Resolution Strategies

Various strategies have been devised to make resolution theorem proving reasonably efficient, some of which preserve completeness

In general, the search for a refutation of a set of clauses is a search in an exponentially explosive search space. The more resolvents we add to the original clauses, the more options we have in choosing additional pairs of clauses for resolving. Thus, if we apply resolution (along with factoring and paramodulation, if relevant) blindly, we are unlikely to find refutations in a reasonable number of steps, even for intuitively simple problems. So much theorem proving research has focused on finding good strategies.

An important issue in the formulation of a resolution strategy is the preservation of completeness; i.e., it may be that a strategy prevents us from finding a refutation in some cases, even though a refutation exists. Consequently much of the research on resolution theorem proving since Robinson's introduction of the method has been concerned with proving the completeness (or otherwise) of various resolution strategies and combinations of strategies. Another important line of research concerns ways of limiting the expressiveness (syntax) of the clausal logic we're working with so that refutations can be found efficiently. The most interesting sublanguage that has been found is *Horn logic*, which is used in Prolog.

The following are some strategies that have proved useful in practice, and in many cases they also come with guarantees of completeness.

Clause elimination strategies

First of all, there are a number of strategies for eliminating clauses that cannot usefully contribute to a refutation. Clearly, limiting the size of the "working set" of clauses will also help to contain the combinatorial explosion in the search for a refutation.

Pure literal elimination: A literal is *pure* if there is no complementary literal in the set of clauses we are working with. For example, the literal P(A) in the clause P(A) $\vee$ Q(x,A) is pure if there are no occurrences of $\neg$P(...) elsewhere among the clauses, or if the only such occurrences are not unifiable with P(A), e.g., $\neg$P(B). Since a pure literal can never be "resolved away", it's clear that we can eliminate all clauses containing pure literals from consideration. Note that when we eliminate a clause, other literals may become pure relative to the reduced clause set, and so further eliminations may be possible. Pure literal elimination evidently preserves completeness.

Tautology elimination: A *tautology* is a clause containing two literals where one is the

negation of the other; e.g., $P(A) \vee \neg P(A) \vee P(B)$, $P(x) \vee \neg P(x) \vee Q(A,x)$. Intuitively, such a clause is "trivially true" and as such cannot help with any refutation. Note that $\neg P(A) \vee P(x)$ ("if A has property P then everything has property P"), $\neg P(x) \vee P(A)$ ("If there is anything with property P then A has property P"), and $P(x) \vee \neg P(y)$ ("Either everything has property P, or nothing does") are intuitively nontrivial, and in fact they are not tautologies.

Let's try to verify our intuition that tautology elimination preserves completeness. In other words, we want to show that if a set of clauses containing a tautology can be refuted, then it can also be refuted without the tautology. For simplicity we just consider the case where all our clauses are ground clauses. So consider a refutation that involves a use of a tautology, say $\pi \vee \neg\pi \vee \rho$, where ρ consists of zero or more literals. Then this refutation must also include steps that "resolve away" π and $\neg\pi$ (as well as all the literals of ρ). Then there must be another pair of clauses that were used in these steps, say $\neg\pi \vee \rho'$ and $\pi \vee \rho''$. Then the rest of the refutation must also "resolve away" all of the literals of ρ, ρ', and ρ''. But then we could have obtained the empty clause just by 1. resolving $\neg\pi \vee \rho'$ against $\pi \vee \rho''$ to obtain $\rho' \vee \rho''$; and 2. resolving away the literals of ρ' and ρ'' by the same steps as were used in the assumed refutation. (Also we would resolve away any additional literals introduced by these steps as before.)[1] In the nonground case, we can make a similar argument, using the fact that if the literals in the two extra clauses can be resolved against π and $\neg\pi$ respectively (in succession), then they can also be resolved against each other.

Subsumption elimination: A clause ϕ *subsumes* another clause ψ if the literals of ϕ, or some substitution instance of these literals, are a subset of the literals of ψ. For instance,

$P(A)$ subsumes $P(A) \vee Q(B)$,
$P(x)$ subsumes $P(A)$,
$P(x)$ subsumes $P(y) \vee Q(y)$,
$P(f(x),B) \vee Q(y)$ subsumes $P(f(B),B) \vee Q(A) \vee v = w$.

