Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT

# Deduction Systems Based on Resolution

N. Eisinger, H.J. Ohlbach

SEKI Report SR-90-12

# Deduction Systems Based on Resolution*

Norbert Eisinger, Hans Jürgen Ohlbach
FB Informatik, University of Kaiserslautern
D-6750 Kaiserslautern,
Germany

**Abstract** A general theory of deduction systems is presented. The theory is illustrated with deduction systems based on the resolution calculus, in particular with clause graphs. This theory distinguishes four constituents of a deduction system: the logic, which establishes a notion of semantic entailment; the calculus which provides the syntactic counterpart of entailment; the logical state transition system, which determines the representation of formulae or sets of formulae together with their interrelationships, and also may allow additional operations reducing the search space; the control, which comprises the strategies and heuristics used to choose the most promising from among all applicable derivation steps.

For the last two levels many alternatives are presented and appropriately adjusted notions of soundness, completeness, confluence, and Noetherianness are introduced in order to characterize the properties of particular deduction systems. For more complex deduction systems, where logical and topological phenomena interleave, these properties can be far from obvious.

* This work was supported by the "Sonderforschungsbereich 314, Künstliche Intelligenz" of the German Science Foundation (DFG).

## Tal·le of Contents

## 1 Introduction

Statements about the real world or about fictive or abstract worlds are often interrelated in that some of them follow from others. For example, given the statements:

*"Every cat eats fish"*
*"Garfield is a cat"*
*"Garfield eats fish"*

one would agree that the third statement follows from the other two. Whenever we assume the first and the second statement to be true (in our considered world), we also have to accept the truth of the third statement. The same holds for the statements:

*"Every human is mortal"*
*"Socrates is a human"*
*"Socrates is mortal"*

where, again, the third statement follows from the other two. Now, the interesting point is that the reason why the third statement follows from the others is apparently the same in both cases. It does not depend on whether we talk about cats eating fish or about humans being mortal or even about objects we don't know having properties we don't know:

*"Every ✌ has property ✿"*
*"❦ is a ✌"*
*"❦ has property ✿"*

Obviously the third statement follows from the other two, no matter what the symbols in these statements are supposed to mean.

The staggering observation that the "follows from" relationship between statements can be established by regarding only their form but disregarding their contents, goes back to the antique Greek philosophers. Expressed in modern terms, there seems to be a syntactic characterization of a relationship which at first sight appears to be semantic in nature. If that's the case, the relationship can also be determined by machines. A program with this capacity is called a **deduction system**.

There are several variations in specifying the precise task of a deduction system. Given some statements called **hypotheses** and some statements called **conclusions**, the task may be: to decide whether the conclusions follow from the hypotheses; to automatically demonstrate that the conclusions follow from the hypotheses if they really do; or, given only hypotheses, to generate new statements that follow from these hypotheses.

In order to achieve such goals, a deduction system requires a series of four constituents, each depending on the former: a logic, a calculus, a state transition system, and a control.

A **logic** is a formal language in which statements can be formulated. It defines syntax and semantics of its formulae, which are the entities of the formal language that correspond to statements. The semantics definitions include a relation $\mathcal{F} \models \mathcal{G}$ ("$\mathcal{F}$ **entails** $\mathcal{G}$" or "$\mathcal{G}$ follows from $\mathcal{F}$" or "$\mathcal{G}$ is a consequence of $\mathcal{F}$"), which formalizes the intuitive relationship between statements in a way that is precise, yet unsuitable to algorithmic treatment.

In this paper we deal with the clausal sublanguage of first-order predicate logic.

The next constituent, a **calculus**, extends a logic by syntactic rules of inference. These rules allow the derivation of formulae from formulae through strict symbol manipulation, without recourse to the semantics. This gives rise to another relation, $\mathcal{F} \vdash \mathcal{G}$, which means that from $\mathcal{F}$ it is possible to derive $\mathcal{G}$ by arbitrarily many successive applications of inference rules of the calculus. Ideally, this syntactic **derivability** relation coincides with the semantic entailment relation. Among the major scientific achievements of this century are the findings that for first-order predicate logic there do exist calculi for which the two relations coincide, and that for more powerful logics, in which the natural numbers can be axiomatized, there don't.

The resolution calculus to be discussed in this paper was developed especially with regard to computer implementations.

To implement a calculus for a logic, one needs a representation of formulae and operations corresponding to the inference rules. Somewhat more abstractly, one has to define a **state transition system**. The states represent (sets of) formulae with their interrelationships, providing information on the development of the derivations up to the respective point and on their possible continuations. The transitions model the changes to the states as inference rules are applied. Better state transition systems for the same calculus can be obtained by refining the states, for instance such that they indicate directly where inference rules can be or have been applied. More common improvements define additional transitions, which are not based on the rules of the calculus, but simplify the formulae or eliminate redundancies in the search space.

The state transition systems described in this paper are based on sets of clauses and on graph structures imposed on them.

Finally, the **control** constituent is in charge of the selection from among the possible transitions and of the administration of the sequences of steps already performed and states thereby produced. In order to choose the most promising transitions, a number of strategies and heuristic criteria can be used.

In this paper we try to focus on the underlying principles of such criteria.

The traditional concern of logicians has been how to symbolically represent knowledge and how to symbolically reason with such knowledge, in other words, they investigated logics and calculi. This does not require a separation of the two constituents as strict as we presented it above. Often, in fact, a calculus is considered a part of the syntax of a logic. In Artificial Intelligence, on the other hand, one is interested in useful and, as far as possible, efficient problem solvers. To that end all four of the constituents have to be investigated, in particular the third and fourth, to which traditional logic did not contribute very much.

In the following sections we shall address all four constituents, presenting techniques to obtain powerful problem solvers based on predicate logic and resolution.

# 2 Logic: Clausal Form of First-Order Predicate Logic

First-order predicate logic is probably the most widely used and most thoroughly studied logic. For this logic the semantic relation of entailment and the syntactic relation of derivability are perfectly balanced, and it is the most expressive logic with this and similar important properties [Lindström 69].

In addition, first-order predicate logic serves as the basis of many other logics. New logics have been developed from it in a variety of ways, for example: relevance logic introduces new junctors; higher-order logics use new quantifiers; temporal, deontic, dynamic, and other modal logics provide a new category of operators different from junctors and quantifiers; fuzzy logic and quantum logic extend the classical set of truth values; default logic and other kinds of non-monotonic logics modify the notion of derivation by new kinds of inference rules. There are many more examples. All of these logics share a substantial fragment with first-order predicate logic. Hence it is useful to be equipped with well-understood methods for this fundamental reference logic. Moreover, the effects of other logics can often be simulated by meta-level components for first-order predicate logic, and presently there even is about to emerge a new discipline investigating the compilation of formulae from other logics using first-order predicate logic as sort of a machine language [Ohlbach 88, 89]. These reasons explain why most of the research on deduction systems has concentrated on first-order predicate logic.

Sometimes one is also interested in restrictions of a logic rather than extensions. The most familiar sublogic of first-order predicate logic is, of course, propositional logic. Other specializations are defined by considering only formulae in a certain normal form. This reduces the number of syntactic forms, often without limiting the expressive power. Clausal logic is a prominent example of that.

## 2.1 Clauses and Clause Sets

A **clause** is a universally closed disjunction of literals. A **literal** is a negated or unnegated atomic formula (**atom**, for short), which in turn consists of a predicate symbol applied to an appropriate number of terms. A term, literal, or clause is called **ground** if it contains no variables. **Unit clauses** are clauses with only one literal.

The meaning of a clause can be defined by specifying an interpretation in the sense of the standard Tarski semantics. Here is an example of a clause:

$$\forall xyz\ \neg Spouse(x, y) \vee \neg Parent(x, z) \vee Parent(y, z) \vee Step\text{-}parent(y, z)$$

Using de Morgan's rule and the definition of the implication junctor, this clause can be transformed into the equivalent formula:

$$\forall xyz\ Spouse(x, y) \wedge Parent(x, z) \Rightarrow Parent(y, z) \vee Step\text{-}parent(y, z)$$

which shows more clearly the intended "natural" interpretation: if $x$ is married to $y$ and parent of $z$, then $y$ is a parent or step-parent of $z$. This syntactic form, which is sometimes called Gentzen form, does without negation sign and uses atoms rather than literals as its elementary parts. Often, the implication sign is reversed (and pronounced "if"), such that the positive literals of the disjunctive form are collected on the left hand side, the negative literals on the right hand side.

Another modification exploits that the formula $\forall x\,(\mathcal{F}(x) \Rightarrow \mathcal{G})$, where the subformula $\mathcal{G}$ contains no free occurrence of the variable symbol $x$, is equivalent to $(\exists x\ \mathcal{F}(x)) \Rightarrow \mathcal{G}$.

Thus the variables occurring only in negative literals can be existentially quantified within the negative part, and our example becomes:

$$\forall yz\ Parent(y, z) \vee Step\text{-}parent(y, z) \Leftarrow \exists x\ Spouse(x, y) \wedge Parent(x, z)$$

This form is most appropriate for a procedural reading of the clause: in order to show that $y$ is a parent or step-parent of $z$, find an $x$ that is married to $y$ and parent of $z$.

Finally, disjunction is associative, commutative and idempotent, which are just the properties of a set constructor. Therefore one can also define a clause as a set of literals:

$$\{\neg Spouse(x, y),\ \neg Parent(x, z),\ Parent(y, z),\ Step\text{-}parent(y, z)\}$$

Here the quantifier prefix is omitted because it is uniquely determined by the set of variables occurring in the clause. This definition abstracts from the irrelevant order of the literals and automatically excludes duplicate occurrences of the same literal in a clause. In an implementation, however, the removal of duplicate literals has to be programmed anyway, and it is not always convenient to place this operation on the abstraction level of the representation of clauses.

Which of these syntactic variants is to be preferred, depends largely on personal habit and taste. Semantically they are all the same. In this paper we adhere to the set syntax, but usually omit the set braces and occasionally do allow duplicate literals.

Clausal logic is the sublanguage of first-order predicate logic consisting of the formulae that are clauses. For this special case some of the standard semantic notions become somewhat simpler. An interpretation **satisfies** a ground clause iff it satisfies some literal of the clause; it satisfies an arbitrary clause iff it satisfies each ground instance of that clause (we only need to consider Herbrand interpretations – see textbooks on classical logic). Obviously each clause is satisfiable, provided that it contains at least one literal. The **empty clause** $\square$, which like the "empty disjunction" corresponds to the truth value *false*, is the only unsatisfiable clause. A clause is valid, i.e., satisfied by all interpretations, iff it contains an atom and its negation. Such a clause is called a **tautology**, the atom and its negation are **complementary literals**.

As usual, a set of formulae is interpreted like the conjunction of its members. Thus an interpretation satisfies a set of clauses iff it satisfies each clause in the set; such an interpretation is also called a **model** of the clause set. A clause set is valid iff each member clause is a tautology. This holds vacuously for the empty clause set, which contains no clauses at all – not even the empty clause. A clause set containing the empty clause, on the other hand, is unsatisfiable. There are some more criteria for special satisfiable or unsatisfiable clause sets, but a general characterization of satisfiability or unsatisfiability based on the syntactic form of the clause set does not (and cannot) exist.

Why should we be interested in such properties? The original problem is, after all, whether some hypotheses $\mathcal{H}_1, \ldots, \mathcal{H}_n$ entail a conclusion $\mathcal{C}$ whether $\mathcal{H}_1, \ldots, \mathcal{H}_n \models \mathcal{C}$ holds. For predicate logic formulae $\mathcal{H}_i$ and $\mathcal{C}$ containing no free variables, this is the case iff the formula $\mathcal{H}_1 \wedge \ldots \wedge \mathcal{H}_n \Rightarrow \mathcal{C}$ is valid, which in turn holds iff the formula $\mathcal{H}_1 \wedge \ldots \wedge \mathcal{H}_n \wedge \neg\mathcal{C}$ is unsatisfiable. (The first iff is known as the **deduction theorem**, the second is straightforward.) As it happens, any formula can be converted into a clause set that is unsatisfiable iff the formula is, and now we have translated our problem into the question whether a certain clause set is unsatisfiable. That's why the unsatisfiability of clause sets is of interest.

## 2.2 Conversion to Clausal Form

Earlier, we used the example that the formula $(\exists x\, \mathcal{F}(x)) \Rightarrow \mathcal{G}$, where the subformula $\mathcal{G}$ contains no free occurrence of the variable symbol $x$, is equivalent to $\forall x\, (\mathcal{F}(x) \Rightarrow \mathcal{G})$. What that really means is that the two formulae entail each other: any interpretation satisfies one of them iff it satisfies the other. In other words, the formulae have the same models. Even stronger, whenever one of them occurs in a larger formula $\mathcal{H}$ and we replace this occurrence by the other, the resulting formula has exactly the same models as $\mathcal{H}$

Thus we can read the pair of formulae as a transformation rule which, expressed procedurally, moves a quantifier from the first subformula of an implication to the front of the entire implication, reversing the quantifier in the process. This transformation rule can be applied to any formulae without affecting their models. There are more model preserving transformations of this kind, and together they allow to move all the quantifiers of a predicate logic formula to the front. The resulting formula is said to be in **prenex form** it consists of a quantifier prefix and a quantifier-free subformula called the **matrix.** In order to convert a predicate logic formula without free variables into clausal form, we start by converting it into prenex form.

As an example consider one of the unfamous "epsilontics" from Analysis, namely the definition that a function $g$ is uniformly continuous:

$$\forall \varepsilon\, (\varepsilon>0 \Rightarrow \exists \delta\, (\delta>0 \wedge \forall xy\, (|x-y|<\delta \Rightarrow |g(x)-g(y)|<\varepsilon)))$$

Using model preserving transformations, we obtain the equivalent prenex form of this formula:

$$\forall \varepsilon\, \exists \delta\, \forall xy \quad \varepsilon>0 \Rightarrow \delta>0 \wedge (|x-y|<\delta \Rightarrow |g(x)-g(y)|<\varepsilon)$$

The next goal in the conversion to clausal form is to eliminate the quantifier prefix. If there are only universal quantifiers, we can simply omit the prefix because it is uniquely determined by the variables occurring in the matrix. If, like in the example above, the prefix contains existential quantifiers, we apply a transformation called **Skolemization**: each existentially quantified variable is replaced by a term composed of a new function symbol whose arguments are all the variables of universal quantifiers preceding the respective existential quantifier in the prefix. In the example above, we replace $\delta$ in the matrix by $f_\delta(\varepsilon)$ for a new function symbol $f_\delta$; the $\exists\delta$ can then be deleted from the prefix.

Skolemization is not a model preserving transformation; applied to a formula $\mathcal{F}$ in prenex form it produces a formula $\mathcal{F}^*$ that is not equivalent to $\mathcal{F}$. However, Skolemization preserves the existence of models: $\mathcal{F}$ has a model iff $\mathcal{F}^*$ has one. In other words, $\mathcal{F}$ is (un)satisfiable iff $\mathcal{F}^*$ is. For more details on Skolemization see textbooks on classical logic.

In any case, after Skolemization there remain only universally quantified variables. Now we can drop the prefix because it is implicitly determined by the matrix. Our example is transformed into the following **quantifier-free form:**

$$\varepsilon > 0 \Rightarrow f_\delta(\varepsilon)>0 \wedge (|x-y|<f_\delta(\varepsilon) \Rightarrow |g(x)-g(y)|<\varepsilon)$$

The remaining conversion uses again model preserving transformations. For instance, subformulae of the form $\mathcal{F}\Rightarrow\mathcal{G}$ are replaced by $\neg\mathcal{F}\vee\mathcal{G}$. With this and similar rules any connectives other than negation, conjunction and disjunction can be eliminated. After that all negation signs are moved inside subformulae by de Morgan's rules. We obtain the

**negation normal form**, which is a formula consisting of literals and arbitrarily nested conjunctions and disjunctions:

$$\neg\varepsilon>0 \vee (f_\delta(\varepsilon)>0 \wedge (\neg|x-y|<f_\delta(\varepsilon) \vee |g(x)-g(y)|<\varepsilon))$$

Finally, the distributivity laws allow the multiplication of this formula into **conjunctive normal form:**

$$\neg\varepsilon>0 \vee f_\delta(\varepsilon)>0) \;\wedge\; (\neg\varepsilon>0 \vee \neg|x-y|<f_\delta(\varepsilon) \vee |g(x)-g(y)|<\varepsilon)$$

This conjunction of clauses can now simply be written as a set of clauses. If we use the set syntax also for each individual clause, we get the following **clausal form** of our example:

$$\{\{\neg\varepsilon>0,\ f_\delta(\varepsilon)>0\},\quad \{\neg\varepsilon>0,\ \neg|x-y|<f_\delta(\varepsilon),\ |g(x)-g(y)|<\varepsilon\}\}$$

In this way, any predicate logic formula without free variables can be converted into a clause set which is (un)satisfiable iff the formula is. If the prenex form of the formula contains no existential quantifiers, the clause set is even equivalent to the formula. In case the formula is a conjunction of subformulae, its clausal form is always the union of the clausal forms of these subformulae. This is especially convenient if we regard our original problem whether the hypotheses $\mathcal{F}_1,\ldots,\mathcal{F}_n$ entail the conclusion $\mathcal{G}$, which was translated into the question whether the formula $\mathcal{F}_1 \wedge \ldots \wedge \mathcal{F}_n \wedge \neg\mathcal{G}$ is unsatisfiable. To convert the latter into a clause set whose unsatisfiability corresponds to the original problem, we can convert each hypothesis individually, convert the negation of the conclusion, and unite all the clause sets thus obtained. In many real examples, each hypothesis corresponds to a single clause anyway.

