is a conjunction, the satisfiability procedure is used to solve the problem by solving } < R^*(\Sigma), R^*(T^c), R^*(\Phi) > \text{ as follows. For each } \phi \in \Phi, \text{ if } \neg R^*(\phi) \text{ is a conjunction of literals, we use the procedure to determine if } R^*(\Sigma) \cup R^*(T^c) \cup \neg R^*(\phi) \text{ is unsatisfiable. Thus the case if and only if } R(\Sigma, T^c, \Phi) \text{ is unsatisfiable. If } \neg R^*(\phi) \text{ is a disjunction of literals, the procedure is used to determine the satisfiability of } R^*(\Sigma) \cup R^*(T^c) \cup \phi, \text{ for each literal } \phi \in \neg R^*(\phi). If any of these is satisfiable, } R^*(\Sigma) \cup R^*(T^c) \cup \neg R^*(\phi) \text{ is satisfiable; otherwise it is unsatisfiable.}

\text{THE COMPLETENESS OF DRAT}
\text{Two simplifying assumptions were made in the previous sections. First, in definition 3, it was assumed that a problem for DRAT was of a restricted form. Second, it was assumed that DRAT's success depended on designing a satisfiability procedure for all of } T^c. \text{ Both of these assumptions are now relaxed and we show how a literal satisfiability procedure is interfaced with a resolution theorem prover in such a way that the procedure/theorem prover combination is complete.}

\text{A problem for DRAT is now taken to be a pair } < T, \phi >, \text{ where } T \text{ is a set of first-order formulas and } \phi \text{ is a first-order formula. A pair } < T, \phi > \text{ is interpreted as the question, } "T \models \phi"?\]

\text{As a typical preprocessing step for resolution theorem proving, } T \text{ and } \neg \phi \text{ are converted to sets of clauses which will be called } T_1 \text{ and } \neg \phi \text{ respectively. Let } L T_1 \text{ be the set of non-ground clauses in } T_1. \text{ As before, DRAT is used to design a literal satisfiability procedure for } T^c. \text{ However, instead of exiting with failure if it is unable to design a procedure for all of } T^c, \text{ it returns the satisfiability procedure and } T^c, \text{ those clauses not incorporated into the satisfiability procedure. Also, as before, DRAT returns the reformulation map } R^*.

\text{The algorithm given in section refers to the set of clauses for which a literal satisfiability procedure has been designed as } T_f. \text{ Here that procedure is referred to as } S_T_r. \text{ We show how } S_T_r \text{ is used along with a resolution theorem prover to demonstrate the unsatisfiability of } CI = R^*(T^c) \cup R^*(\neg \phi). \text{ The non-ground clauses of } CI \text{ are manipulated by the theorem prover in the usual way, except that clauses in } T_f \text{ are prohibited from resolving with ground clauses. These resolutions are unnecessary because } S_T_r \text{ is a "compression" of any resolution steps that can result from such a resolvent.}

S_T_r \text{ is used in the manipulation of ground clauses in } CI \text{ and ground clauses derived from } CI \text{ during theorem proving. It is interfaced to the theorem prover via theory resolution[Stieke85]. One type of theory resolution, called total narrow theory resolution, requires a decision procedure for a theory } T \text{ given a set of literals } L, \text{ to compute subsets } L' \text{ of } L \text{ such that } L' \cup T \text{ is unsatisfiable. Such a procedure is used to compute } T \text{-resolvants of a set of clauses as follows. Consider the decomposition of the clauses into } K_i \lor L_i, \text{ where each } K_i \text{ is a single literal in } L(T) \text{ and } L_i \text{ is disjunction of literals (possibly empty). For each subset of the } K_i, \text{ say } \{ K_{i1}, \ldots, K_{in} \}, \text{ determine if there exists a model of } T, \text{ the clause } L_1 \lor \ldots \lor L_n \text{ is a } T \text{-resolvent.}

\text{The theorem prover constructs } T_f \text{-resolvants from ground clauses, using } S_T_r \text{ to compute sets of ground literals that are unsatisfiable in } T_f. \text{ Let } GrL \text{ be the set of ground unit clauses in } CI \text{ and let } GrCI \text{ be the set of ground non-unit clauses in } CI. \text{ First, the ground clauses are separated into clauses that are in } L(T_f) \text{ and clauses...}
that are not. This is accomplished for the clauses in \( GrL \) using the procedure described in section; it is accomplished for clauses in \( GrCl \) in a similar fashion.

If a ground clause \( e_1 \) contains a literal that is not in \( L(T_I) \) and a ground clause \( e_2 \) contains the negation of that literal, the theorem prover computes the resolvent of \( e_1 \) and \( e_2 \) in the normal way. \( T_I \)-resolvants are computed using \( S_{T_I} \) to compute sets of ground literals that are unsatisfiable in \( T_I \) as follows. Let \( GrLT_I \) be the set of literals in \( GrL \) that are in \( L(T_I) \). Let \( GrCIT_I \) be the set of literals in \( L(T_I) \) appearing in clauses of \( GrCl \). We input progressively larger subsets of \( GrLits = GrLT_I \cup GrCIT_I \) to \( S_{T_I} \) as long as those sets are satisfiable in \( T_I \). Once a set is unsatisfiable in \( T_I \), all supersets of it will also be unsatisfiable. When the theorem prover deduces a new ground literal in \( GrLT_I \), it is added to \( GrLits \). The smallest subsets of \( GrLits \) found to be unsatisfiable in \( T_I \) are used to compute \( T_I \)-resolvants of ground clauses.