Note that the subsuming clauses on the left are all intuitively "stronger" than the subsumed clauses – they allow fewer alternatives, and make claims about more of the individuals in the domain of discourse (when they have variables where the subsumed clause has functional terms or constants).

We can test whether ϕ subsumes ψ by trying to find a unification that unifies the literals of ϕ with some of the literals of ψ, where the unifier is allowed to substitute for variables in ϕ but not variables in ψ. It's easy to see that elimination of subsumed clauses preserves completeness. Any refutation that uses a subsumed clause can be modified to use the subsuming clause instead. The resolvent will be at most as long, and at least as

[1] It may be that some of the steps in the rest of the refutation, after the assumed use of the tautology, again involve use of the same tautology. However, we've shown how to eliminate one use of it, so clearly we can eliminate all uses of it.

general, as in the original proof. Thus the refutation can still be completed, in at most as many steps as before.

Finally note that tautology and subsumption elimination are not just preprocessing strategies, but can be used throughout a proof attempt, since resolvents (and factors and paramodulants) can be tautologous and can subsume or be subsumed by clauses already present.

Proof strategies

Breadth-first

A systematic strategy with the sole advantage of being simple and complete is the *breadth-first* strategy. Roughly speaking, this corresponds to forming all 1-step proofs, then all 2-step proofs, and so on, until we find a proof of the empty clause. More exactly, suppose we designate the original clauses (premises and denial clauses) as level-0 clauses. Then the level-1 clauses are the resolvents, factors, and paramodulants of level-0 clauses. (Note: there are only finitely many.) The level-2 clauses are the resolvents and paramodulants with one level-0 parent and one level-1 parent, and the factors of level-1 clauses. In general, a resolvent or paramodulant with one level-i parent and one level-j parent yields a level-(i+j+1) clause, and a factor of a level-i parent yields a level-(i+1) clause.

The breadth-first strategy consists of first generating all level-1 clauses, then all level-2 clauses, and so on. It's not hard to see that the level of a clause is the number of proof steps used to derive it, so obviously if there is a refutation of the original clauses, we will eventually find it. However, chances are we'll do a huge amount of unnecessary work in finding a refutation, so the breadth-first strategy is not used in practice. It can be theoretically useful in showing that certain other strategies, such as "unit preference" (below) preserve completeness.

Depth-first

In a *depth-first* strategy we first derive a level-1 clause, then immediately try to derive a level-2 clause using that level-1 clause, then try to derive a level-3 clause from the level-2 clause (if any), and so on. If at some point we cannot derive a level-i clause, we see if there is a further level-(i-1) clause we can derive and do so if possible; etc. In other words, we always try to derive a clause at the highest possible level, using an instance of resolution, factoring, or paramodulation that we haven't used yet.

This strategy is as simple as the breadth-first strategy and is often rather efficient (at least compared to the latter), but it is not complete. This is because it is possible to go off on an infinite regress that never yields the empty clause, even when a finite refutation exists.

Unit resolution

A *unit resolution* step is one where at least one parent clause is a unit clause. Note that if we resolve a unit clause against a clause of length n, the result is a clause of length n-1. By contrast, non-unit resolution always gives a resolvent at least as long as the longer of the two parent clauses. Since our goal is to derive the empty clause, doing unit resolutions should take us toward this goal more quickly.

However, a unit resolution strategy, where we insist on doing only unit resolutions, is incomplete. For example, the following four clauses are easily refuted, but no unit resolution steps are possible initially: $\{P \vee Q, P \vee \neg Q, \neg P \vee R, \neg P \vee \neg R\}$.

Unit preference

Unit preference modifies a strict unit resolution strategy by insisting only that unit resolution steps should be performed as long as new unit resolution steps are possible, while at other times there are no constraints on permissible resolution steps. (A way of carrying over the idea of unit resolution to other steps is to always perform those steps in which at least one parent clause is as short as possible.)

This strategy turns out to be complete, in the sense that there exists a unit preference proof (refutation) whenever there exists any proof. So this strategy is commonly used in practical resolution theorem proving systems, often in combination with other methods, such as the next one.

Set-of-support

The *set-of-support* strategy is a particularly important and effective one, since it implements the idea of focusing the theorem proving attempt on the "goal" – the theorem to be proved. Without such a strategy, a theorem prover might endlessly generate conclusions from a knowledge base without paying particular attention to the specific result we are after.