There are a number of technical improvements that avoid certain redundancies in the conversion. One of them is relevant if the logic supplies an equivalence junctor $\Leftrightarrow$. A formula $\mathcal{F} \Leftrightarrow \mathcal{G}$ has to be transformed such that the equivalence junctor disappears, at latest when converting to negation normal form; if $\mathcal{F}$ and $\mathcal{G}$ contain quantifiers, the transformation is necessary already for the prenex form. $\mathcal{F} \Leftrightarrow \mathcal{G}$ can be replaced by $(\neg\mathcal{F} \vee \mathcal{G}) \wedge (\mathcal{F} \vee \neg\mathcal{G})$, corresponding to $(\mathcal{F} \Rightarrow \mathcal{G}) \wedge (\mathcal{F} \Leftarrow \mathcal{G})$, or alternatively by $(\mathcal{F} \wedge \mathcal{G}) \vee (\neg\mathcal{F} \wedge \neg\mathcal{G})$. Both are model preserving transformations, but the first has a disadvantage if afterwards multiplied into conjunctive form: it results in $(\neg\mathcal{F} \wedge \mathcal{F}) \vee (\mathcal{G} \wedge \mathcal{F}) \vee (\neg\mathcal{F} \wedge \neg\mathcal{G}) \vee (\mathcal{G} \wedge \neg\mathcal{G})$, which is just the second form plus two tautologies containing four additional copies of the subformulae. In general these tautologies cannot be recognized as such if $\mathcal{F}$ and $\mathcal{G}$ are themselves complex formulae which are changed during the conversion. The second form would avoid this redundancy. However, if the whole equivalence occurs within the scope of a negation, disjunctions and conjunctions exchange as the negations are moved to the literals. Then it is the second form which results in redundancies avoided by the other form. Thus, an equivalence in the scope of an even number of (explicit and implicit) negations should be replaced by a conjunction of disjunctions, an equivalence in the scope of an odd number of negations by a disjunction of conjunctions.

Regardless which of the forms is used, the transformation of an equivalence involves a replication of subformulae. The same holds for an application of the distributivity rule in order to multiply into conjunctive normal form. Depending on the nesting of junctors, this may lead to an exponential increase in the size of the formula, which can be limited to a linear increase by a special technique. Consider the formula $\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H})$. An application of the distributivity law would duplicate the subformula $\mathcal{F}$. Instead, we can abbreviate the subformula $(\mathcal{G} \wedge \mathcal{H})$ by $P(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ are the free variables in $(\mathcal{G} \wedge \mathcal{H})$ and $P$ is a new predicate symbol. The original formula $\mathcal{F} \vee (\mathcal{G} \wedge \mathcal{H})$ is then transformed into

$(\mathcal{F} \vee P(x_1, \ldots, x_n)) \wedge (\neg P(x_1, \ldots, x_n) \vee \mathcal{G}) \wedge (\neg P(x_1, \ldots, x_n) \vee \mathcal{H})$, which uses three copies of $P(x_1, \ldots, x_n)$. But this is only an atom and need not be further transformed. The possibly very complex formula $\mathcal{F}$, on the other hand, still appears only once.

Another improvement involves the movements of quantifiers. The prenex form of the formula $\forall x\, ((\forall y\, P(x, y)) \vee (\exists z\, Q(x, z)))$ is $\forall x\, \forall y\, \exists z\, ((P(x, y)) \vee (Q(x, z)))$. Skolemization of the prenex form would replace $z$ by $f(x, y)$. The original formula shows, however, that the existential quantifier does not at all depend on the second universal quantifier. The $\exists z$ came into the scope of $\forall y$ only because the latter happened to be moved into the prefix before the existential quantifier. Thus we can use the smaller Skolem term $f(x)$ for $z$. For this and similar reasons one often converts in a different way: the negation normal form is constructed from the un-Skolemized matrix, then all the quantifiers are moved into subformulae as far as permitted by model preserving transformations. The resulting **anti-prenex form** is then Skolemized.

## 2.3 Specializations and Modifications

Clauses with at most one positive literal are called **Horn clauses**. They are usually written in the implication syntax: $L_1 \wedge \ldots \wedge L_n \Rightarrow L_{n+1}$ or, more frequently, $L_{n+1} \Leftarrow L_1 \wedge \ldots \wedge L_n$ for atoms $L_i$. Sets of Horn clauses have many convenient special properties. In this paper, we shall not go into details here.

While this is just a sublogic, there are some modifications that change the underlying logic. The most common is the incorporation of special symbols with "built-in" fixed interpretations, for example for the equality relation. More recently many-sorted logics have become increasingly popular. Such modifications may allow more concise or more "natural" representations. Much more important, however, is whether they contribute to better derivations. Therefore we address them in a later section (3.3), after derivations have been discussed.

## 3 Calculus: Resolution

A calculus presupposes a logic and provides syntactic operations to derive new formulae of this logic from given ones. The basis for the operations are so-called **rules of inference**, which have the following general form:

$$\frac{\mathcal{F}_1 \ldots \mathcal{F}_n}{\mathcal{F}}$$

The objects above the line are called the **premises** of the inference rule, the object below is its **consequent**. Premises and consequent are formulae or rather schemata of formulae. An application of the rule is possible if the formulae $\mathcal{F}_1, \ldots, \mathcal{F}_n$ are given or have been derived by previous rule applications; the effect of the application is that the consequent formula $\mathcal{F}$ is derived in addition. For obvious reasons we exclude inference rules with infinitely many premises and consider only so-called **finite premise rules.**

Two well-known inference rules are the **modus ponens** rule and the **instantiation** rule:

$$\frac{\mathcal{F} \quad \mathcal{F} \Rightarrow \mathcal{G}}{\mathcal{G}} \qquad\qquad \frac{\forall x\ \mathcal{F}[x]}{\mathcal{F}[t]} \quad \text{for a term } t$$

We demonstrate the application of these rules to the following first-order predicate logic formulae:

$$\forall x\ Cat(x) \Rightarrow Fish\text{-}eater(x)$$
$$Cat(Garfield)$$

The first formula has the form of the premise of the instantiation rule, where $\mathcal{F}[x]$ is $Cat(x) \Rightarrow Fish\text{-}eater(x)$. Taking *Garfield* as the term *t* to be substituted for the variable *x*, we derive the new formula:

$$Cat(Garfield) \Rightarrow Fish\text{-}eater(Garfield)$$

This has the form of the second premise $\mathcal{F} \Rightarrow \mathcal{G}$ of the modus ponens rule. Since the formula corresponding to the first premise $\mathcal{F}$, in this case *Cat(Garfield)*, is also given, we can now derive the formula

$$Fish\text{-}eater(Garfield)$$

with the modus ponens rule.

There may be many different calculi for the same logic. For first-order predicate logic a calculus was designed by David Hilbert. Later, further calculi for this logic were presented by Gerhard Gentzen and by others. In a sense all of these calculi are equivalent, and they are often subsumed under the collective name **classical calculi.** Some people even go as far as talking about "*the* predicate calculus"[1].

Classical calculi are designed such that the formulae following from given formulae can be enumerated by applying the inference rules. There may also be no given formulae at all, in which case the formulae enumerated by the calculus are just the valid formulae of first-order predicate logic. In order to apply its inference rules in this case, a calculus needs some elementary tautologies as a starting point. These are provided by the **logical axioms**, the second ingredient of a calculus beside the inference rules. Hilbert's

[1] AI-ticians seem to be particularly fond of saying that something can or cannot be expressed "in the predicate calculus" when they usually mean "in first-order predicate logic"

calculus, for instance, contains all formulae of the form $\mathcal{F} \Rightarrow (\mathcal{G} \Rightarrow \mathcal{F})$ among its logical axioms, and the modus ponens rule among its inference rules.

Such a calculus for valid formulae is called a **positive calculus**. Dual to that, one can also construct **negative calculi** for unsatisfiable formulae; then the logical axioms provide the elementary contradictions as a starting point.

Independent of the distinction between positive and negative calculi, there is a distinction between the ways the inference rules are to be used [Richter 78]. With a **generating calculus**, one starts from the logical axioms and applies inference rules until the formula to be proven (valid or unsatisfiable, depending on whether the calculus is positive or negative) has been derived. The inference rules of a **testing calculus**, on the other hand, are applied starting from the formula whose validity or unsatisfiability is to be shown, until arriving at logical axioms. Generating calculi can also be called forward calculi or synthesizing calculi, testing calculi can be called backward calculi or analyzing calculi.

A generating calculus can be converted into a testing calculus and vice versa, by simply exchanging the premises and consequents of each inference rule. If we do that for the modus ponens rule, however, it says that given a formula $\mathcal{G}$ we derive the two formulae $\mathcal{F}$ and $\mathcal{F} \Rightarrow \mathcal{G}$ where $\mathcal{F}$ is any arbitrary formula. Such a rule is not very useful. In the original form the rule conforms to the **subformula principle**: given concrete premise formulae, the consequent formula is determined as some subformula of these. As rules are reversed, the subformula principle is often violated. A famous calculus by Gerhard Gentzen, the sequent calculus, was defined as a generating calculus and then turned into a testing calculus: in his "Hauptsatz" Gentzen showed that the only rule whose reversal violates the subformula principle was unnecessary.

## 3.1 The Resolution Rule

Specialized to clausal logic, the modus ponens rule would have the form

$$\frac{L \quad \neg L, M_1, \ldots, M_m}{M_1, \ldots, M_m}$$

for literals $L$, $\neg L$, $M_i$. The **ground resolution rule** is a generalization in that the first premise may be a clause with other literals beside L, which are then also part of the consequent clause (from now on we write the premises below each other):

$$\frac{\begin{array}{c} L, K_1, \ldots, K_n \\ \neg L, M_1, \ldots, M_m \end{array}}{K_1, \ldots, K_n, M_1, \ldots, M_m}$$

The consequent is called a **resolvent** of the premises, which are also called the **parent clauses** of the resolvent. $L$ and $\neg L$ are called **resolution literals**. Thus, from the clauses

*Cat(Garfield), Thinker(Garfield)*
*¬Cat(Garfield), Fish-eater(Garfield)*

where the second is the clausal form of *Cat(Garfield)* $\Rightarrow$ *Fish-eater(Garfield)* used above in the modus ponens example, we can derive the resolvent

*Thinker(Garfield), Fish-eater(Garfield)*

It is easy to see that a resolvent is a consequence of its parent clauses: Suppose there is an interpretation satisfying both parent clauses, we have to show that it satisfies the resolvent as well. To satisfy the parent clause, the interpretation has to satisfy at least one literal each of them. Let us first consider the case that the interpretation satisfies $L$. Then it cannot satisfy $\neg L$ in the second premise clause and hence satisfies one of the $M_i$. This literal belongs to the resolvent, which is therefore also satisfied. In the other case the interpretation does not satisfy $L$ and hence satisfies one of the $K_i$ of the first premise clause and thus also the resolvent.

The essential point of the ground resolution rule is that there must be two complementary literals in the parent clauses. For the non-ground case, the requirement that they are complementary is relaxed such that they need not be, but can be made complementary by substituting terms for their variables. Since x and y are different in the clauses

$$\begin{gathered} Cat(x),\ Thinker(x) \\ \neg Cat(y),\ Fish\text{-}eater(y) \end{gathered}$$

*Cat(x)* and *¬Cat(y)* are not complementary. They can be made complementary by instantiating *x* and *y* with, say, *Garfield,* yielding the same clauses as above. Therefore

*Thinker(Garfield), Fish-eater(Garfield)*

is also a consequence of these more general clauses. Instantiating *x* and *y* with *Garfield* is not the only possibility to obtain complementary literals. The term *friend-of(Odie)* for example would also do, as well as infinitely many others.

Mappings like $\{x \leftarrow Garfield,\ y \leftarrow Garfield\}$ and $\{x \leftarrow friend\text{-}of(Odie),\ y \leftarrow friend\text{-}of(Odie)\}$ are called **substitutions**. They can be applied to terms, atoms etc. resulting in objects that differ only in that each variable appearing to the left of an arrow is replaced by the corresponding term to the right of the arrow. For example, the application of the first substitution to *Cat(x)* results in *Cat(Garfield)*, and so does the application of the substitution to *Cat(y)*. A **unifying substitution** or simply **unifier** for two terms or atoms is a substitution whose application to either of them produces the same result. Thus each of the substitutions above is a unifier for the atoms *Cat(x)* and *Cat(y)*. For these two atoms the effect of any unifier can be obtained by first applying the substitution $\{x \leftarrow y\}$ and then instantiating further. For example, the effect of the unifier $\{x \leftarrow Garfield,\ y \leftarrow Garfield\}$ is the same as that of $\{x \leftarrow y\}$ followed by $\{y \leftarrow Garfield\}$. We call $\{x \leftarrow y\}$ a **most general unifier** for *Cat(x)* and *Cat(y)*. Fortunately, if two terms have a unifier at all, they always have a most general unifier, which is unique up to variable renaming [Robinson 65].

Various **unification algorithms** computing a most general unifier for two terms, term lists, or atoms have been developed. The earliest known stems from 1920. It was discovered by Martin Davis in Emil Post's notebooks. Most of the algorithms are exponential in the size of the terms. Using special representations for terms it is, however, possible to unify two terms in linear time [Paterson & Wegman 78]. The most intuitive version of the unification algorithm views unification as a process of solving equations by a series of transformations: To compute a most general unifier of term lists $(p_1,\ldots,p_k)$ and $(q_1,\ldots,q_k)$ start from the set of equations $\{p_1=q_1,\ \ldots,\ p_k=q_k\}$ and transform the set with the following rules as long as any of them is applicable:

| | | | |
|---|---|---|---|
| $\{x=x\} \cup E$ | $\rightarrow$ | $E$ | (tautology) |
| $\{x=t\} \cup E$ if $x$ occurs in $E$ but not in $t$ | $\rightarrow$ | $\{x=t\} \cup E[x \text{ replaced by } t]$ | (application) |
| $\{t=x\} \cup E$ if $t$ is a non-variable term | $\rightarrow$ | $\{x=t\} \cup E$ | (orientation) |
| $\{f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)\} \cup E$ | $\rightarrow$ | $\{s_1=t_1, \ldots, s_n=t_n\} \cup E$ | (decomposition) |
| $\{f(s_1,\ldots,s_n)=g(t_1,\ldots,t_m)\} \cup E$ | $\rightarrow$ | failure | (clash) |
| $\{x=t\} \cup E$ if $x$ occurs in $t$ | $\rightarrow$ | failure | (cycle) |

where $x$ stands for a variable, $t$ for a term, and $E$ for a set of equations. ♦

**Example**: We want to unify the term lists $(f(x,g(a,y)),\ g(x,h(y)))$ and $(f(h(y),g(y,a)),\ g(z,z))$.
The system of equations

$$\{f(x,g(a,y)) = f(h(y),g(y,a)),\ g(x,h(y)) = g(z,z)\ \}$$

can be transformed into

$$\{\ x = h(y),\ a = y,\ y = a,\ x = z,\ h(y) = z\ \}$$

by applying the decomposition rule several times, then into

$$\{\ x = h(a),\ y = a,\ a = a,\ h(a) = h(a),\ z = h(a)\ \}$$

with the application and orientation rules, and finally into

$$\{\ x = h(a),\ y = a,\ z = h(a)\ \}$$

by using the tautology rule.

No further rule application is possible. Thus, the substitution $\{x \leftarrow h(a),\ y \leftarrow a,\ z \leftarrow h(a)\}$ is a most general unifier for the original term lists. ♦

With the concept of most general unifiers we can now define the full **resolution rule** [Robinson 65]:

| | | |
|---|---|---|
| clause1: | $L, K_1, \ldots, K_n$ | $\sigma$ is the most general unifier |
| clause2: | $\neg L', M_1, \ldots, M_m$ | of $L$ and $L'$. |
| resolvent: | $\sigma K_1, \ldots, \sigma K_n, \sigma M_1, \ldots, \sigma M_m$ | |

The two clauses should not contain the same variables. This can be achieved by automatically replacing the variables in a newly generated clause by completely new variables. The logical justification for the replacement is that formulae $\forall x\ \mathcal{F}[x]$ and $\forall x'\ \mathcal{F}[x']$ are equivalent.

Since any instance of a literal is a consequence of this literal, it is easy to see that the resolvent is a consequence of its parent clauses also for the full resolution rule.

Let us now look at some sample applications of the resolution rule:

$$\begin{array}{ll} Human(Socrates) & \\ \neg Human(x),\ Mortal(x) & \sigma = \{x \leftarrow Socrates\} \\ \hline Mortal(Socrates) & \end{array}$$

$$\begin{array}{ll} P(x,\ a),\ Q(x) & \\ \neg P(f(y),\ y),\ R(y) & \sigma = \{x \leftarrow f(a),\ y \leftarrow a\} \\ \hline Q(f(a)),\ R(a) & \end{array}$$

$$\begin{array}{ll} Shaves(x,\ x),\ Shaves(Barber,\ x) & \sigma = \{x \leftarrow Barber,\ y \leftarrow Barber\} \\ \neg Shaves(Barber,\ y),\ \neg Shaves(y,\ y) & \\ \hline \end{array}$$

*Shaves(Barber, Barber), ¬Shaves(Barber, Barber)*

The last example is the famous Russel antinomy in clausal form: the barber shaves a person if and only if that person does not shave himself. This statement is inconsistent. It also shows that a refinement of the resolution rule remains necessary. The contradiction can be derived only if, before generating the resolvent, in either parent clause the two literals are instantiated by a substitution such that they become equal and merge into one literal:

| | | |
|---|---|---|
| *Shaves(x, x), Shaves(Barber, x)* | $\vdash$ | *Shaves(Barber, Barber)* |
| *¬Shaves(Barber, y), ¬Shaves(y, y)* | $\vdash$ | *¬Shaves(Barber, Barber)* |
| | | $\square$ |

Originally, Robinson integrated this instantiating and merging operation into the resolution rule; for practical reasons, though, it is usually handled as a supplementary inference rule of its own, called **factoring**.

Most classical calculi are positive generating calculi. In contrast, the **resolution calculus** is a negative testing calculus: the empty clause is its only logical axiom and represents the elementary contradiction; resolution and factoring are its rules of inference, which are applied to the clause set whose unsatisfiability is to be shown, until the logical axiom has been reached.