**Theorem 2** Given the problem \( < \Gamma, \phi > \), let \( S_{T_I} \) be a literal satisfiability procedure for \( T_I \subseteq \mathcal{R}(\Gamma) \). If \( \Gamma \models \phi \), \( S_{T_I} \) combined with the theorem prover will demonstrate the unsatisfiability of \( Cl \).

**Proof:** In [Stickel85], Stickel shows that, given a set of clauses \( K, \vee L_i \), if a decision procedure for a theory \( T \) computes all subsets \( k_i \) that are minimally unsatisfiable in \( T \), total narrow theory resolution is complete. We must show that the above procedure for computing \( T_I \)-resolvants computes all subsets of \( GrLits \) that are minimally unsatisfiable in \( T_I \). Clearly, so long as \( S_{T_I} \) is a literal satisfiability procedure, the above procedure computes all these subsets. Thus, the completeness result follows directly from the results of section.

The procedure described above can be made much more efficient. There are several refinements used by the DRAT implementation to consider far fewer subsets for unsatisfiability in \( T_I \). We discuss two of these here. One refinement is to distinguish between literals in \( GrLT_I \) and \( GrCIT_I \). First, we consider the satisfiability of \( GrLT_I \). If this is unsatisfiable, we are done. Otherwise, we consider progressively larger sets of literals appearing in clauses in \( GrCIT_I \). For each such set \( s \), \( S_{T_I} \) is used to determine whether or not \( GrLT_I \cup s \) is unsatisfiable in \( T_I \).

Note that the subsets identified with this refinement are not always minimal: it is possible for a subset of \( GrCIT_I \) union a subset of \( GrLT_I \) to be unsatisfiable in \( T_I \). However, it turns out that completeness of theory resolution is retained in this case, since the extraneous literals are in \( GrLT_I \) and, therefore, are unit clauses.

A second simpler refinement only considers subsets of \( GrCIT_I \), each of whose elements appears in a different clause in \( GrCl \).

As a final point about the efficiency of the procedure for computing subsets that are minimally unsatisfiable in \( T_I \), recall that schemes are required to be incremental. Because of this, \( S_{T_I} \) is used very efficiently to consider progressively larger sets of literals.

It is often most effective to leverage the use of \( S_{T_I} \) by doing as much of the theorem proving as possible at the "ground level." The DRAT implementation uses "set of support" strategy which is very effective in accomplishing this when \( \neg \phi \) is ground because it tends to produce ground resolvants.

**Summary and Ongoing Work**

We have presented a formalization of DRAT: a technique for automatic design of satisfiability procedures. We have shown how these procedures are interfaced to a theorem prover so that it can, in many cases, prove theorems more efficiently. Given \( \Psi \), the set of axioms of a problem, and \( S_{\Psi} \), a literal satisfiability procedure designed for \( \Psi \subseteq \mathcal{R}(\Psi) \), we have proven that for any first-order statement \( \phi \), if \( \Psi \models \phi \), the theorem prover/\( S_{\Psi} \) combination will prove \( \phi \).

The major steps of our argument were as follows:

1. We showed that a combination of satisfiability procedures with certain properties is again a satisfiability procedure.
2. We showed that the reformulation that is essential to DRAT's effectiveness is isomorphic reformulation and, therefore, a satisfiability procedure of a reformulated theory can be used to solve problems in the original theory.
3. We proved the completeness of our technique for combining literal satisfiability procedures with a theorem prover. In this combination, \( S_{\Psi} \) is used to compute \( \Psi \)-resolvants from ground clauses and the theorem prover is restricted so that it does not resolve ground clauses on literals in \( L(\Psi) \).

In our ongoing work, we are attempting to extend DRAT’s scheme combination technique. As much as possible, we would like to remove the restriction on the sharing of nonlogical symbols between component scheme instances in combinations. We are exploring the conditions under which limited types of overlap between nonlogical symbols is allowed. When overlap is allowed, component schemes must propagate more information than just equalities between constant symbols. In most cases where overlap is allowed and in which the schemes propagate at least the set of equalities between constants, it is not difficult to show the completeness of a propagation technique. The major issue that arises is proving that the propagation terminates.

As an example, consider allowing two schemes to share function symbols. The schemes must propagate all equalities between ground terms involving shared function symbols. The proof technique used in section can be extended to prove that such schemes combined by an appropriately extended propagation technique will produce semi-decision procedures for the combinations of their theories. However, in general, it is not possible to prove that the propagation will terminate.
One situation in which overlap is allowed occurs when the theories of schemes are sets of clauses in a sorted first-order logic. In this case, a function symbol $F$ whose range is disjoint from its domain can be shared between schemes because terms of the form $F(F(x))$ are not well formed and, hence, it is easy to show that propagation of terms involving $F$ will terminate.

Acknowledgements Richard Fikes, Bob Nado, Mike Lowry, and David McAllester provided helpful comments on drafts. Dave McAllester pointed out the error in Nelson & Oppen's proof of the combination of satisfiability procedures and suggested the technique we used in our proof. Bob Nado participated in numerous discussions on many aspects of the paper.

References


Korf, R.E., "Toward a Model of Representation Changes," Artificial Intelligence, 14, pp.41-78, 1980.