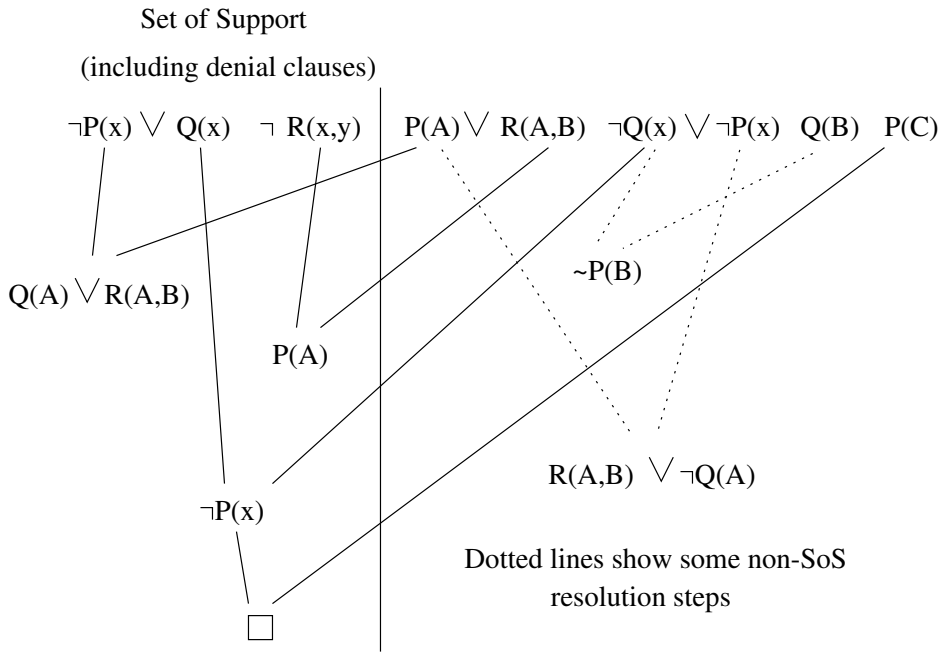
In practice, the set-of-support strategy usually consists of requiring each inference step to use at least one denial clause, or a descendant of a denial clause, as a parent. Of course, as long as we are using such a strategy, *every* new clause we derive is a descendant of a denial clause. The denial clauses and their descendants are called the *set of support*. Note that since the denial clauses come from the denial of the theorem to be proved, this is a way of focusing on the goal of the theorem-proving attempt.[2]

Another way of understanding the set-of-support strategy is this: since we can generally be reasonably sure that the premises we are using in a theorem proving attempt are consistent, no amount of reasoning based entirely on the premises will ever lead to the empty clause. The only way a contradiction can arise is that the denial of the de-

[2]More generally, the initial set of support may consist of any subset of the set of clauses such that removing that subset leaves us with a satisfiable set of clauses. However, if that subset is too large, we don't gain much. In practice, the smallest set of support we can identify with some confidence is usually the set of denial clauses.

sired conclusion is either self-contradictory (i.e., the denial clauses by themselves lead to a contradiction) or contradicts the premises. Therefore it is crucial that we focus our effort on the denial clauses in the refutation attempt.

The following picture illustrates the set-of-support strategy. To the left of the vertical line we have the set of support, consisting of the denial clauses and their descendants; this includes the empty clause, i.e., the refutation attempt succeeded. To the right we have the remaining clauses (the premises), and some non-set-of-support resolvents are indicated with dotted lines.



As a final comment, note that the set-of-support strategy is complete even when combined with certain other strong strategies, such as unit preference (also connection-graph resolution, below).

The connection-graph method

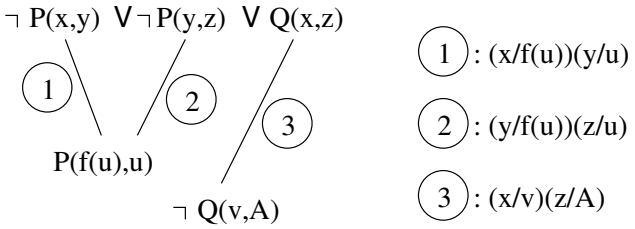
The *connection-graph* method starts off by setting up possible "resolving links" between all possible unifiable, complementary pairs of literals (from distinct clauses). It then chooses one of those links, computes the resolvent, and deletes the link resolved upon. The literals of the resolvent "inherit" the links of the corresponding parent literal. All "resolving links" are labelled by the unifier of the literals they connect.