**Example**: We illustrate the whole procedure by proving the transitivity of the set inclusion relation. Our hypothesis is the definition of $\subseteq$ in terms of $\in$:

$$\forall x,y \quad x \subseteq y \;\Leftrightarrow\; \forall w \;\; w \in x \Rightarrow w \in y$$

We want to show that the following conclusion is a consequence of the hypothesis:

$$\forall x,y,z \;\; x \subseteq y \wedge y \subseteq z \;\Rightarrow\; x \subseteq z$$

Transforming the hypothesis and the negated conclusion into clause form we obtain the initial clause set:

| | |
|---|---|
| H1: $\neg x \subseteq y, \neg w \in x, w \in y$ | ($\Rightarrow$ part of the hypothesis) |
| H2: $x \subseteq y, f(x,y) \in x$ | (two $\Leftarrow$ parts of the hypothesis, |
| H3: $x \subseteq y, \neg f(x,y) \in y$ | $f$ is a Skolem function for $w$) |
| C1: $a \subseteq b$ | (three parts of the negated conclusion, |
| C2: $b \subseteq c$ | $a$, $b$, $c$ are Skolem constants for $x$, $y$, $z$) |
| C3: $\neg a \subseteq c$ | |

Resolution derivation (H1,1 denotes the first literal of clause H1 as a resolution literal):

| | |
|---|---|
| H1,1 & C1, $\{x \leftarrow a, y \leftarrow b\}$ | $\rightarrow$ R1: $\neg w \in a, w \in b$ |
| H1,1 & C2, $\{x \leftarrow b, y \leftarrow c\}$ | $\rightarrow$ R2: $\neg w \in b, w \in c$ |
| H2,2 & R1,1, $\{x \leftarrow a, w \leftarrow f(a,y)\}$ | $\rightarrow$ R3: $a \subseteq y, f(a,y) \in b$ |
| H3,2 & R2,2, $\{y \leftarrow c, w \leftarrow f(x,c)\}$ | $\rightarrow$ R4: $x \subseteq c, \neg f(x,c) \in b$ |
| R3,2 & R4,2, $\{x \leftarrow a, y \leftarrow c\}$ | $\rightarrow$ R5: $a \subseteq c, a \subseteq c$ |
| R5 (factoring) | $\rightarrow$ R6: $a \subseteq c$ |
| R6 & C3 | $\rightarrow$ R7: $\square$ |

The derivation of the empty clause means that the initial clause set is unsatisfiable. Therefore the conclusion follows indeed from the hypothesis. ♦

## 3.2 Properties of Resolution and other Types of Calculi

Classical calculi usually have two properties, soundness and completeness. A positive generating calculus is **sound**, if each of its logical axioms is valid and any formula derived by applying inference rules follows from the formulae from which it was derived. Obviously, for the latter property it is sufficient to show that each individual inference rule is sound, i.e., that its consequent follows from its premises. As we have seen earlier, the resolution rule has this property. So has the factoring rule, and thus the resolution calculus is sound.

The requirement for **completeness** of a positive generating calculus is that each formula following from given ones can be derived from the given ones by applying inference rules of the calculus. The resolution calculus is not complete in this sense.

For example, consider the formula $\mathcal{F} = \forall x\, P(x)$, which is just a unit clause. The resolution rule cannot be applied here at all; so even though there are an infinite number of formulae following from $\mathcal{F}$, none can be derived in the resolution calculus.

One consequence of $\mathcal{F}$ would be the formula $P(t)$ for an arbitrary term $t$. In most classical calculi it can be derived with the instantiation rule or a similar rule. But by the same rule any other instances of $P(x)$ would also be derivable. For every variable in a formula, the instantiation rule provides as many alternative derivations as there are terms. This tremendous branching rate is alleviated in the resolution calculus by the idea of unification: variables are instantiated just as far as necessary to apply a rule, and the derivations are carried out at the "most general" level possible.

$\mathcal{F}$ also entails the formula $\forall x\, P(x) \vee Q$. To derive it one needs an inference rule which, in the special case of a clausal form, allows from any clause the derivation of a new one containing arbitrary additional literals. Obviously this violates the subformula principle and thus results in a large search space, which the resolution calculus avoids in the first place.

Further, any tautology follows from $\mathcal{F}$, for instance $Q \vee \neg Q$. In a complete calculus they can all be derived. But it is not at all desirable to derive all formulae that are valid independent of $\mathcal{F}$. After all, an unsatisfiable formula $\mathcal{F}$ would entail every predicate logic formula, but we are not interested in being able to derive them all from $\mathcal{F}$. Derivations of this kind are also ruled out by the resolution calculus.

Thus the resolution calculus is not complete in the sense that all consequences of a formula are derivable from it. But resolution has the property of **refutation completeness**: From an unsatisfiable clause set it is always possible to derive the empty clause, the elementary contradiction, in finitely many steps. Since no interpretation satisfies the empty clause and the calculus is sound, this means that a clause set is unsatisfiable if and only if the empty clause can be derived from it in the resolution calculus.

This property is sufficient to prove all consequences. By the deduction theorem a formula $C$ follows from given formulae $\mathcal{H}_1, \ldots, \mathcal{H}_n$ iff the formula $\mathcal{H}_1 \wedge \ldots \wedge \mathcal{H}_n \Rightarrow C$ is valid. This is the case iff $\mathcal{H}_1 \wedge \ldots \wedge \mathcal{H}_n \wedge \neg C$ is unsatisfiable, which is equivalent to saying that the clausal form of the last formula is unsatisfiable. This in turn holds if and only if the empty clause can be derived from this clause set.

However, the property is not sufficient to *decide* whether a formula is a consequence. One can systematically enumerate all resolvents derivable from the appropriate clause set.

If the empty clause is derivable, it will be obtained after finitely many steps. Otherwise the generation of new resolvents may go on forever. In cases where it is known a-priori that only finitely many different resolvents can be derived from any given clause set, for instance in the ground case, the resolution calculus can of course be the basis of a decision procedure for the special class of formulae.

In summary, the resolution calculus has the following properties:

- The resolution calculus is sound. This implies in particular that whenever the empty clause can be derived from the clausal form of *Hypotheses* $\wedge$ $\neg$*Conclusion*, then the hypotheses do entail the conclusion.
- The resolution calculus is not complete, but refutation complete. Whenever some hypotheses entail a conclusion, it is possible to derive the empty clause from the clausal form of *Hypotheses* $\wedge$ $\neg$*Conclusion*.
- Since first-order predicate logic is not decidable, the resolution calculus can at best be the basis for a semidecision procedure for the problem whether some hypotheses entail a conclusion. Only for certain subclasses of first-order predicate logic can it provide a decision procedure.

The abandonment of classical completeness was a major step in improving the efficiency of a calculus. Of course implementations and thus efficiency were beyond the concern of the time when classical calculi were developed, when the objective was to study whether the semantic entailment relation could in principle be determined by syntactic operations.

Resolution does no longer bother to derive all consequences, but still sufficiently many. The resolution calculus also has the following, less widely known property: for every non-tautologous clause $D$ following from a given clause set, a clause $C$ is derivable that entails $D$. The entailment is of a special, syntactically easy to recognize form: from $C$ one can obtain $D$ by instantiation and addition of further literals. This trivial form of entailment between two clauses is called **subsumption**. For example, from the clauses $\{P(x), Q(x)\}$ and $\{\neg P(f(y))\}$ the clause $D = \{Q(f(a)), R\}$ follows. It cannot be derived, the only possible resolvent is $C = \{Q(f(y))\}$. From $C$ one can obtain $D$ by instantiating $C$ with $\{y \leftarrow a\}$ and by adding the literal $R$, in other words, $C$ subsumes $D$. The resolution calculus can derive any consequences "up to subsumption", and in this sense it is powerful enough to derive all "interesting" consequences.

## 3.3 Modifications

The branching rate in the search space generated by the resolution rule is always finite and in general not too high. Compared to the usually infinite branching rate of classical calculi, this was such a tremendous improvement that in the early days of automated theorem proving many researchers thought the problem was solved once and for all. Very soon, however, the first implementations of resolution theorem provers brought the inevitable disillusionment. Although some really nontrivial theorems could be proved now, there was still no chance of proving routinely everyday mathematical theorems. Various modifications of the basic resolution calculus were then developed in order to strengthen its power by further reducing the search space. All the rest of this paper is dedicated to such improvements on the different levels of deduction systems.

## 3.3.1 Macro Resolutions

The first group of modifications is aimed at overcoming the problem that different sequences of resolution steps may result in the same final resolvent. The idea is to group these steps together into one macro step which generates the resolvent only once.

**Unit resulting resolution** (or **UR-resolution**) simultaneously resolves *n* unit clauses with a clause consisting of *n+1* literals, called a "nucleus". The result is a new unit clause.

For example, let the following clauses be given:

C1: $\neg P(x,y),\ \neg P(y,z),\ P(x,z)$
C2: $P(a,b)$
C3: $P(b,c)$

Then, among others, the following resolution steps are possible:

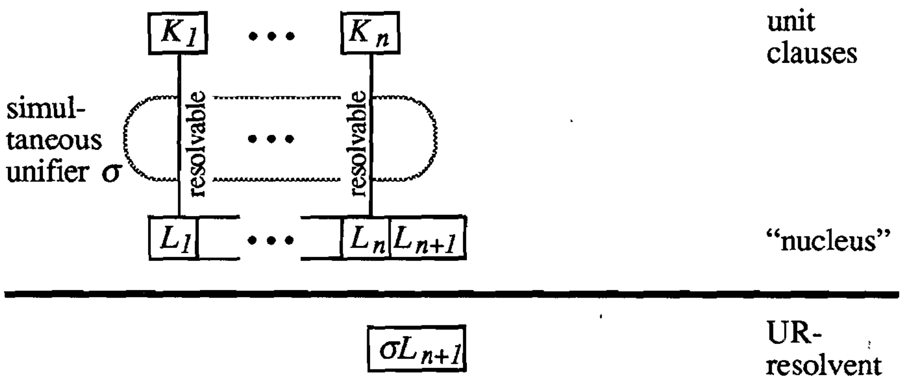
| | | |
|---|---|---|
| C1,1 & C2 | $\vdash$ | R1: $\neg P(b,z),\ P(a,z)$ |
| R1,1 & C3 | $\vdash$ | R2: $P(a,c)$ |

The second resolvent can also be obtained by a another derivation, which differs from the first one only in an insignificant reversal of the order of the steps:

| | | |
|---|---|---|
| C1,2 & C3 | $\vdash$ | R1': $\neg P(x,b),\ P(x,c)$ |
| R1',1 & C2 | $\vdash$ | R2: $P(a,c)$ |

UR-resolution would combine the two steps so that R2 is derived in one go and the order of the steps does no longer matter.

The general schema for UR-resolution can be graphically represented as follows:



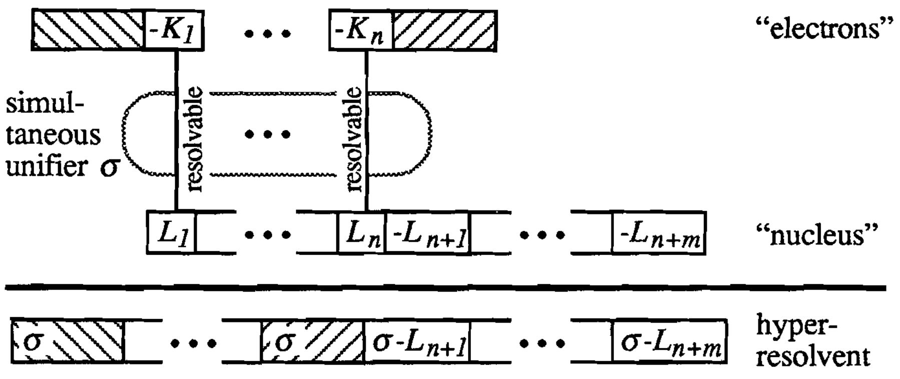
This representation illustrates that all unit clauses have the same status and that the order in which they are used for resolution has no effect on the final result. The simultaneous unifier can be computed from two term lists, one obtained by concatenating the term lists of the unit clauses[1], the other by concatenating the term lists of their partner literals in the nucleus. The two term lists have equal lengths and are unified as usual. In our transitivity example above, the term lists are $(x, y, y, z)$ and $(a, b, b, c)$. Their most general unifier is $\{x \leftarrow a,\ y \leftarrow b,\ z \leftarrow c\}$.

[1] The variables of the unit clauses used more than once have to be renamed.

The abstraction from the order of the $n$ unit resolution steps is not the only effect of UR-resolution. It also omits the $n$-$1$ clauses occurring as intermediate results. In the example above, neither R1 nor R1' would be added to the clause set. On the face of it there is no justification for this omission. Thus UR-resolution actually represents a new rule of inference, for which the same properties as for the resolution rule have to be shown.

UR-resolution is not in general refutation complete, but it is for the unit refutable class of clause sets. Clause sets from this class can be refuted by allowing resolution only when at least one resolution partner is a unit clause. All unsatisfiable Horn clause sets belong to this class. The procedure described in section 4.5 for the extraction of refutation trees essentially simulates a UR-derivation.

**Hyper-resolution** can be regarded as a generalization of UR-resolution. It was developed by John Alan Robinson [Robinson 65b] and is described by the following schema:



Here a clause with at least one positive literal serves as "nucleus". In unsatisfiable clause sets, such clauses always exist. For every positive literal of the nucleus, a so-called "electron" is needed, a clause containing only negative literals. Again, such clauses always exist in an unsatisfiable clause set. The nucleus is resolved with all electrons simultaneously, resulting in a purely negative clause which, in turn, can be used as an electron for the next hyper-resolution step. The purely negative clauses take on the part of the unit clauses in UR-resolution.

Dual to this so-called **negative hyper-resolution**, one can define **positive hyper-resolution** simply by reversing the signs of the literals in the nucleus and the electrons. Since normally a negated conclusion contains only negative literals and can thus be used as electron for negative hyper-resolution, the latter is suitable for backward reasoning from the conclusion toward the hypotheses, whereas positive hyper-resolution can work in the forward direction from the hypotheses toward the conclusion. Both variants of hyper-resolution (with factoring built in) are refutation complete for arbitrary clause sets.

### 3.3.2 Theory Resolution

Resolution is a universal rule of inference. From a theoretical point of view this has the advantage that any first-order predicate logic formula that is provable at all is provable with the resolution calculus. From a practical point of view, however, the disadvantage is that the rule does not know anything about the semantics of the symbols it manipulates. Domain specific knowledge and algorithms can therefore not directly be exploited in a pure resolution theorem prover. Even such simple things like adding two numbers have to be done by resolution with the axioms of number theory. A control component would have to be quite intricate to select the resolution steps in such a way as to simulate an execution of the addition algorithm. While this simulation of an algorithm is possible, it is certainly not the way humans work. Humans tend to apply techniques as specific as possible and resort to general purpose techniques only when they lack specific knowledge. This kind of considerations were the motivation for the ideas presented in the next section.

**Theory resolution** is a scheme to exploit information about the meaning of predicate symbols and function symbols directly within the calculus, by using specially tailored inference rules instead of axioms for these symbols. General theory resolution was proposed by Mark Stickel at SRI [Stickel 85]. Many special cases, however, were known before by different names.

As a motivation for the approach, let us recall the justification for the soundness of the resolution rule:

$$\begin{array}{ll} \text{clause1:} & L, K_1, \ldots, K_n \\ \text{clause2:} & \neg L, M_1, \ldots, M_m \\ \hline \text{resolvent:} & K_1, \ldots, K_n, M_1, \ldots, M_m \end{array}$$

The essential argument for the parent clauses' entailing the resolvent was that an interpretation satisfying the literal $L$ falsifies $\neg L$. The crucial point is that no interpretation can satisfy both $L$ and $\neg L$. This is the case for two literals whenever they meet the purely syntactic condition of being complementary, i.e., if they have opposite signs, equal predicate symbols, and equal term lists.

In many cases one can generalize this syntactic notion of complementarity by utilizing the fact that not any arbitrary interpretations need to be considered, but only certain classes of interpretations. For instance, a set of formulae might contain axioms for a predicate symbol <, such that interpretations can be models only if they associate with < a strict ordering on the universe. Due to the properties of strict ordering relations, no such interpretation can satisfy both $a < b$ and $b < a$. These two literals are not syntactically complementary, but, as it were, semantically contradictory in the assumed context, where the following derivation step would also be sound:

$$\begin{array}{ll} \text{clause1:} & a < b, K \\ \text{clause2:} & b < a, M \\ \hline \text{resolvent:} & K, M \end{array}$$

As a further generalization, we can even abandon the restriction to two parent clauses. No interpretation of the assumed class can satisfy each of the literals $a < b$ and $b < c$ and $c < a$. Analogous to the justification for the simple resolution rule, only with more cases, the following step can also be shown to be sound:

$$\begin{array}{ll} \text{clause1:} & a < b, K \\ \text{clause2:} & b < c, M \\ \text{clause3:} & c < a, N \\ \hline \text{resolvent:} & K, M, N \end{array}$$

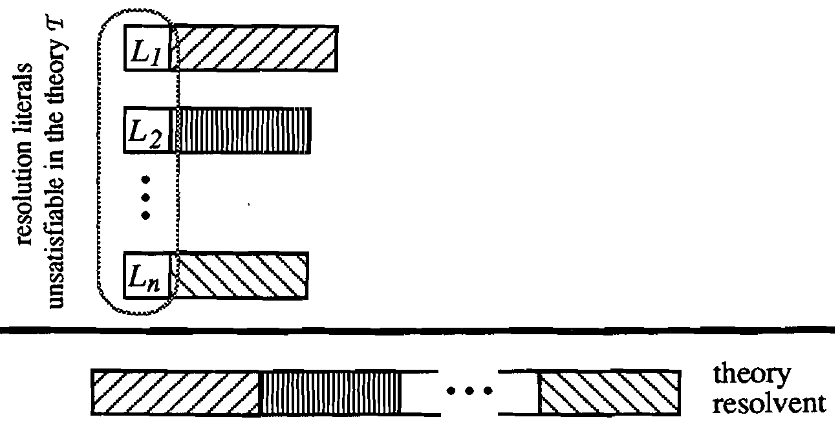
Thus the idea is to proceed from the special case of two syntactically complementary resolution literals to an arbitrary set of resolution literals such that no interpretation of a given class can satisfy all of them. The "class of interpretations" is defined more precisely by the notion of a theory.

In first-order predicate logic, a satisfiable set $\mathcal{A}$ of formulae can be uniquely associated with the class $\mathcal{M}$ of its models, i.e., of the interpretations satisfying all the formulae in $\mathcal{A}$. This class of interpretations in turn uniquely corresponds to a maximal (in general infinite) set $\mathcal{T}$ of formulae that are satisfied by all interpretations in $\mathcal{M}$. The set $\mathcal{T}$ is maximal in the sense that any additional formula would restrict the class $\mathcal{M}$ of models because it would be falsified by at least one model of $\mathcal{A}$. By definition, $\mathcal{T}$ is just the set of consequences of $\mathcal{A}$. From this perspective, $\mathcal{M}$ and $\mathcal{T}$ contain the same information, and both are often called the **theory** of $\mathcal{A}$. Since different sets of formulae may have the same models, any specific $\mathcal{A}$ is just one alternative in defining the theory. $\mathcal{A}$ is also called a **presentation** or **axiomatization** of the theory.

For a given theory $\mathcal{T}$ and a formula $\mathcal{F}$, the **$\mathcal{T}$-models** of $\mathcal{F}$ are simply all those models of $\mathcal{T}$ that are models of $\mathcal{F}$ as well. The notions **$\mathcal{T}$-consequence**, **$\mathcal{T}$-satisfiable**, **$\mathcal{T}$-unsatisfiable**, etc. are then defined correspondingly.

For example, let $\mathcal{A}$ be the set $\{\forall xy\ P(x, y) \Rightarrow P(y, x)\}$, consisting only of the symmetry axiom for $P$. The theory $\mathcal{T}$ results from all interpretations associating with the predicate symbol $P$ a symmetric relation on the universe. The formula $P(a, b) \wedge \neg P(b, a)$ is satisfiable, but not $\mathcal{T}$-satisfiable for this theory. $P(b, a)$ is a $\mathcal{T}$-consequence of $P(a, b)$.

Now the propositional schema for **total theory resolution** is as follows: let $\mathcal{T}$ be a theory and let $C_1, \ldots, C_n$ be clauses, each of them containing a literal $L_i$ such that the conjunction of all these literals is $\mathcal{T}$-unsatisfiable. The union of these n clauses minus the resolution literals $L_i$ constitutes a **$\mathcal{T}$-resolvent.** This clause is a $\mathcal{T}$-consequence of the formula $C_1 \wedge \ldots \wedge C_n$.



For predicate logic, in analogy to the simple resolution rule, the conjunction of the $L_i$ need not be directly $\mathcal{T}$-unsatisfiable. We have to use a substitution $\sigma$, a so-called **$\mathcal{T}$-unifier**, such that the formula $\sigma L_1 \wedge \ldots \wedge \sigma L_n$ is $\mathcal{T}$-unsatisfiable. The $\mathcal{T}$-resolvent is then instantiated with $\sigma$. However, a most general $\mathcal{T}$-unifier for a set of expressions needs no longer be unique (up to variable renaming). Depending on $\mathcal{T}$, there may be one, a finite number, or infinitely many most general $\mathcal{T}$-unifiers independent of each other. Two substitutions are independent if it is not possible to obtain one from the other simply by instantiating variables. In nasty cases there don't even exist most general $\mathcal{T}$-unifiers, but only non-most-general ones.

The theory of the symmetry of $P$ mentioned above is an example of a theory having a finite number of most general $\mathcal{T}$-unifiers. It generates at most two most general unifiers. For

| | |
|---|---|
| clause1: | $P(a, b), Q$ |
| clause2: | $\neg P(x, y), R(x)$ |

the first two literals have the most general $\mathcal{T}$-unifiers $\sigma_1 = \{x \leftarrow a, y \leftarrow b\}$ and $\sigma_2 = \{x \leftarrow b, y \leftarrow a\}$, hence two independent $\mathcal{T}$-resolvents $\{Q, R(a)\}$ and $\{Q, R(b)\}$ can be derived.

The concept of theory resolution allows a much more natural and efficient treatment of frequent specific interpretations of symbols than would the usual axiomatization and normal resolution. The knowledge about the particular theory is essentially encoded in the unification algorithm, which, however, has to be developed for each theory anew. To ensure the refutation completeness of theory resolution, this theory unification algorithm must generate (or, in the infinite case, at least enumerate) all most general unifiers.

The unification algorithm required for an implementation of theory resolution may, for some theories, be too expensive or not even known. This holds in particular when the theory actually consists of several subtheories that are not independent of each other.

As an example consider the theory $\mathcal{T}_\leq$ whose models associate with the predicate symbol $\leq$ a reflexive and transitive relation on the universe and with the predicate symbol $\equiv$ the largest equivalence relation contained in the former relation. Each of these interpretations satisfies an atom $s \equiv t$ for two terms $s$, $t$, if and only if it satisfies both $s \leq t$ and $t \leq s$. Another theory $\mathcal{T}_=$ be such that its models associate with the predicate symbol $\equiv$ the equality relation. In the combination of these two theories, the conjunction of the literals $a \leq b$, $b \leq a$, $P(a)$, $\neg P(b)$ is unsatisfiable, so that these are candidates for resolution literals in a theory resolution step.

However, an appropriate theory unification algorithm would have to be designed for just this combination of theories. As soon as a third theory was added, the algorithm could no longer be used. Therefore it would be more convenient to develop algorithms for the individual theories only and to have available a general mechanism that takes care of the interaction between theories.

Consider the combination of the theories $\mathcal{T}_\leq$ and $\mathcal{T}_=$ above and the clauses:

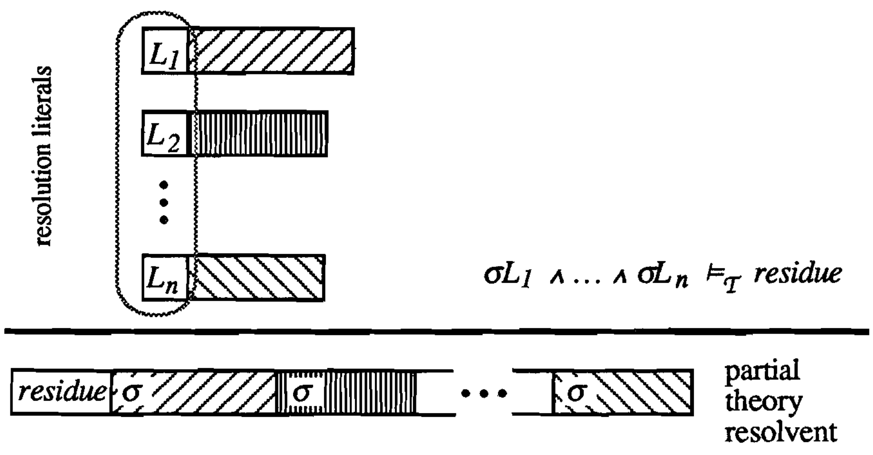
| | | | |
|---|---|---|---|
| clause1: | $a \leq b, K$ | clause3: | $P(a), M$ |
| clause2: | $b \leq a, L$ | clause4: | $\neg P(b), N$ |

from which we ought to be able to derive the resolvent $\{K, L, M, N\}$. We can obtain this clause through a generalized $\mathcal{T}_\leq$-resolution followed by a $\mathcal{T}_=$-resolution. If an interpretation of the theory $\mathcal{T}_\leq$ satisfies $a \leq b$ as well as $b \leq a$, then by construction it satisfies the literal $a \equiv b$ as well. It is easy to verify that the clause $C = \{a \equiv b, K, L\}$ is a $\mathcal{T}_\leq$-consequence of clause1 and clause2. The literals $a \equiv b$, $P(a)$, $\neg P(b)$ can now be recognized by the algorithm for $\mathcal{T}_=$ as resolution literals for an "equality theory resolution step" involving the intermediate clause $C$ and clause3 and clause4, which results in the desired resolvent $\{K, L, M, N\}$.

The first step, producing the intermediate clause $C$, goes beyond theory resolution as presented so far, because the conjunction of the resolution literals is not $\mathcal{T}_\leq$-unsatisfiable and moreover a new literal was added to the resolvent. This so-called **residue** is characterized by the property that in the theory under consideration it follows from the resolution literals. Its predicate symbol does not even need to appear in the parent clauses. If a residue is included, one speaks of **partial theory resolution**, otherwise of **total theory resolution.**

As a residue we may also admit a disjunction of several literals. The empty residue then stands for *false*, and hence follows from the resolution literals only if their conjunction is unsatisfiable in the current theory. This special case corresponds to total theory resolution.

For the most general case, (partial) theory resolution is described by the following schema:



The prerequisite for partial theory resolution to be refutation complete is that the unification algorithm generates not only all most general unifiers but also all "most general residues". More thorough investigations on the completeness for combinations of theories have not yet been carried out, however.

## 3.3.3 Kinds of Theories

The concrete instances of theory resolution look quite different. So far three major classes can be identified which shall be presented by a few but important representatives. The classes can be called algorithmic theories, representation theories, and compiled theories.

### 3.3.3.1 Algorithmic Theories

In most applications special symbols with a very particular meaning occur. The equality predicate symbol is a typical example. In standard predicate logic this particular meaning must be axiomatized with predicate logic axioms. A typical formula set to be presented to a theorem prover now consists of three parts:

*Axiomatization of special symbols*
*Hypotheses*
*Conclusion*

where the axiomatization of special symbols is the same[1] in all problems where these symbols occur. If it is possible to replace these axioms, or at least some of them, by special inference rules, the user is not only relieved from providing them each time again,

[1] Sometimes the axiomatization is not exactly the same but depends on the set of predicate and function symbols occurring in the remaining formulae. This is for example the case for the equality symbol.

but part of the search is replaced by execution of an algorithm, which in general increases the efficiency of the deduction system. We call these theories algorithmic theories because the semantics of these special symbols is implicitly contained in the algorithm implementing the special inference rule.

##### 3.3.3.1.1 Equality Reasoning

The very first symbol for which special inference rules have been developed is the equality predicate. In first-order predicate logic, the equality predicate cannot be formalized by a finite set of axioms. Depending on the symbols appearing in the formula set, the necessary axiom set is obtained by instantiating an axiom schema.

| | | |
|---|---|---|
| $\forall x$ | $x = x$ | (Reflexivity) |
| $\forall x,y$ | $x = y \Rightarrow y = x$ | (Symmetry) |
| $\forall x,y,z$ | $x = y \wedge y = z \Rightarrow x = z$ | (Transitivity) |

For each argument of each function symbol $f$ appearing in the formula set, a substitution axiom of the following form is needed:

$$\forall x_1 \ldots x_n\, y \;\; x_i = y \Rightarrow f(x_1,\ldots,x_i,\ldots,x_n) = f(x_1,\ldots,y,\ldots,x_n) \qquad \text{(Substitution axiom)}$$

For each argument of each predicate symbol $P$, another substitution axiom is needed:

$$\forall x_1 \ldots x_n\, y \;\; x_i = y \wedge P(x_1,\ldots,x_i,\ldots,x_n) \Rightarrow P(x_1,\ldots,y,\ldots,x_n) \qquad \text{(Substitution axiom)}$$

Alan Bundy has investigated the size of the search space generated by these axioms for a relatively simple example [Bundy 83]. The problem is to show that a group in which $x^2 = 1$ for each $x$, is commutative. The axioms for a group with a binary function symbol $\bullet$ (group operation) and a unary function symbol $i$ (inverse) are:

| | | |
|---|---|---|
| $\forall x\,y\,z$ | $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ | (Associativity) |
| $\forall x$ | $1 \bullet x = x$ | (Left identity) |
| $\forall x$ | $x \bullet 1 = x$ | (Right identity) |
| $\forall x$ | $i(x) \bullet x = 1$ | (Left inverse) |
| $\forall x$ | $x \bullet i(x) = 1$ | (Right inverse) |

The additional assumption is

$$\forall x \quad x \bullet x = 1 \qquad \text{(Assumption)}$$

The conclusion is

$$\forall x\,y \quad x \bullet y = y \bullet x$$

One way to prove this conclusion would be

| | |
|---|---|
| $x \bullet y = (1 \bullet x) \bullet y$ | (Left identity) |
| $= ((y \bullet y) \bullet x) \bullet y$ | (Assumption) |
| $= ((y \bullet y) \bullet x) \bullet (y \bullet 1)$ | (Right identity) |
| $= ((y \bullet y) \bullet x) \bullet (y \bullet (x \bullet x))$ | (Assumption) |
| $= (y \bullet ((y \bullet x) \bullet (y \bullet x))) \bullet x$ | (Associativity) |
| $= (y \bullet 1) \bullet x$ | (Assumption) |
| $= y \bullet x$ | (Right identity) |

Using resolution, every transformation step would have to be painstakingly replaced by several resolution steps, and the search space would again be enlarged intolerably because of the several new deduction alternatives. There would be about $10^{21}$ deductions using a level saturation search [Bundy 83]. This method is just not feasible. Numerous calculi and control strategies have been developed in order to overcome this complexity problem (see for example: [Wos et al 67, Robinson & Wos 69, Sibert 69, Morris 69, Knuth & Bendix 70, Brand 75, Shostak 78, Harrison & Rubin 78, Digricoli 79, Huet &

Oppen 80, Lim & Henschen 85, Bläsius 87]). Since a discussion of equality reasoning is beyond the scope of this paper, we here present only the most elementary method as an illustration for partial theory resolution.

The equality axioms actually axiomatize the well known **Leibniz principle** which states that two objects are equal if all their properties are equal. In any context, an object can be replaced by another equal one. Replacing equals by equals as an inference rule, however, is too weak in the presence of variables. Similar to the case of the modus ponens rule, the application of an instantiation rule becomes necessary in order to establish the prerequisites for applying the inference rule.

The idea of unification, introduced by J.A. Robinson [Robinson 65], eliminates the need for arbitrary instantiations and enables goal-oriented instantiation on the most general level. In the same way modus ponens was generalized to resolution using unification, G. Robinson and L. Wos generalized the "replace equals by equals" rule to obtain the **paramodulation rule** [Robinson &Wos 69].

Formally, the paramodulation rule as a partial theory resolution rule is defined as follows: Consider the two clauses

$$C_1 : \{L_1, L_2, \ldots, L_n\}$$
$$C_2 : \{l = r, K_2, \ldots, K_m\}.$$

If $L_1$ contains the subterm $s$, and if $s$ and $l$ can be unified with the most general unifier $\sigma$ (i.e. $\sigma s$ and $\sigma l$ are syntactically identical terms), then the clause

$$\sigma\{L_1', L_2, \ldots, L_n, K_2, \ldots, K_m\}$$

is a **paramodulant** of the clauses $C_1$ and $C_2$, where $L_1'$ is generated from $L_1$ by replacing term $s$ by term $r$. A paramodulation step is really a partial theory resolution step. Its special "algorithmic" part computes the residue $L_1'$.

**Example**: Given the clauses $P(c, h(f(a, y), b)), R(y)$
$f(x, e) = g(x), Q(x),$
the paramodulant of these clauses is $P(c, h(g(a), b)), Q(a), R(e).$
The terms $f(a,y)$ and $f(x,e)$ were unified by $\{x \leftarrow a, y \leftarrow e\}$. This substitution was applied to the new clause. ♦

In two aspects, paramodulation is more general than the principle mentioned above, to "replace equals by equals".

- Paramodulation handles not only unconditional but also conditional equations. In other words, the clause containing the equation can also contain additional literals.
- The two terms used to make a substitution possible do not have to be equal; they just have to be unifiable, i.e. there have to be instances of the participating clauses such that the corresponding terms are equal.

The paramodulation rule is sound: if $S$ is a clause set and $C$ is paramodulant of two clauses in $S$, then any E-models of $S$ (that is, any model of the equality axioms that satisfy $S$) is also an E-model of $S \cup \{C\}$

The resolution calculus, extended to include the paramodulation rule and the **reflexivity axiom** $\{x = x\}$, constitutes a refutation complete calculus (called **RP-calculus**) for predicate logic with equality: for every E-unsatisfiable clause set there is a derivation of the empty clause using the rules and axioms in the calculus. The reflexivity axiom is necessary, because otherwise the empty clause could not be derived from the E-unsatisfiable clause set containing only the clause $\{\neg a = a\}$.

Compared to the explicit application of equality axioms, the search space for the derivation of the empty clause is reduced significantly when the paramodulation rule is included. Many useless resolutions with and between the equality axioms are then no longer possible. But without skillfully controlling paramodulation, the resulting search spaces are still far too large, because this rule can also be applied almost anywhere in the clause set. For the first example in section 1, a breadth-first search requires about $10^{11}$ steps to find the proof to the theorem [Bundy 83].

### 3.3.3.1.2 Theory Unification

Paramodulation brought a considerable improvement compared to equality reasoning with resolution and the equality axioms. Nevertheless the search space is still enormously large and, mostly due to the symmetry of the equality, full of redundancies. For example, formulae like $\forall x\, y\;\; g(x,y) = g(y,x)$ defining the commutativity of certain function symbols are especially troublesome to deal with. The commutativity formula may lead to repeated switching of arguments of the commutative function symbol. For this reason, there were already quite soon attempts to remove such equational formulae from the formula set and replace them by modified deduction rules.

Gordon Plotkin suggested a modification of the resolution rule in such a way that ordinary unification is replaced by a unification procedure that takes the removed equational formulae into consideration. He also determined the condition under which this replacement is allowed to take place [Plotkin 72]. Assuming that the equality predicate appears only in unit clauses, i.e. that the clause set contains a finite number of clauses $\{l_1 = r_1\}, \ldots, \{l_n = r_n\}$, and the equality predicate does not appear in any other clause, unification may be replaced by so-called **theory unification** respecting these equational axioms. This rather strong restriction can be weakened, but doing so here would only complicate the matter further.