If initially a clause has a literal with no link to any other literal, that clause is deleted. Note that this is really a version of pure literal elimination: an unconnected (and hence unresolvable) literal cannot possibly be gotten rid of, so there is no point in using a clause with such a literal. We also apply this strategy during the proof: when a link that

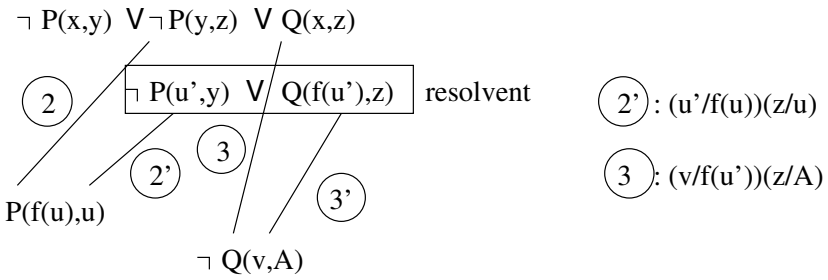
was just resolved upon is deleted, this may leave either of the two parent clauses with a disconnected literal, and if so, we remove the clause concerned. In removing clauses, we also remove their connections, so the clause deletion may "snowball", leaving us with a greatly reduced set of clauses.

Also, there is another way that clauses may get deleted: when links are inherited by a resolvent, the unifiers on the inherited links are replaced by their "composition" with the unifier used to obtain the resolvent (e.g., if a link label was (x/f(y))(u/A) and the unifier used in the resolution step was (y/B)(z/C), then the revised link label is (x/f(B))(u/A)). But this composition may not exist (e.g., if the unifier had been (y/B)(u/C)), and in that case the inherited link disappears. But this may again lead to disconnected literals, and hence to clause deletion.

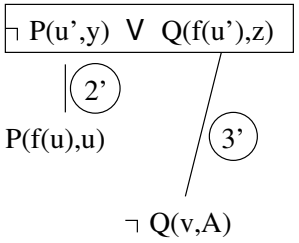
The following is a simple example of a connection-graph proof



Suppose we choose (1) as the first resolution step. So we form the resolvent, eliminate edge (1), and inherit appropriate versions of edges (2) and (3) to the resolvent:



At this point clause 1 and its arcs are dropped, since it has an unconnected literal:

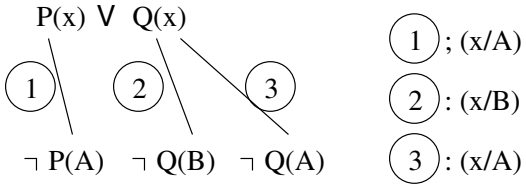


If we now choose (2'), then the two literals it connects be–come unconnected and so the two clauses it connects van–ish, and hence so does edge (3').

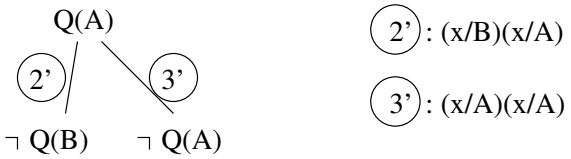
This leaves only the following resolvent and one previous unit clause:

This immediately yields the empty clause.

An example where "inheritance failure" of an edge leads to clause elimination is as follows:



If we now resolve on (1):



Clearly the substitutions (x/B)(x/A) are incompatible, hence edge (2') and along with it the clause ¬ Q(B) can be eliminated.

A disadvantage of the connection-graph method is its rather high front-end cost – we may have $O(n^2)$ connections to compute before we can start resolving. This is not good if there is a 1-step proof, for example. However, the method tends to work well for harder problems, especially when combined with a set-of-support strategy.

Ordered resolution

Ordered resolution restricts resolution to the *initial* literals of the parent clauses; i.e., we can only resolve the initial literal of one clause against the initial literal of another. Evidently, this greatly reduces the number of resolution steps allowed at any point, and so restricts the combinatorial explosion of the search space.