**Example**: In section 3 we demonstrated the unification algorithm for the system of equations

$$\{f(x,g(a,y)) = f(h(y),g(y,a)),\; g(x,h(y)) = g(z,z)\}$$

for which we obtained the unifier $\{x \leftarrow h(a),\, z \leftarrow h(a),\, y \leftarrow a\}$. If we use the unification algorithm for the commutativity of $g$, we get $\{x \leftarrow h(y),\, z \leftarrow h(y)\}$, which is obviously even more general than the previous one. The "old" one can be obtained from the new one by substituting $a$ for $y$. ♦

Naturally, we now want to find a most general unifying substitution. In general, however, this cannot be accomplished. There may exist more than one most general unifier. Our commutativity example illustrates this: the equation $\{g(x,y) = g(a,b)\}$ has two independent solutions $\{x \leftarrow a, y \leftarrow b\}$ and $\{x \leftarrow b, y \leftarrow a\}$. The latter reflects that subterms may be exchanged because $g$ is commutative. There is no more general solution, i.e. there cannot be a common more general one, either. Even more problematic cases exist, for example those involving an associative function $f$, where $\forall xyz\, f(x,f(y,z)) = f(f(x,y),z)$. In this case the system of equations $\{f(x,a) = f(a,x)\}$ has an infinite number of independent solutions:

$$\{x \leftarrow a\},\; \{x \leftarrow f(a,a)\},\; \{x \leftarrow f(a,f(a,a))\},\; \{x \leftarrow f(a,f(a,f(a,a)))\},\ldots\,.$$

Since $f$ is associative, all other possible solutions will be equal to one of these solutions; in other words, the term substituted for $x$ differs from one of the terms listed above only in the way it is parenthesized. But even in this awkward case it pays to use theory unification: using resolution or paramodulation without theory unification, terms containing associative function symbols would constantly be reparenthesized.

Let E := $\{\{l_1 = r_1\}, \ldots, \{l_n = r_n\}\}$ be a set of clauses, all of them unnegated equations. On the set of all terms, E induces an equivalence relation $=_E$ which is the smallest equivalence relation containing all term pairs $(l_i, r_i)$ from E and being closed under term construction and instantiation:

if $s_1 =_E t_1, \ldots, s_n =_E t_n$ and $f$ is an n-ary function symbol then $f(s_1, \ldots, s_n) =_E f(t_1, \ldots, t_n)$
if $s =_E t$ and $\sigma$ is a substitution then $\sigma s =_E \sigma t$.

It is possible to show that a pair $(s, t)$ of terms is in the equivalence $=_E$ if and only if the equation $s = t$ follows from axioms in E, i.e. if $s = t$ belongs to the theory defined by E. For simplicity we call the theory just E, too.

**Example**: Let C := $\{g(x,y) = g(y,x)\}$ be the commutativity theory for $g$.
Then $g(a, b) =_C g(b, a)$, and $f(x, g(a, b), z) =_C f(x, g(b, a), z)$. ♦

Given a theory E and a set $\Gamma = \{s_1 = t_1, \ldots, s_n = t_n\}$ of equations, we denote by $U_E(\Gamma)$ or $U_E(s_1 = t_1, \ldots, s_n = t_n)$ the set of all substitutions $\sigma$ with $\sigma s_i =_E \sigma t_i$ for $1 \leq i \leq n$. These substitutions are called **E-unifiers** of $\Gamma$. Assuming that there exists a procedure computing $U_E(\Gamma)$ for arbitrary $\Gamma$, the resolution rule can be modified as follows:

| | | |
|---|---|---|
| clause1: | $P(s_1, \ldots, s_n), K_1, \ldots, K_m$ | |
| clause2: | $\neg P(t_1, \ldots, t_n), L_1, \ldots, L_k$ | $\sigma \in U_E(s_1 = t_1, \ldots, s_n = t_n)$ |
| E-resolvent: | $\sigma K_1, \ldots, \sigma K_m, \sigma L_1, \ldots, \sigma L_k$ | |

In general, a set $U_E(\Gamma)$ is infinite. It is therefore desirable to use only a representative subset $\mu U_E(\Gamma) \subseteq U_E(\Gamma)$, which is as small as possible. In the case of common syntactical unification, which corresponds to unification with respect to the theory with empty axiomatization, we could always use a singleton subset containing the most general unifier. Depending on E, this is not always possible. We need a **minimal** and **complete** set of E-unifiers, which has the following properties:

- $\mu U_E(\Gamma) \subseteq U_E(\Gamma)$ (Soundness)
- For all $\delta \in U_E(\Gamma)$ there exists a $\sigma \in \mu U_E(\Gamma)$ and some substitution $\lambda$ with $\delta x =_E \lambda \sigma x$ (for all $x$ in $\Gamma$) (Completeness)
- For all $\sigma, \tau \in \mu U_E(\Gamma)$: if there is a substitution $\lambda$ with $\tau x =_E \lambda \sigma x$ (for all $x$ in $\Gamma$) then $\sigma = \tau$ (Minimality)

In other words, the members of $\mu U_E(\Gamma)$ must really be E-unifiers of $\Gamma$, each E-unifier must be an instance of a member of $\mu U_E(\Gamma)$, and no two members of $\mu U_E(\Gamma)$ may be instances of each other.

Gordon Plotkin could show that for a refutation complete resolution calculus it suffices to use only $\sigma \in \mu U_E(\Gamma)$ in the E-resolution rule above. Of course this still leaves the problem whether there is an algorithm computing $\mu U_E(\Gamma)$. Unification theory is the field investigating this problem.

Theory unification needs not be restricted to equational theories. Also certain equivalences are suitable for treatment by a unification algorithm. For example, the symmetry of a predicate $P$, namely $\forall xy\ P(x,y) \Leftrightarrow P(y,x)$, can be handled by a unification algorithm that unifies two atoms $P(s,t)$ and $P(s',t')$ by either unifying the arguments directly or with one argument list reversed. The only difference for the resolution rule is that the whole atoms and not only their argument lists have to be submitted to the unification algorithm.

When the equations treated by a theory unification algorithm are the only equations occurring in a clause set, resolution with theory unification suffices to refute it. In case there are still other equations, equality reasoning with theory unification becomes necessary. The following example shows that just substituting theory unification for standard unification in the paramodulation rule is not sufficient to obtain a complete calculus. Suppose the function $f$ is declared associative, i.e. $\forall xyz\ f(x,f(y,z)) = f(f(x,y), z)$ holds and this axiom is replaced by an A-theory unification algorithm. Let the remaining clauses be

$$\begin{array}{ll} \text{A1:} & f(a,b) = f(c,d) \\ \text{A2:} & P(f(a,f(b,e))) \\ \text{A3:} & \neg P(f(c,f(d,e))) \end{array}$$

The atoms $P(f(a,f(b,e)))$ and $P(f(c,f(d,e)))$ are not unifiable, neither with the standard algorithm nor with the A-unification algorithm. Neither $f(a,f(b,e))$ nor $f(b,e)$ nor $f(c,f(d,e))$ nor $f(d,e)$ is unifiable with either side of the equation. Therefore no paramodulation is possible. Nevertheless, if for example $f(a,f(b,e))$ is reparenthesized to $f(f(a,b),e)$, which equals $f(a,f(b,e))$ because of the associativity of $f$, paramodulation is possible with A2 yielding $P(f(f(c,d),e)$ which in turn is A-unifiable with $P(f(c,f(d,e)))$ such that the empty clause can be derived.

Reparenthesizing, however, is not an allowed inference rule. The problem is that sometimes paramodulation into a subterm of an element of a term's $\mathcal{T}$-equivalence class is necessary, where $\mathcal{T}$ is the equational theory handled by the theory unification algorithm. That means that one needs a mechanism to iterate over the equivalence class of terms and to find subterms unifiable with a given side of an equation.

One possibility is to allow paramodulation with functional reflexive axioms. The functional reflexive axiom for $f$ in the example above is $\forall x\ y\ f(x,y) = f(x,y)$. A-unification of $f(x,y)$ and $f(a,f(b,e))$ yields the two most general unifiers $\{x \leftarrow a, y \leftarrow f(b,e)\}$ and $\{x \leftarrow f(a,b), y \leftarrow e\}$. Using the second unifier, paramodulation with A2 yields just the desired reparenthesized literal $P(f(f(a,b),e))$. The main observation we have exploited is that a minimal and complete set of theory unifiers for the terms $t = f(t_1,\ldots,t_n)$ and $f(x_1,\ldots,x_n)$ generates the equivalence class of $t$, and that is just what we wanted.

### 3.3.3.2 Representational Theories

Predicate logic formulae are surely not always the best way to represent information. Facts have to be represented such that inference algorithms have easy access to relevant information. The statements that for example *0* is an *Integer*, all *Integers* are *Reals*, *Socrates* is a *Human*, the function *father-of* maps *Humans* to *Humans* and the function *number-of-children* maps *Humans*×*Humans* to *Integers*, would have to be encoded as follows:

$$
\begin{array}{l}
Integer(0) \\
Human(Socrates) \\
\forall x\, Integer(x) \Rightarrow Real(x) \\
\forall xy\, Human(x) \wedge Human(y) \Rightarrow Integer(number\text{-}of\text{-}children(x,y)) \\
\forall xy\, Human(x) \Rightarrow Human(father\text{-}of(x))
\end{array}
$$

In order to conclude from these formulae that the term *number-of-children(Socrates, Socrates)* denotes a *Real*, three resolutions are necessary. When at the same time thousands of resolutions among other clauses of the current problem are possible, it is not at all obvious that this sequence of three resolutions makes any sense.

In sorted logics this kind of information is represented in an entirely different way. Some properties, such as being an Integer or a Real or a Human, are not encoded as unary predicates but as sorts. Sort information is attached to the other symbols, and the subsort relationship, such as between Integer and Real, is represented in a tree or a graph such that transitivity is automatically built in. The sort of a term needs now no longer be deduced, but can be computed by accessing this information directly. This is done only when the information is really necessary, typically during the unification of a variable with a term.

We call this kind of theories **representational theories**, because formulae are not simply removed and replaced by inference rules, but the information encoded in them is represented in a different syntactic structure. The prototype of this kind of theories is many-sorted logic [Cohn 87, Walther 87, Schmidt-Schauß 89].

Syntactically a many-sorted logic enriches the notion of the **signature**, which in the classical case is just the set of all constant symbols, function symbols, and predicate symbols together with their arities. Now there is another set of primitive symbols called the **sort symbols**. An example for a set of sort symbols is *{Human, Integer}*. In addition, the signature specifies: for each constant symbol a sort (typically in a syntactic form like *Socrates:Human, 0:Integer*); for each function symbol a list of argument sorts and a result sort *(father-of: Human →Human, number-of-children: Human ×Human →Integer* is a typical syntax); for each predicate symbol a list of argument sorts (typical syntax: *Contemporary: Human ×Human, Prime-factor: Integer ×Integer).*

This determines for each ground term whether it is ill-sorted or well-sorted and what its sort is in the latter case. For instance, *father-of(0)* is ill-sorted because the sort of *0* is not the argument sort of *father-of*. The term *father-of(Socrates)* is well-sorted and has the sort *Human*. The term *number-of-children(father-of(Socrates), 0)* is ill-sorted although each of its subterms is well-sorted. Note that these are purely syntactic categories, just as *number-of-children(father-of(Socrates))* is not well-formed because the defined arity of the function symbol is violated. Analogously we define the well-sortedness of ground atoms and other ground formulae. By giving sorts to variable symbols, this extends to first-order predicate logic. For instance, *∀x:Human Contemporary(x, father-of(x))* is well-sorted, *∀x:Human Contemporary(x, number-of-children(x, x))* is not. The restriction of formulae to well-sorted ones prevents the formulation of many meaningless statements by purely syntactic criteria.

To adapt the resolution rule to this simple many-sorted logic, the only change required is a modification of the unification algorithm. It has to ensure that for a variable $x$ of sort $S$ only a term $t$ of the same sort (or of a subsort of $S$) can be substituted. When dealing with more complex sort relationships, this may mean that even though a term $t$ is not of the correct sort, instantiations of $t$ with the correct sort can be found by instantiating variables with other variables of a weaker sort.

As an example, consider the sorts *{Even, Odd, Integer}*, with the subsort relationships $Even \sqsubseteq Integer$ and $Odd \sqsubseteq Integer$. Further, let the signature specify a function symbol $+$: $(Even \times Even \rightarrow Even,\ Odd \times Odd \rightarrow Even,\ Even \times Odd \rightarrow Odd,\ Odd \times Even \rightarrow Odd,\ Integer \times Even \rightarrow Integer, \ldots, Integer \times Integer \rightarrow Integer)$. Unification of the terms *x:Even* and *+(y:Integer, z:Integer)* yields two different solutions, namely $\{x \leftarrow +(y'{:}Even, z'{:}Even), y \leftarrow y'{:}Even, z \leftarrow z'{:}Even\}$ and $\{x \leftarrow +(y'{:}Odd, z'{:}Odd), y \leftarrow y'{:}Odd, z \leftarrow z'{:}Odd\}$. These reflect the fact that the sum of two integers is even iff both summands are even or both are odd. Thus, in this case we have not only one but two most general unifiers, independent of each other.

If there are only a finite number of sorts, the simplest way to find all solutions is to systematically check all combinations to instantiate variables with variables of weaker sorts. Often, however, a clever organization of the search results in more efficient methods.

Altogether, sorted logic has a number of advantages over unsorted logic. Some of them concern questions of the cognitive adequacy of the representation. For instance, the sorted formula $\forall x{:}Even\ \neg Divides(x,3)$ comes closer to saying that no even number divides three than its unsorted counterpart $\forall x\ Even(x) \Rightarrow \neg Divides(x,3)$, which expresses that anything in the world has the property that if it is an even number then it does not divide three. But more important for our purposes are the advantages with respect to the search space. One of them is that sorted representations result in smaller clause sets. For instance, the clause expressing that all even numbers are integers is not present in the sorted representation and has therefore not to be considered as a potential parent clause. Further, the fact that no even number divides three is represented by a two-literal clause in the unsorted case, but by a unit clause in the sorted case, reducing the number of literals to be "resolved away". Finally, among the remaining literals there are fewer resolution possibilities. If we have another unit clause *Divides(3,3)*, we cannot resolve it against the unit clause in the sorted case, because *3* has the sort *Odd* and can therefore not be substituted for *x* of sort *Even*. However, we can resolve it against the corresponding literal in the unsorted case, producing the redundant resolvent $\neg Even(3)$.

Sorted logics are not the only logical formalisms with an alternative representation of certain information. Feature types, for example, are a kind of sort structures where the sorts are not atomic, but contain more complex descriptions of sets [Aït-Kaci & Nasr 86, Aït-Kaci & Smolka 87]. An example for a feature sort is *car[speed:nat, colour = red]* denoting a set of objects (cars) whose *speed* feature is of type *nat* and whose *colour* feature has the value *red*. Feature types and feature unification play an important role in unification grammars [Shieber 86].

Even more complex taxonomic hierarchies and relations can be represented in KL-ONE like knowledge representation systems [Brachman & Schmolze 85]. How theory resolution can be extended to handle such rich sort structures instead of the still rather simple feature types, has not yet been investigated.

### 3.3.3.3 Compiled Theories

For the theories presented so far the algorithms and inference rules have been developed mainly from semantical considerations and not by looking at the corresponding axioms. And in fact, a rule like paramodulation is much easier to understand from the semantics of the equality symbol than from the form of the equality axioms. In certain less complex cases, however, it is possible to take an axiom and straightforwardly translate it into a theory resolution rule. And this translation can even be done automatically. The basic idea, which goes back to [Dixon 73], is as follows: From a clause $C = L_1, L_2$ generate a resolution rule as follows:

$$\begin{array}{ll} K_1, R_1 & K_1 \text{ is resolvable with } L_1 \text{ and } K_2 \text{ with } L_2 \\ K_2, R_2 & \text{with a (most general) simultaneous unifier } \sigma^1. \\ \hline \sigma R_1, \sigma R_2 & \end{array}$$

This rule is sound because the resolvent can also be obtained by two successive resolutions with $C$. The first thing we need for completeness is an additional factoring rule which comprises a resolution and a factoring operation:

$$\begin{array}{ll} K_1, K_2\ R & K_1 \text{ is resolvable with } L_1 \text{ and } K_2 \text{ is unifiable with } L_2 \\ & \text{with a (most general) simultaneous unifier } \sigma^1. \\ \hline \sigma K_2, \sigma R & \end{array}$$

**Examples:** Let C = $\{\neg P(x,y),\ \neg P(y,x)\}$ (asymmetry clause)

A C-resolution is

$$\begin{array}{ll} P(a,z),\ Q(z) & \\ P(b,v),\ R(v) & \sigma = \{z \leftarrow b,\ v \leftarrow a\} \\ \hline Q(b), R(a) & \end{array}$$

A C-factoring is

$$\begin{array}{ll} P(a,z),\ \neg P(b,v),\ S(z,v) & \sigma = \{z \leftarrow b,\ v \leftarrow a\} \\ \hline \neg P(b,a),\ S(b,a) & \end{array}$$

♦

The rule is not complete for recursive (self-resolving) clauses, i.e. clauses which are resolvable or theory resolvable with a renamed copy of themselves. For example, the clauses

C = $\neg R(x),\ R(f(x))$

$R(a)$

$\neg R(f(f(a)))$

are refutable with three successive resolutions. There is, however, no C-resolvent with $R(a)$ and $\neg R(f(f(a)))$ because $x$ with $a$ and $f(x)$ with $f(f(a))$ are not simultaneously unifiable. An extension of the compilation idea to handle at least self-resolving clauses with only two literals is presented in [Ohlbach 90].

The compilation of clauses into resolution and factoring rules is in the same way possible for clauses with more than two literals. In the general case C-theory resolution involves as many resolution partners as the compiled clause contains literals. Again there is the restriction that this clause must not be recursive.

1 The domain of the unifier can of course be restricted to the variables occurring in $K_1$ and $K_2$.

**Example**: Let C = $\{\neg Father(u,v, \neg Father(v,w), Grandfather(u,w)\}$

A C-resolution is

$$\begin{array}{l} Father(x, Tom), P(x) \\ Father(Tom, Jane) \\ \neg Grandfather(Jim, y), Q(y) \quad \sigma = \{x \leftarrow Jim, y \leftarrow Jane\} \\ \hline P(Jim), Q(Jane) \end{array}$$

The algorithm into which C is "compiled" essentially selects two *Father* literals and a *¬Grandfather* literal as potential resolution literals, concatenates their term lists, and unifies the result with the term lists *(u,v,v,w,u,w)*. If there is a unifier, it is restricted to the variables in the selected literals to produce the σ in the resolution step. ♦

The compiled theory resolution rule is a simultaneous n-step resolution. It has the advantage that no intermediate results and therefore in particular no useless intermediate results are produced. It realizes a deeper look-ahead into the search space and is therefore able to cut dead ends earlier than standard resolution.

The modifications of the resolution rule presented in section 3.3 are by no means the only ones. The basic idea, namely to look for complementary subformulae, or at least for subformulae which can be made complementary by instantiation, and to join the rests of the formulae to form a resolvent, has been applied to many other logics: full predicate logic with arbitrary formulae, nonclassical logics etc. To present all these developments would surely require more than one book.

## 4 Logical State Transition Systems

There is a straightforward way to obtain a computer program from a calculus:

- design a representation for sets of formulae, these sets are the possible "states";
- define initial states and final states according to the calculus;
- for each inference rule $\frac{\mathcal{F}_1 \dots \mathcal{F}_n}{\mathcal{F}}$ implement the following transition operation, which can be applied to any state $S$:
  check if $S$ contains each of $\mathcal{F}_1 \dots \mathcal{F}_n$; if so, perform a transition to $S \cup \{\mathcal{F}\}$.

Then one just has to implement an appropriate control regime that uses these operations to transform initial states into final states.

Let's call the above the "**trivial logical state transition system**" for a calculus. A state in this system is a set of formulae, for the resolution calculus a set of clauses. For a generating calculus there would be just one initial state consisting of the logical axioms of the calculus and the hypotheses, while each formula set containing the conclusion would be a final state. For the resolution calculus an initial state consists of the clauses representing the hypotheses and the negated conclusions, while each clause set containing the empty clause is a final state. The transition operation for the resolution calculus goes from a clause set $S$ to the clause set $S \cup \{\mathcal{F}\}$ where $\mathcal{F}$ is a resolvent or factor of members of $S$.

Several problems that are either vacuous or trivial for the trivial logical state transition system, become considerably hard in the context of more sophisticated systems. Their description requires an adequate level of abstraction going beyond traditional notions like completeness. The following conceptual framework, which covers both old and new phenomena, largely relies upon the principles extracted from different problem areas in Artificial Intelligence by Nils Nilsson [Nilsson 80] and upon Gérard Huet's digestion of classical results on the lambda calculus and other systems [Huet 80].

A **state transition system** consists of a set S of states and a binary relation $\rightarrow$ on S termed the **transition relation**. Frequently $\rightarrow$ is the union of some simpler relations conceived as a set of elementary transition rules. There are two distinguished subsets of S, the **initial** and the **final states**. A sequence of states successively related by $\rightarrow$, beginning with S and ending with S', represents a **derivation** of S' from S. As usual, $\xrightarrow{+}$ and $\xrightarrow{*}$ denote the transitive and the reflexive-transitive closure of $\rightarrow$. A state S' is **reachable**, if $S\xrightarrow{*}S'$ holds for some initial state S, and unreachable otherwise. With the appropriate restriction of the transition relation the reachable states define the reachable subsystem of a state transition system.

If the states represent logical formulae and the transitions are based on the inference rules of a calculus, we speak of a **logical state transition system**.

The selection from among the possible transition steps and the administration of the sequence of steps already performed and states thereby produced are subject to a separate constituent named the **control strategy**. Control strategies come under two major classes: when applying a transition rule, **tentative control strategies** make provisions for later reconsideration of alternatives, whereas **irrevocable control strategies** do not. Backtracking and hill-climbing, respectively, are prominent examples of the two types of control strategies. Tentative control strategies essentially require the storage of more than one state at a time, which tends to render them unfeasible for state transition systems with complex states. With an irrevocable strategy just a single state at a time needs to be stored, and the transition can be implemented by destructive modifications of this state.

The following property characterizes a **commutative** state transition system: whenever two transition rules can be applied to some state, each of them remains applicable after application of the other, and the resulting state is independent of the order in which the two steps are performed. The advantage of commutative state transition systems lies in their automatic admittance of irrevocable control strategies, because the choice of an irrelevant rule only delays, but never prevents the "right" steps.

Most logical state transition systems happen to be commutative. For the most famous exception, the lambda calculus [Church 41], a weaker property bearing the name of its investigators Church and Rosser could be shown. Equivalently, a state transition system is **confluent**, if for all states S, $S_1$, $S_2$ with $S \xrightarrow{*} S_1$ and $S \xrightarrow{*} S_2$ there exists a state S' with $S_1 \xrightarrow{*} S'$ and $S_2 \xrightarrow{*} S'$. In other words, any two derivations from the same ancestor state can be continued to a common descendant state. A less restrictive requirement than commutativity, confluence still allows for irrevocable control strategies, especially for **Noetherian** systems where no derivations of infinite length exist.

State transition systems provide a general framework for the coherent description of a wide range of computational systems, such as term rewriting systems [Huet & Oppen 80], semi thue systems [Book 82], various types of automata [Hopcroft & Ullman 79], or logical calculi [Richter 78]. They trace back to the Postian production systems, which in contrast to computationally equipotent formalisms like Turing machines do without inherent control structure.

One advantage of this abstraction lies in the possibility to independently investigate properties of the state transition system and properties of (classes of) control strategies for the state transition system. But it also reflects a shift in paradigm brought about by recent developments in Artificial Intelligence, where a clean distinction between procedural knowledge and control knowledge proved superior to the conventional hierarchical organization of programs.

The basic notions describing qualities of interest for logical state transition systems are soundness and completeness, further confluence and Noetherianness. In a later subsection we shall see that the distinction of some more specific properties is necessary for non-trivial state transition systems.

There are two kinds of potential refinements of the trivial logical state transition system: the structure of the states may be enriched to represent not only formulae but also information as to where rules can be and have been applied; and additional transition rules may be provided, which are not necessarily based on the inference rules of a calculus, but reduce the search space. In the following section we present some refinements of the second kind. After that we deal with improvements based on richer states.

## 4.1 Resolution with Reduction Rules

In the trivial logical state transition system for the resolution calculus each transition rule allows a transition to a superset containing an additional clause. Depending on the system's organization, there might also be rules adding several clauses in one go, for instance all possible UR-resolvents for a given nucleus clause. Let us use the name **deduction rule** for any transition rule that produces a state containing objects not present in the predecessor state.

As deduction rules are applied in the course of a derivation, increasingly larger states are obtained, which tend to contain more and more useless parts. To make deduction systems feasible, it is expedient to also provide **reduction rules**, which allow transitions to smaller states by removing superfluous fragments of the predecessor state.

Many popular reduction rules for resolution are based on logical simplifications. For instance, the **tautology rule** allows a transition from a clause set $\mathcal{S}$ to $\mathcal{S} - \{D\}$ where $D$ is a tautological clause in $\mathcal{S}$. A tautology $D$ is satisfied by all interpretations, thus any interpretation satisfies $\mathcal{S}$ if and only if it satisfies $\mathcal{S} - \{D\}$, hence the two states are logically equivalent.

If a clause set $\mathcal{S}$ contains two clauses $C$ and $D$ such that $C$ entails $D$, then $\mathcal{S}$ and $\mathcal{S} - \{D\}$ are logically equivalent. It is not in general decidable whether or not a clause $C$ entails a clause $D$, but there are simple sufficient criteria: for example, if $D$ is subsumed by $C$. The **subsumption rule** allows a transition from $\mathcal{S}$ to $\mathcal{S} - \{D\}$ where $D$ is subsumed by another member of $\mathcal{S}$[1].

Other reduction rules eliminate "useless" formulae. A typical example is the **purity principle** for the resolution calculus: a clause containing a literal that is not resolvable with any other literal in the clause set, is useless; any resolvent or factor derivable from it would in turn contain such a "pure" literal. Therefore the clause cannot contribute to a derivation of the empty clause and may be removed from the clause set.

It appears natural to define as reduction rules also those rules that eliminate literals from clauses, without removing entire clauses. A simple example for this kind of reduction rule is the **merging rule**, which deletes multiple occurrences of literals from a clause by an explicit operation (rather than hiding the idempotence law in the definition of a clause as a set of literals).

In this spirit we speak of a reduction rule whenever the rule only removes something without adding anything. Reduction rules decrease the number of objects in the current state and thus the number of alternatives from which the next deduction step has to be selected, whereas just the opposite holds for deduction rules. The application of reduction rules alone, quite unlike deduction rules, always terminates after finitely many steps, and intuitively they can never hurt because they reduce the size of the problem at hand ("never" actually depending on certain properties to be discussed later). This simplification effect is the stronger the more reduction rules a system has at its disposal and the more powerful they are. Therefore it is useful to enrich the reduction rule repertoire.

One way to find reduction rules beyond subsumption, tautology removal, and merging, lies in the analysis of the combined effect of sequences of transition steps in special situations and in defining shortcuts simulating this effect. To get a feeling what that means, let us go through an example:

***Example:*** *"The police investigate a theft committed in a hotel. They ascertain that exactly one of the suspects Billy, Lucky, or Jacky is the thief and that none of the*

[1] By this definition each factor is subsumed by its parent clause. It is up to the control constituent to prevent that factoring and subsumption cancel each other's effect. To avoid this phenomenon, one sometimes tightens the definition of subsumption and requires that $C$ must not have more literals than $D$.

*three is able to utter any three sentences without lying at least once. When interrogated, the men make the following statements, which are sufficient for the police to determine who is the thief:*

| | |
|---|---|
| *Lucky:* | *I'm innocent. I haven't even been in the hotel. The man you want is Billy.* |
| *Billy:* | *Nonsense, it wasn't me. Everything Lucky said was a lie. Jacky's innocent too.* |
| *Jacky:* | *You bet I'm innocent. It's not true that Lucky hasn't been in the hotel. But Billy's second statement is a lie."* |

Using the constant symbols $b, l, j$ for the suspects and the predicates $T(x)$ for "$x$ is the thief" and $H(x)$ for "$x$ was in the hotel", the facts can be coded in first-order predicate logic as follows, and the formulae directly convert into the set of ten clauses below (sorted by their lengths):

| | |
|---|---|
| $T(b) \vee T(l) \vee T(j)$ | the thief is one of the suspects |
| $\neg[\,T(b) \wedge T(l) \vee T(b) \wedge T(j) \vee T(l) \wedge T(j)\,]$ | only one of them is the thief |
| $\forall x\, T(x) \Rightarrow H(x)$ | the thief was in the hotel |
| $\neg[\,\neg T(l) \wedge \neg H(l) \wedge T(b)\,]$ | Lucky's statements are not all true |
| $\neg[\,\neg T(b) \wedge (T(l) \wedge H(l) \wedge \neg T(b)) \wedge \neg T(j)\,]$ | Billy's statements are not all true |
| $\neg[\,\neg T(j) \wedge H(l) \wedge \neg(T(l) \wedge H(l) \wedge \neg T(b))\,]$ | Jacky's statements are not all true |

| | | | |
|---|---|---|---|
| C1: | $\neg T(b), \neg T(l)$ | C6: | $T(l), H(l), \neg T(b)$ |
| C2: | $\neg T(b), \neg T(j)$ | C7: | $T(j), \neg H(l), T(l)$ |
| C3: | $\neg T(l), \neg T(j)$ | C8: | $T(j), \neg H(l), H(l)$ |
| C4: | $\neg T(x), H(x)$ | C9: | $T(j), \neg H(l), \neg T(b)$ |
| C5: | $T(b), T(l), T(j)$ | C10: | $T(b), \neg T(l), \neg H(l), T(b), T(j)$ |

We now insert these clauses one by one into the current clause set, starting with the empty set and applying between any two insertions as many reduction rules as possible. Nothing interesting happens during insertion of the first five clauses. Having added to {C1, C2, C3, C4, C5} the clause C6, we notice that a resolution step between C6,1 and C1,2 would result in $\neg T(b), H(l), \neg T(b)$, from which the first literal could then be removed by merging. The remaining clause C6': $H(l), \neg T(b)$ is a proper subset of C6 and would now subsume C6. We simulate the total effect of this resolution-merging-subsumption sequence by simply removing the first literal from C6 and call this reduction rule **subsumption resolution.** After that we add C7 with no further consequences and obtain the current clause set:

| | | | |
|---|---|---|---|
| C1: | $\neg T(b), \neg T(l)$ | C5: | $T(b), T(l), T(j)$ |
| C2: | $\neg T(b), \neg T(j)$ | C6': | $H(l), \neg T(b)$ |
| C3: | $\neg T(l), \neg T(j)$ | C7: | $T(j), \neg H(l), T(l)$ |
| C4: | $\neg T(x), H(x)$ | | |

Being a tautology, the clause C8 disappears right after its insertion, and we proceed with

C9: $T(j), \neg H(l), \neg T(b)$.

Now a resolution between C9,2 and C6',1 would produce the proper subset $T(j), \neg T(b)$ of C9, because the literal $\neg T(b)$ descending from C6' can be merged into the last literal of the resolvent. Again we simply remove the second literal from C9 by the subsumption resolution rule simulating a resolution step followed by merging followed by subsumption. Note that this reduction operation would not be possible if we had not reduced C6 to C6' before. Subsumption resolution using C2 as the partner further removes the first literal from C9, and there remains only C9':$\neg T(b)$. This clause subsumes C1, C2, and C6' and serves to remove the first literal from C5 by subsumption resolution, and after all these reductions the current clause set is

| | | | |
|---|---|---|---|
| C3: | $\neg T(l), \neg T(j)$ | C7: | $T(j), \neg H(l), T(l)$ |
| C4: | $\neg T(x), H(x)$ | C9': | $\neg T(b)$. |
| C5': | $T(l), T(j)$ | | |

The last clause, C10: $T(b), \neg T(l), \neg H(l), T(b), T(j)$ can first be merged, then the removal of the second occurrence of $T(b)$ simulates a resolution with C9' followed by a subsumption. In the same way $T(j)$ can be removed by subsumption resolution with the partner C3, and $\neg H(l)$ with the partner C4 (the literal descending from C4 is the instance $\neg T(l)$ of $\neg T(x)$ and can be merged away). Altogether the last clause becomes C10': $\neg T(l)$, which subsumes C3 and enables a subsumption resolution of C5' to $T(j)$, which finally subsumes C7. We end up with

| | | | |
|---|---|---|---|
| C4: | $\neg T(x), H(x)$ | C9': | $\neg T(b)$ |
| C5'': | $T(j)$ | C10': | $\neg T(l)$. |

This clause set was obtained from the original set {C1, ..., C10} by applying only reduction rules. Incidentally, the last three unit clauses directly tell us who is or is not the thief, which in the original clause set was far from obvious. The reduction rules happened to solve the problem before we even asked for a solution by adding a clause corresponding to a negated conclusion. If we insert C11: $\neg T(j)$ as such a clause, we can use the partner C5'' to remove the first (and only) literal from C11 by subsumption resolution. Thus we derive the empty clause from the original problem using only reduction rules, without ever performing a proper deduction step adding a new clause. A control component capable of selecting from among several applicable deduction rules would never have to be activated for this example. ♦

Any step performed in the example can be explained in terms of resolution, merging, subsumption, and tautology removal, therefore the last clause set is logically equivalent to the original. Even stronger, any refutations using clauses from the original set can be transformed into refutations using clauses from the final set instead, such that the complexity of the transformed refutations (measured, for instance, in terms of the rm-size [Kowalski & Kuehner 71], which essentially counts the number of applications of deduction rules) remains the same as before or even improves. This is a property all reduction rules ought to guarantee.

More precisely, if a reduction rule removes literals from clauses, it obviously neither destroys the refutability nor increases the complexity of possible refutations. However, it might turn a non-refutable clause set into a refutable one, thus we need sound justifications for the literal removals. In the case of merging, soundness is trivial. Reduction rules removing entire clauses from clause sets do not cause a soundness problem, but we have to make sure that they preserve the refutability and do not increase the complexity of refutations. For subsumption and tautology removal these are well-established properties, see [Chang & Lee 73, Loveland 78].

There is hardly any bound to the ingenuity with which the developer of a deduction system may design such reduction rules. Whether they can actually be used in a particular situation, however, depends in general not only on a formula's logical status but also on the overall state of the search procedure in that situation. The search algorithm might be able to succeed with the unreduced set of formulae, but might fail with the reduced one. We shall discuss this problem when presenting reduction rules for clause graphs.

It goes without saying that reduction rules like subsumption resolution are only useful if there is a reasonably efficient way to recognize situations in which they can be applied. All the rules presented here have been implemented in an automated deduction system named Markgraf Karl [Ohlbach & Siekmann 89], which is based on Kowalski's connection graph proof procedure [Kowalski 75]. Its underlying clause graph structure, which will be presented in the next section turned out to be a good basis to detect the applicability of many reduction rules.

## 4.2 Clause Graphs and Transition Rules

We now turn to a more complex type of states for logical state transition system based on the resolution calculus, and to transition rules exploiting the richer structure. The idea to use a graph of clauses instead of a set of clauses goes back to Robert Kowalski and his **connection graph proof procedure** [Kowalski 75]. While this is an approach for standard resolution, we present a slightly more generalized version covering total theory resolution. However, we restrict ourselves to theories for which there always exist finitely many independent most general unifiers.