This strategy is not complete in general. However, it *is* complete for *Horn clauses*. These are clauses with at most one positive literal, such as

$$P(A),\ \neg Q(x),\ \neg P(x) \vee Q(x).$$

Horn clauses are the basis for the PROLOG programming language, and in that context are generally written in backward implicative form, e.g.,

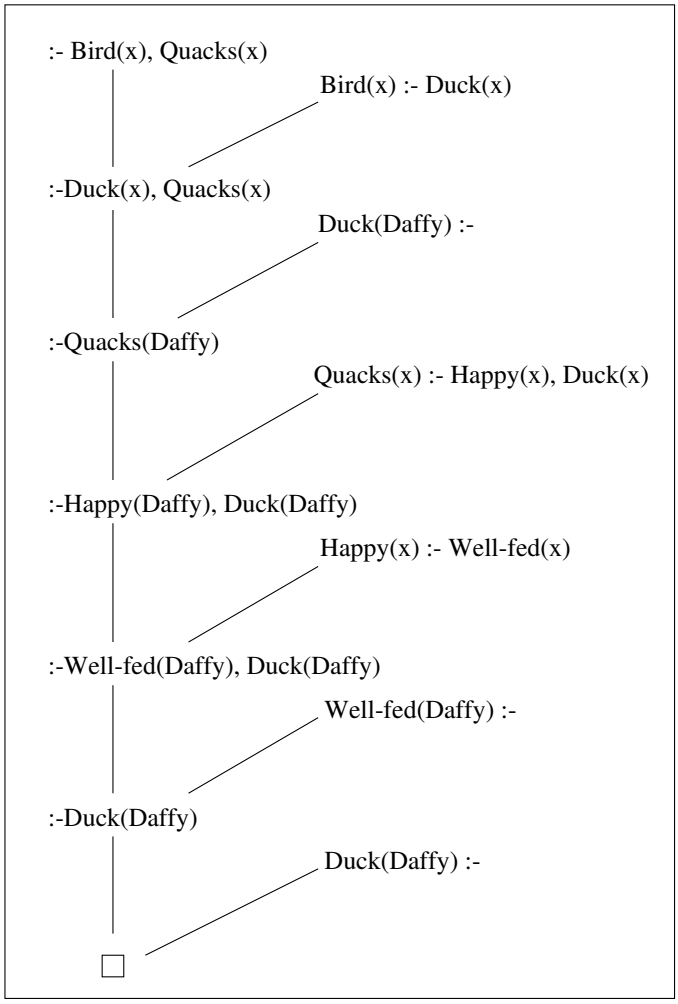
$$P(A) \Leftarrow \text{True},\ \text{False} \Leftarrow Q(x),\ Q(x) \Leftarrow P(x).$$

(You should verify that these are equivalent.) In PROLOG, these are further abbreviated as

P(A) :-, :- Q(x), Q(x) :- P(x).

PROLOG uses ordered resolution along with set-of-support, and this typically leads to very efficient theorem proving. In the above set of clauses, suppose that :- Q(x) is the goal clause. (Note: this is what we would get from denying a desired conclusion ($\exists$x Q(x)). So we can think of :- Q(x) as stating the goal of proving that Q(x) holds for some x.) Then ordered resolution, and set of support, allow us to resolve the second and third clauses, giving :- P(x); i.e., this is our new subgoal. Now we resolve against the first clause, and get the empty clause, establishing the original goal.

Here is a more meaningful example borrowed from James Allen's lecture notes:



The goal here is to show that there is a bird that quacks. The negation of this goal leads to the clause at the top of the figure. The first fact we use is that every duck is a bird. The resultant clause in Horn form can be viewed as a new goal of showing that there is a duck that quacks. Using the given fact that Daffy is a duck, this further reduces to the goal of showing that Daffy quacks. We now bring in the fact that all happy ducks quack, and this leads to the goal of showing that Daffy is a happy duck; and so on.

Note that the fact that we are using ordered resolution, and that we begin with the goal of the proof attempt (thus ensuring a set of support strategy), in this case leads to a completely deterministic refutation. At each step, exactly one of the available facts can be resolved against the current goal, given the ordered resolution constraint.