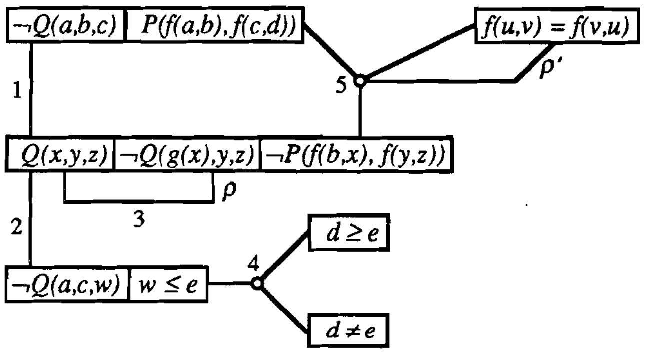
### 4.2.1 Clause Graphs

A clause graph is based on a set of nodes which are labelled with literals. These **literal nodes** are grouped together to **clause nodes**, which represent sets of literals (i.e., clauses). Usually, literal nodes are graphically depicted as little boxes in which the labelling literals are written, clause nodes as contiguous clusters of such boxes.

Arbitrary relations between literals can now be represented by links between literal nodes. The most important relation, the resolvability relation, is represented by so-called **R-links**. They connect the resolution literals that can participate in a (theory) resolution step. The R-links themselves are often marked with the most general unifiers for the atoms in the incident boxes.

The reason for the distinction between literals and literal nodes and between clauses and clause nodes is purely technical. Different nodes may very well be labelled with the same literal but be linked to entirely different places. If the literals themselves were regarded as the nodes of the graph, one couldn't even formulate a phenomenon like this. However, in the sequel we will not strictly distinguish between nodes and formulae, as long as there will be no confusion.

**Example:**



This clause graph contains six clause nodes. R-link 1 connects two resolution literals for a simple resolution step with most general unifier $\{x \leftarrow a, y \leftarrow b, z \leftarrow c\}$. Performing this step would produce the resolvent $\{P(f(a,b),f(c,d)), \neg Q(g(a),b,c), \neg P(f(b,a),$

$f(b,c))\}$. R-link 2 also represents a simple resolution step, its unifier is $\{x \leftarrow a, y \leftarrow c, z \leftarrow w\}$.

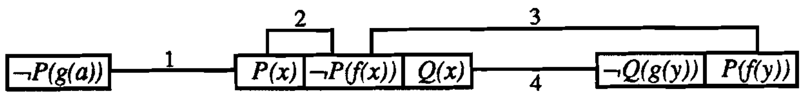
R-link 3 connects two literals within the same clause. They have equal predicate symbols and opposite signs, but their term lists are not directly unifiable. This R-link indicates a possible resolution step using a copy of the clause, $\{Q(x',y',z'), \neg Q(g(x'),y',z'), \neg P(f(b,x'),f(y',z'))\}$, in which the variables have been renamed by the substitution $\rho = \{x \leftarrow x', y \leftarrow y', z \leftarrow z'\}$. The first literal in the original clause is now resolvable with the second literal in the copy, using the unifier $\{x \leftarrow g(x'), y \leftarrow y', z \leftarrow z'\}$ and generating the resolvent $\{Q(x',y',z'), \neg P(f(b,x'),f(y',z')), \neg Q(g(g(x')),y',z'), \neg P(f(b,g(x')),f(y',z')\}$. (The analogous step using the first literal of the copy and the second of the original would result in a resolvent in which the primed and unprimed variables are simply exchanged; therefore one of the variants suffices.) In actual implementations such a **self-resolution** of a clause with a copy of itself is almost always omitted. For theory resolution it is necessary, however.

R-link 4 is a proper theory-R-link. In the theory of ordering relations and equality, the conjunction of the literals $w \leq e$, $d \geq e$, $d \neq e$ becomes contradictory when instantiated with the substitution $\{w \leftarrow d\}$. Thus the theory resolvent $\{\neg Q(a,c,d)\}$ can be derived. Finally, R-link 5 involves two different variants of the commutativity clause, $\{f(u,v) = f(v,u)\}$ and $\{f(u',v') = f(v',u')\}$. This link is marked with two most general unifiers: $\{x \leftarrow a, y \leftarrow c, z \leftarrow d, u \leftarrow a, v \leftarrow b\}$, and $\{x \leftarrow a, y \leftarrow d, z \leftarrow c, u \leftarrow a, v \leftarrow b, u' \leftarrow c, v' \leftarrow d\}$. The first unifier corresponds to applying the commutativity law to the subterm $f(a,b)$ before unifying it with $f(b,x)$, whereas $f(c,d)$ and $f(y,z)$ are unified without prior swapping. The second unifier uses commutativity a second time to also switch the arguments in $f(c,d)$. ♦

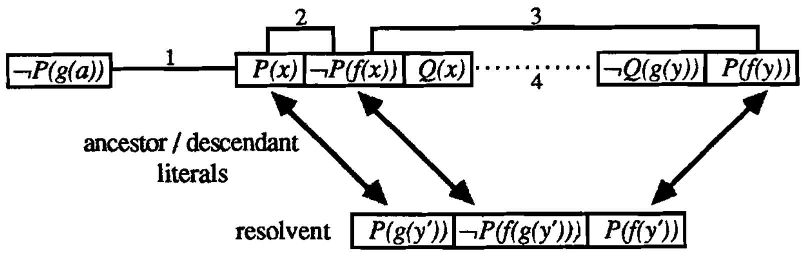
### 4.2.2 Deduction Rules for Clause Graphs

A naive transfer of the resolution rule to clause graphs is as follows: generate, in the usual way, the resolvent indicated by an R-link, create the clause node representing it, and compute the new R-links by examining all resolution possibilities between the new literals and those present before this step. The latter operation is very expensive, but it can be considerably simplified: the literals of the resolvent are instances of literals appearing in the parent clauses, their **ancestor literals**. There cannot be a resolution possibility with a new literal unless there already is a corresponding resolution possibility with its ancestor literal. Thus it suffices for each new literal to examine the resolution possibilities with literals connected to its ancestor by R-links. The new R-links can be obtained from the old ones by **inheritance**.

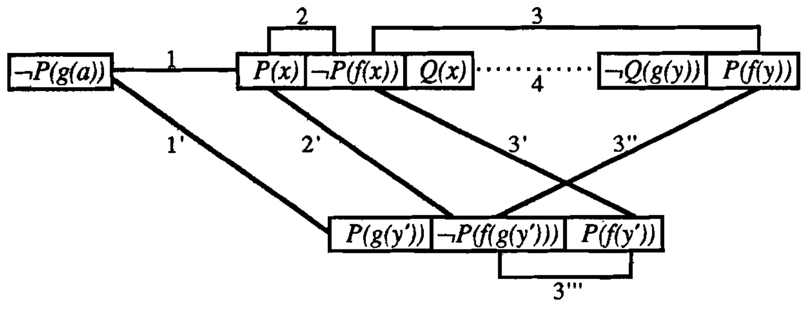
**Example**: Inheritance of R-links



In this initial clause graph let us resolve "on" R-link 4, resulting in the following intermediate situation:



The new clause graph is obtained by inheritance of the old links 1, 2, 3:

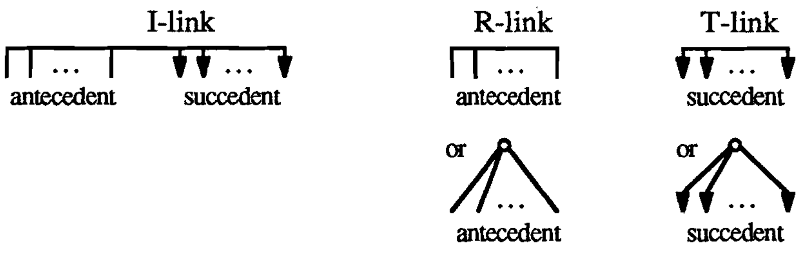


♦

Thus there is an advantage of the clause graph representation: the R-links provide an excellent indexing to compute the resolution possibilities between a resolvent and the old clauses. For a comparatively large class of theories it is even possible to compute the unifiers of the new R-links directly from the unifiers of the old links, without having to unify any literals (see [Ohlbach 87]).

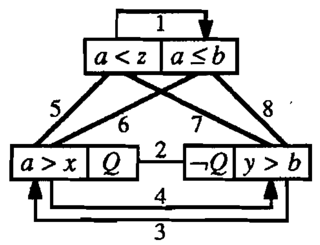
The R-links described so far are just special cases of a more general type of links. As a motivation let us consider the logical meaning of an R-link. If there is an R-link for a theory $\mathcal{T}$ connecting literals $L_1, \ldots, L_n$ and marked by a unifier $\sigma$, then the conjunction of the literals $\sigma L_1, \ldots, \sigma L_n$ must be $\mathcal{T}$-unsatisfiable. This in turn is the case if and only if the formula $\sigma L_1 \wedge \ldots \wedge \sigma L_n \Rightarrow \square$ is $\mathcal{T}$-valid.

Now the generalization suggests itself to connect two groups of literals, the **antecedent** $L_1, \ldots, L_n$ and the **succedent** $K_1, \ldots, K_m$, by a so-called implication link (or simply **I-link**, for short), whenever the formula $\sigma L_1 \wedge \ldots \wedge \sigma L_n \Rightarrow \sigma K_1 \vee \ldots \vee \sigma K_m$ is $\mathcal{T}$-valid. For an empty succedent we get just the special case of an R-link, indicating a resolution possibility. The other special case, with an empty antecedent, signifies that the formula $\sigma K_1 \vee \ldots \vee \sigma K_m$ is $\mathcal{T}$-valid and thus indicates a tautology clause. Links of this type are called **T-links**. Graphically, we depict the different links as follows:

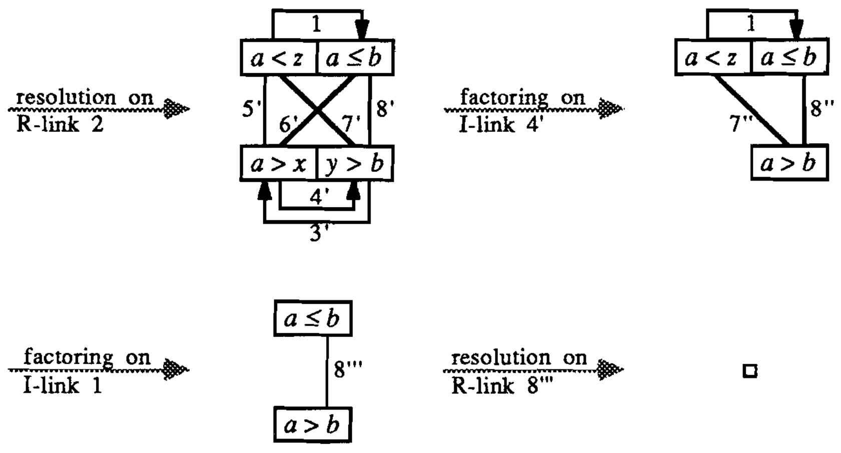


A general I-link corresponds to a partial theory resolution step in which the instance $\sigma K_1 \vee \ldots \vee \sigma K_m$ of the succedent is the residue. This interpretation of the link types requires the antecedent literals (joined conjunctively) to be parts of different clauses, and the succedent literals (joined disjunctively) to be parts of the same clause. Several antecedent literals within the same clause are taken to mean that they belong to different copies of this clause. If succedent literals are scattered over several clauses, the I-link represents no executable operation. However, other steps may cause instances of these succedent literals to become part of the same resolvent, so that inheritance creates an executable I-link. If antecedent and succedent literals belong to the same clause, it is possible to derive a new clause by removing the antecedent literals and instantiating the remainder. This corresponds to an oriented factoring operation.

**Example:** Execution and inheritance of I-links



$1: z \leftarrow b$
$3: x \leftarrow b, y \leftarrow a$
$4: x \leftarrow b, y \leftarrow a$
$5: z \leftarrow x$
$6: x \leftarrow b$
$7: y \leftarrow a, z \leftarrow b$
$8: y \leftarrow a$



I-link 1 is a proper theory-I-link for the theory of ordering relations. It represents the validity of the implication $a<b \Rightarrow a\le b$ in this theory. The I-links 3 and 4, on the other hand, simply denote the propositional equivalence $a>b \Leftrightarrow a>b$, and are thus somewhat redundant. But the representation of the equivalence by two implications allows more flexibility for factoring, as we shall see in the second step.

The first step is to resolve on R-link 2. The resolvent and the inherited links are shown in the second diagram, but the parent clauses are left out just to save space; they and their links still belong to the graph. The new I-links 3' and 4' are generated by inheritance of I-links 3 and 4.

In the next step we derive the factor $a>b$ using I-link 4' (showing, again, only the interesting subgraph in the diagram). The same factor could also be derived with I-link 3'; however, if the literals $a>x$ and $y>b$ in the parent clause had links to different places in the graph, the resulting graphs would be different, because the factor's literals and links always descend from the non-antecedent literals. If reduction rules as discussed in the next section are used, it is indeed possible that literal nodes labelled with equal literals have links to different places.

The last two operations are self-explanatory. ♦

Now we can already obtain a logical state transition system for the resolution calculus using clause graphs as states rather than clause sets. Given a clause set whose unsatisfiability is to be shown, we construct the initial clause graph examining all resolution possibilities and computing the links. The deduction rules above can be seen as operating "on" links, enabling transitions to supergraphs. These rules can be applied until a clause graph containing the empty clause has been derived from the initial state, which proves the unsatisfiability of the initial clause set.

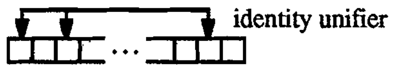
This logical state transition system does note differ very much from the trivial one. Its disadvantages are the overhead for the computation of the initial state and the increased cost of handling the more complex states. In return there is an advantage. In classical procedures the order in which resolution steps take place is more or less fixed by the search algorithm. The explicit representation of resolution possibilities by R-links in clause graphs, on the other hand, allows the assessment of all R-links prior to execution and a selection of the best alternative by heuristic criteria.

### 4.2.3 General Reduction Rules for Clause Graphs

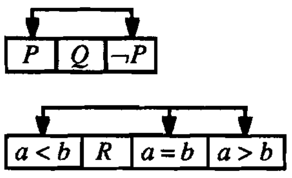
The clause graph data structure lends itself easily to an efficient implementation of the reduction rules mentioned at the beginning of section 4.1. We now present the clause graph versions of the most important of the reduction rules that are generally applicable to resolution based systems: tautology removal, subsumption, and literal removals.

A **tautology clause** is a disjunction of literals that is valid (in the given theory). It is indicated by a T-link with "empty unifier", the identity substitution.

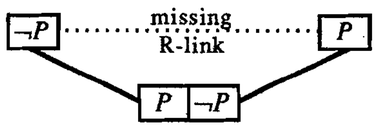
General schema for tautology recognition:



Examples:



From a logical point of view, tautology clauses are useless when searching for a contradiction and should therefore be removable from the clause set. In a more complex search procedure, however, it is not just the logical status of a formula that counts, but also its context in the derivation process. For systems using clause graphs this context is determined, among others, by the presence or absence of links. Link removal rules as described in the next section can lead to situations where resolvable literals are *not* connected by an R-link. As a consequence it may happen that a tautology is in fact necessary for the derivability of the empty clause, as in the following graph:

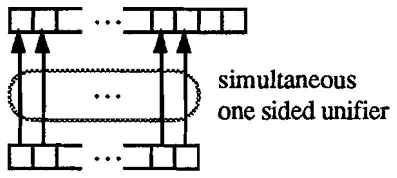


Resolution on the first R-link and subsequent resolution on the successor of the second R-link produces the empty clause. The removal of the tautology clause would result in a clause graph with two complementary unit clauses but no links; the empty clause could no longer be derived.

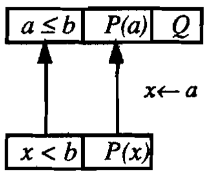
The so called **bridge link condition** [Bibel 81] guarantees that a tautology clause is indeed superfluous: if $L_1 \vee \ldots \vee L_n$ is a tautology in theory $\mathcal{T}$, then the formula $\neg L_1 \wedge \ldots \wedge \neg L_n$ is $\mathcal{T}$-unsatisfiable. That means that any n-tuple of literals that are reachable from the tautology via simple R-links (without a theory), can be the parent literals of a $\mathcal{T}$-resolution step and should therefore be connected by an R-link. If that is the case, the tautology may be removed, otherwise not. Of course one can reinsert missing links to make the tautology removable.

The second important reduction rule, **subsumption**, is a special form of entailment between clauses that is syntactically easy to recognize. The original definition (without theories) is: a clause $C$ subsumes a clause $D$ if there is a substitution $\sigma$ such that $\sigma C \subseteq D$ holds. With theories the definition can be slightly generalized: when testing for $\sigma C \subseteq D$, the literals are not only tested for syntactic equality but also for implication in the given theory. In a clause graph such implications are indicated by I-links.

General schema for subsumption recognition:



Example:



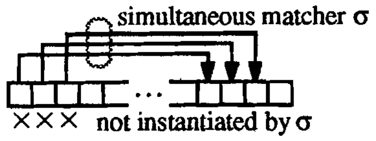
Here, the bottom clause subsumes the clause on top. The subsumed clause, that is the longer one, can usually be removed. However, factors are always subsumed by their parent clause, without being superfluous in general. The application of subsumption again requires the consideration of the clauses' context. In systems using clause graphs, there is another condition on the links: a subsumed clause may be removed only if each R-link at a literal in the subsumer has a counterpart at the corresponding literal in the subsumed clause [Bibel 81].

Another class of reduction rules modifies single clauses by **literal removals.** Literals may be removed from a clause in a clause set whenever the clause set with shortened clause is logically equivalent to the original. In contrast to the removal of whole clauses, literal removals do not require consideration of existing or nonexisting links in a clause graph.