The PROLOG proof illustrates "backward reasoning", but the same KB could be used for "forward reasoning". Given that Daffy is a duck and well-fed, as new facts, we could systematically apply the other general facts and thus "discover" that Daffy is happy and quacks. In a knowledg-based system that does automatic forward reasoning, the answers to many questions the user might later ask will already have been worked out in advance.

Model generation

Model generation (or elimination) is one of the more recent, and most successful, resolution strategies.[3] The term "model generation" is somewhat deceptive – the strategy can be described in purely syntactic terms. Essentially we try systematically to find a set of positive unit ground clauses which in a sense define a model – a so-called "Herbrand model", in which individual constants and other ground terms are interpreted as denoting *themselves*. If the model generation fails, we have refuted the original clauses.

The strategy assumes that all clauses are *range-restricted*: any variables that occur in the positive literals of a clause also appear in its negative literals. (If we put clauses in implicative form, making all negative literals antecedent literals and all positive literals

[3]See R. Manthey & F. Bry, "SATCHMO: a theorem prover implemented in Prolog", *9th CADE*, May 1988; and more recently, D. Loveland, "METEOR: Exploring model elimination theorem proving", *Journal of Automated Reasoning 13*, 1994, 283-296.

consequent literals, then this means that all variables in the consequent must appear in the antecedent.) In particular this implies that clauses consisting entirely of positive literals must be ground clauses.

The importance of the range-restrictedness assumption is that it allows us to gradually add unit ground clauses to the clause set by forward inference, in such a way that these unit clauses define a (Herbrand) model – and if we have exhausted all ways of trying to construct such models, then we have refuted the original clauses. For example, if we have clauses

$$\mathrm{P(x)} \Rightarrow \mathrm{Q(x)},\ \ \mathrm{P(A)},\ \ \mathrm{P(B)},\ \ \mathrm{Q(C)}$$

then the Prolog inference mechanism would immediately derive consequences

$$\mathrm{Q(A)},\ \ \mathrm{Q(B)},$$

and together with the original unit clauses these define a Herbrand model with domain ("universe") {A,B,C} and interpretation {A,B} for P and interpretation {A,B,C} for Q.

Handling non-range-restricted clauses can be problematic, even though such clauses are easily converted to range-restricted form. For instance, we can convert

$$\mathrm{Unique(Our\text{-}Cosmos)} \Rightarrow \mathrm{Part\text{-}of(x,Our\text{-}Cosmos)}$$

("If our cosmos is unique, then everything is part of it") to

$$\mathrm{Unique(Our\text{-}Cosmos)} \wedge \mathrm{Thing(x)} \Rightarrow \mathrm{Part\text{-}of(x,Our\text{-}Cosmos)},$$

where we take 'Thing' to be universally true. However, how do we express that 'Thing' is universally true? We cannot write

$$\mathrm{Thing(x)},$$

since this is again not range-restricted! It turns out to be sufficient to posit

$$\mathrm{Thing}(C_1),\ \mathrm{Thing}(C_2),\ ...$$

for every constant C_i in the premise set, and

$$\mathrm{Thing}(x_1) \wedge ... \wedge \mathrm{Thing}(x_n) \Rightarrow \mathrm{Thing}(\mathrm{f}(x_1,...,x_n))$$

for every n-ary function symbol occurring in the premise set. This ensures that every object in the Herbrand universe – i.e., every ground term – is a 'Thing'. (Manthey and Bry use 'Dom' (domain) rather than 'Thing' for this special predicate). While this conversion only incurs a linear expansion in the size of the premise set in the worst case, the model construction may "blow up" as a result. For instance, in the above example we can now generate

$$\mathrm{Part\text{-}of}(C_1,\mathrm{Our\text{-}Cosmos}),\ \ \mathrm{Part\text{-}of}(C_2,\mathrm{Our\text{-}Cosmos}),\ ...$$

as well as

$$\mathrm{Thing}(\mathrm{f}(C_1,C_2,...)),\ \ \mathrm{Thing}(\mathrm{f}(C_2,C_1,...)),\ ...,\ \ \mathrm{Thing}(\mathrm{f}(\mathrm{f}(C_1,C_2,...),C_2,...)),\ ...\ \text{etc.},$$

and hence also

Part-of(f(C_1,C_2,...),Our-Cosmos), Part-of(f(C_2,C_1,...),Our-Cosmos), ..., etc.,

by forward inference, assuming we have constants C_1, C_2, ... and function F in the premises. Thus much subsequent research has focused on limiting such explosions in the model generation strategy.[4]

The SATCHMO strategy starts by dividing the set of clauses into Horn clauses and non-Horn clauses. We then try to refute the Horn clauses. If we succeed, we are done (successful refutation). We can do this efficiently using Prolog. If we fail, then we know there is a model of the current Horn clauses (and this model is determined by the positive unit clauses in the expanded Horn set).

We now try to extend this implicit model to the non-Horn clauses. We look for a non-Horn clause whose negative literals we can "resolve away" (i.e., in implicative form, a clause whose antecedent we can prove). If there is no such clause, we know there is a model of all the clauses and we can stop. If there is such a clause, then this clause can only be true in the model we are "generating" if its consequent is true, under the substitution that was used to prove the antecedent. So we *assume* one of the consequent literals, after applying the substitution – this will be a positive unit *ground* clause, because of the range-restrictedness assumption. We now add this unit clause to our Horn set, and run another Prolog refutation attempt.

If the refutation attempt fails, we have managed to extend the model (using the unit clause) so that it supports the truth of the non-Horn clause. We proceed likewise for the other non-Horn clauses, and if we manage to extend the model to support the truth of all of them (using various unit clauses derived from them), the model generation is complete and we cannot refute the original clauses. If we run into a non-Horn clause for which we cannot extend the model, we backtrack, and if we backtrack to the current clause, we try to assume the next consequent literal of that clause and go forward again.

If the assumed unit clause derived from the non-Horn clause does lead to a refutation of the (expanded) Horn set, then we backtrack immediately, i.e., we retract the assumed unit clause, and make a similar attempt for the next consequent literal of the non-Horn clause. If we succeed this time we can again proceed to the next non-Horn clause. If, however, all attempts to add a unit clause to the Horn set, based on the non-Horn clause, lead to a refutation of the (expanded) Horn set, then we know there is no model of the initial Horn clauses plus the selected non-Horn clause, i.e., we have refuted the clauses.

In general, we may have to try all combinations of assumptions for all the non-Horn clauses, before we have exhausted all ways of trying to generate a model, and thus have

[4] e.g., R. Rankin and R. Wilkerson, "Proving functionally difficult problems through model generation", *ACM/SIGAPP Symposium on Applied Computing*, 1992, 526 - 529; Lifeng He, "UNSATCHMO: Unsatisfiability checking by model rejection", *Int. Joint Conf. on Automated Reasoning* (IJCAR 2001), June 18 - 23, Siena, Italy, Short Papers, 64 - 75; Lifeng He, "I-SATCHMO: An improvement of SATCHMO", *J. of Automated Reasoning 27*(3), Oct. 2001, 313-322.

refuted the original clauses. However, since non-Horn clauses tend to be relatively rare, the method nonetheless tends to be efficient.

Here's a very simple example. Assume we have the following clauses:

$\neg P,\ \neg Q,\ \neg R \vee S;\quad \neg S \vee P \vee Q,\ \ R \vee P \vee Q.$

The Horn clauses are the ones before the semicolon. In implicative form:

$\neg P,\ \neg Q,\ \ R \Rightarrow S;\quad S \Rightarrow (P \vee Q),\ \ R \vee P \vee Q.$

Now the attempt to refute the initial Horn set will fail. So we look for a non-Horn clause whose antecedent we can prove. We cannot prove S (i.e., refute $\neg$S). However, we can "prove" the antecdent of the last clause, because it has no antecedent (or we can take the antecedent to be "True"). So now we add R to the Horn set, and try to refute the set. We fail, and so we have managed to extend the model. It's not hard to see that the extended model now allows us to prove S, the antecedent of $S \Rightarrow (P \vee Q)$.

So we now add P to the Horn set, and try to refute the set, and succeed. So we know we can't allow P to be true in the model, and retract this assumption. We now try assuming Q (the second consequent literal of $S \Rightarrow (P \vee Q)$) instead. The Prolog refutation again succeeds. So, given the assumption of R, there is no way to get a model. So now we backtrack, retracting that assumption, and assume P (the second literal of R $\vee$ P $\vee$ Q) instead. This also leads to inconsistency of the Horn set. So we try assuming Q instead, but this again leads to an inconsistent Horn set. This completes the refutation.