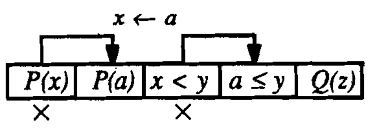
A trivial case is literal **merging**: one of two syntactically identical literals in a clause may be removed. This application of the idempotence law for disjunction is in a sense automatically built into the formal definition of a clause as a set of literals; in an actual program it has to be implemented anyway. The above-mentioned view of a clause node as a set of literal nodes that may be labelled with equal literals but may have links to different places, also suggests an explicit merging operation.

Somewhat more general is **subsumption factoring.** If a clause can be split into two disjoint subsets $C$ and $D$ such that $C$ subsumes $D$ with a substitution $\sigma$ that does not have any effect on $D$, then all literals in the subset $C$ may be removed. This time it is not the subsumed part, but the subsuming part that is not needed, and we're left with $D$. The rule has its name from the fact that $D$ is a factor of the original clause $C \cup D$, subsuming its own parent clause. The removal of the $C$ part simulates a sequence of factoring and subsumption operations. Again, the subsumption test can be modified to test for the implication in a theory instead of syntactic equality.

General schema for subsumption factoring:



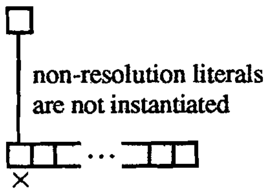
Example:
$D = \{P(a),\ a \leq y,\ Q(z)\}$



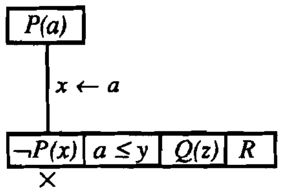
Literals marked by × may be removed from the clause.

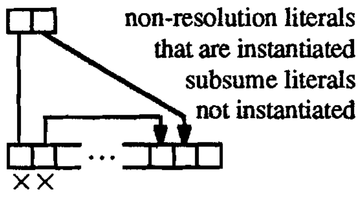
A further generalization is **subsumption resolution**. It covers all cases in which a proper subset $D$ is derivable from a clause by a sequence of resolution and factoring operations, without instantiating the remaining literals. In this case, $D$ subsumes the parent clause and all intermediate clauses, so that technically the operation can be performed simply by removing all literals not belonging to $D$.

Some schemas for subsumption resolution:

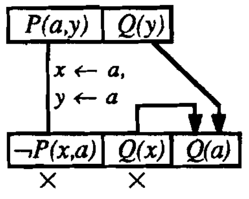


Examples:
$D = \{a \leq y,\ Q(z),\ R\}$




non-resolution literals that are instantiated subsume literals not instantiated
···
××

$D = \{Q(a)\}$



In principle the power of literal removal rules can be pushed as far as one desires; in the extreme case, up to the point where all literals may be removed from a clause (which thus becomes empty) because the whole clause set is unsatisfiable. Of course this would require criteria as powerful as the whole proof procedure itself, and thus would only shift the overall problem. But by cleverly exploiting the link structure and the substitutions in a clause graph, quite a lot of situations in which literal removals are applicable can be recognized efficiently.

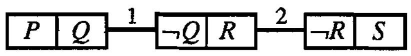
As a further advantage of the clause graph representation we note that the links support a great number of algorithms for the recognition of redundancies in the clause set.

### 4.2.4 Specific Reduction Rules for Clause Graphs

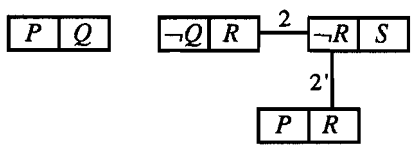
The form of the clause graphs and of the operations on clause graphs as presented so far can be seen as an implementation-oriented rendering of the resolution calculus. In this section we study further operations that enable the removal of links or clauses. They block certain derivation alternatives that would be possible with the trivial logical state transition system.

The first idea is to remove R-links and I-links once the corresponding derivation step has been executed, in order to prevent a repetition of the same step. This seemingly harmless administrative measure has a tremendous effect when combined with link inheritance: the removal of a link disables the creation of any links that could potentially be inherited from it, and thus blocks later generations of resolution steps.

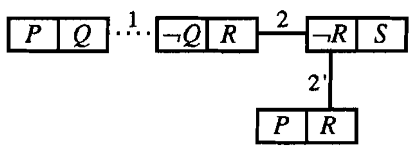
**Example**: We compare two derivations from the following initial clause graph. On the left hand side link removal is applied, on the right hand side the links operated upon are drawn as dotted lines.



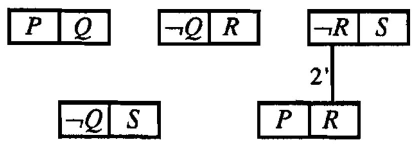
Resolution on link 1 with removal



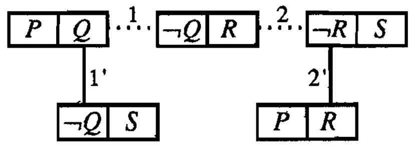
without removal



Resolution on link 2 with removal



without removal

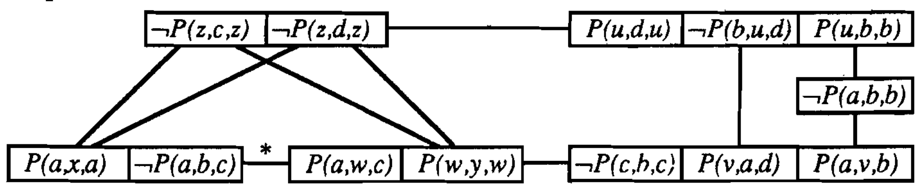


In the graph to the left only one resolution possibility is represented, whereas there are two in the graph to the right, both leading to the same resolvent, though. ♦

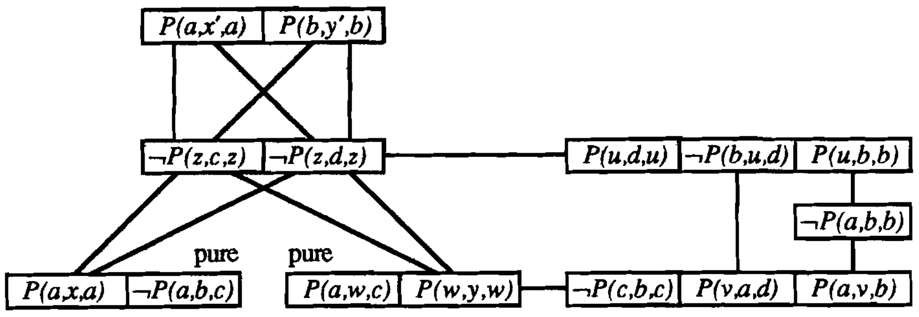
In general, link removal prevents multiple derivations of resolvents from the same clauses through different orders of the resolution steps.

The second specific reduction rule allows the removal of clauses that contain a "pure" literal without any links. This **purity rule** is based on the observation that a derivation of the empty clause requires that all literals of a clause involved in the refutation must eventually be "resolved away". A literal node without links cannot be resolved away. If a clause contains a purity, so does any resolvent derived from it or its descendants, hence the empty clause cannot be among the clauses derivable from the pure one. The removal of a pure clause of course implies the removal of all its links, which can result in new purities in neighbouring clauses. Thus, one application of this rule can cause a chain reaction of further reductions.

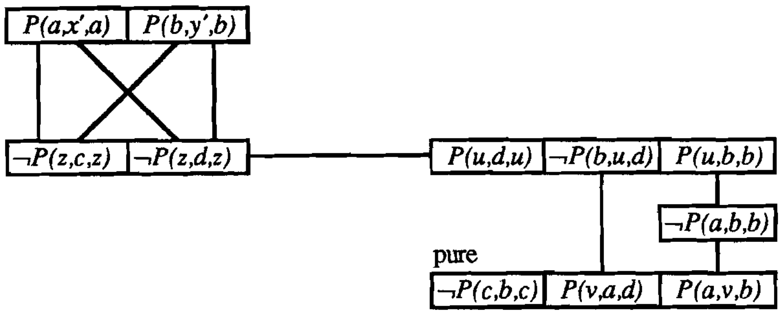
**Example:**

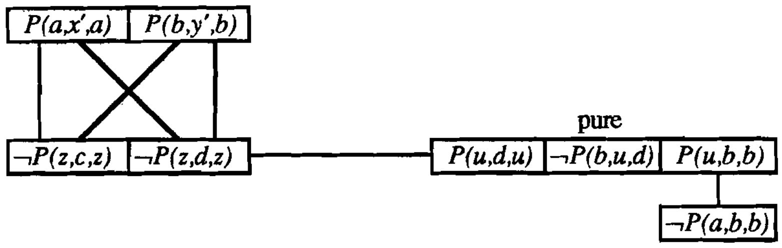


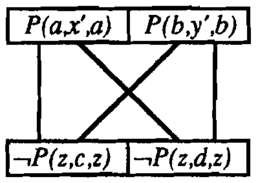
Resolution on the R-link marked by *, with link removal:



Chain reaction:







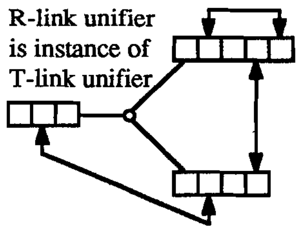
♦

It is in fact possible that all clauses disappear due to this chain reaction – the graph **collapses** to the empty graph. In this case the initial set of clauses was satisfiable.

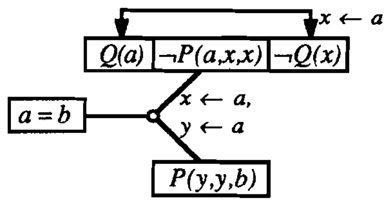
Another class of specific reduction rules exploits that an R-link or I-link actually represents a potential new clause. If after its creation such a new clause would be eliminated right away by some reduction rule for clauses, one can try to detect this in advance. Then a simple removal of the link simulates the operation on the link followed by the application of a reduction rule to the new clause. Such a **look-ahead** link removal can result in purities and thus cause further reductions.

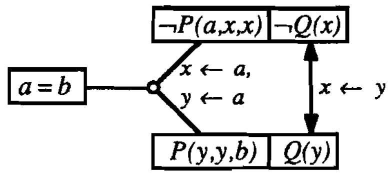
As an example, let us sketch the recognition of tautological R-links, where T-links are used as indicators:

General schema for tautology R-link recognition:



Examples:





Algorithms that recognize links leading to subsumed or pure clauses are also possible, but increasingly complicated and time-consuming, even when using I-links as indicators. In addition, to ensure refutation completeness, one has to test whether a logically redundant link may really be removed from the graph, or whether the clause generated by it would violate one of the link conditions discussed above.

On the whole, it is advisable to weigh the cost for the detection of reduction possibilities against their potential benefit. A fourth advantage of the clause graph representation is that it supports additional reduction rules and thus restrictions of the search space which are not possible with the trivial logical state transition system for resolution.

## 4.3 Properties of Logical State Transition Systems

Since this paper is concerned with deduction systems based on resolution we limit the following considerations to transition systems based on testing calculi. A generalization covering also generating calculi would be straightforward, but would complicate the formalism.

So we assume a logical state transition system with a set of states and a transition relation. Further, we assume that each formula or set of formulae $\mathcal{F}$ of the appropriate logic corresponds to an initial state INIT($\mathcal{F}$) of the state transition system. It is actually not necessary that non-initial states of the system represent logical formulae, although they usually do. Of the final states we assume that they are partitioned into classes, each class standing for a semantic property like satisfiability or unsatisfiability. The idea is that whenever transitions from INIT($\mathcal{F}$) lead to a final state, the system ascribes to $\mathcal{F}$ the property for which the class of this final state stands.

The trivial logical state transition system for the resolution calculus has two classes of final states: the final unsatisfiability states are the clause set containing the empty clause; the final satisfiability states are the clause sets that are closed under resolution and factoring, without containing the empty clause.

Depending on what kinds of links are permitted in a graph and which transition rules are allowed, one can define many different logical state transition systems using clause graphs. The one underlying Kowalski's connection graph proof procedure is as follows: its states are clause graphs with binary (non-theory) links. Its initial states are clause graphs where no possible link is missing. Its final unsatisfiability states are the clause graphs containing the empty clause, and there is just one final satisfiability state, the empty clause graph, which contains neither links nor clauses (not even the empty one). Its transition rules are the clause graph versions of factoring and resolution, including link removal as an integral part of the transition, further the merging, purity, and tautology rule. Let us call this the **cg state transition system.**

For logical state transition systems as described we now define the following properties. The system is called:

**unsatisfiability sound** iff whenever INIT($\mathcal{F}$) $\xrightarrow{*}$ some final unsatisfiability state then $\mathcal{F}$ is unsatisfiable,

**unsatisfiability complete** iff whenever $\mathcal{F}$ is unsatisfiable then INIT(S) $\xrightarrow{*}$ some final unsatisfiability state,

**unsatisfiability confluent** iff whenever $\mathcal{F}$ is unsatisfiable and INIT($\mathcal{F}$) $\xrightarrow{*}$ $S_1$ and INIT($\mathcal{F}$) $\xrightarrow{*}$ $S_2$ then $S_1 \xrightarrow{*} S'$ and $S_2 \xrightarrow{*} S'$ for some state S',

**unsatisfiability closed** iff whenever some final unsatisfiability state $\xrightarrow{*}$ S then S is a final unsatisfiability state.

In exactly the same way we define the properties of the logical state transition system with respect to satisfiability and to other properties of formulae.

These definitions cover the phenomenon that logical state transition systems may have non-initial and unreachable states. In the trivial logical state transition system any formula set could be the one the system was given to start from, thus it has no unreachable states. On the other hand, there are clause graphs that cannot be reached from any initial state in the cg state transition system. One can even construct states from which both the empty clause and the empty graph can be reached, but this is irrelevant because they are not part of the reachable subsystem. The definitions above allow to express properties of the relevant subsystems.

Soundness with respect to a semantic property means that if the system ascribes this property to a formula, then the formula does indeed have the property. The trivial logical state transition system for the resolution calculus is unsatisfiability sound and satisfiability sound. If we defined the clause sets consisting of only tautologies as the final validity states and included the purity rule among the transition rules, the system would not be validity sound.

Completeness with respect to a semantic property guarantees that a formula's having this property can always be demonstrated with the system. The trivial system for resolution is unsatisfiability complete. For some decidable subclasses of first-order predicate logic, for instance for the ground case and for the Herbrand class (where only unit clauses occur), it is also satisfiability complete. See [Joyner 73] for more details on this subject.

Confluence and closedness are the properties that allow irrevocable control strategies in the respective subsystems. Suppose that $\mathcal{F}$ is unsatisfiable and INIT($\mathcal{F}$) $\xrightarrow{*}$ U for a final unsatisfiability state U and also INIT($\mathcal{F}$) $\xrightarrow{*}$ S with an alternative derivation. Then unsatisfiability confluence ensures that there is a continuation S $\xrightarrow{*}$ U' to a common successor state. If the system is unsatisfiability closed, U' is also a final unsatisfiability state. To be closed is just a technicality that can easily be ensured for any system. Confluence, and even commutativity, on all subsystems is an immediate property of trivial logical state transition systems.

For the cg state transition system the results are as follows [Eisinger 89]

- it is unsatisfiability sound and satisfiability sound
- it is unsatisfiability complete
- it is unsatisfiability confluent but not satisfiability confluent.

These results also hold if the subsumption rule is included. Thus the more advanced reduction rules, which are just combinations of simple ones, are also covered. If all of these reduction rules are included, the system is one of the strongest with respect to the ability to recognize the satisfiability while trying to prove the unsatisfiability. This is when satisfiability soundness carries weight.

If we define the tautology rule without the special "bridge link" condition (section 4.2.3), which incidentally is Kowalski's original definition, the system retains all of the properties above for the unit refutable class of clauses (see section 5.1) [Smolka 82]. However, in the general case it loses satisfiability soundness and unsatisfiability confluence. Thus with the original definition there always exists a refutation, but in some cases an attempt to find it may lead into a dead end from which the refutation is no longer possible. This result clearly demonstrates the uselessness of traditional completeness alone. In the case of subsumption there is no similar irrelevance of the link condition for the unit refutable class.

Finally, let us demonstrate that completeness and confluence are indeed independent properties. For propositional Horn clauses it is known that if there is a resolution refutation at all, then there also is one in which no clause is used more than once as a parent clause in resolution steps. We now modify the trivial logical state transition system for the resolution calculus such that each transition replaces two resolvable clauses by their resolvent; that is, the parent clauses of the resolvent are no longer contained in the successor states. By the above, the resulting logical state transition system is unsatisfiability complete for propositional Horn clause sets. The following example shows that on the other hand it is not unsatisfiability confluent for this class of formulae:

$$\begin{tikzcd}
& \{\ \{P\},\{\neg Q,R\},\{\neg R,Q\},\{\neg Q,P\},\{\neg P\}\ \} \arrow[dl] \arrow[dr] & \\
\{\Box,\{\neg Q,R\},\{\neg R,Q\},\{\neg Q,P\}\} & & \{\ \{P\},\ \{\neg Q,R\},\ \{\neg R,Q\},\ \{\neg Q\}\ \} \arrow[d] \\
& & \{\ \{P\},\ \{\neg Q,R\},\ \{\neg R\}\ \} \arrow[d] \\
& & \{\ \{P\},\ \{\neg Q\}\ \}
\end{tikzcd}$$

The clause set on the left is a final state of the unsatisfiability class; from the last set on the right no further transition is possible; no common state can be reached from the two states. In the search space there does exist a clause set containing the empty clause, but an irrevocable control regime might miss it and run into a dead end.

## 4.4 Graphs as Representations for Proofs

Clause graphs can be used for entirely different purposes. So far we have seen them as a richer data structure for the states of a deduction system. However, one can also use special clause graphs to represent the *result* of a deduction system, namely the sequence of derivation steps through which the empty clause was obtained. These clause graphs represent refutations and thus proofs.

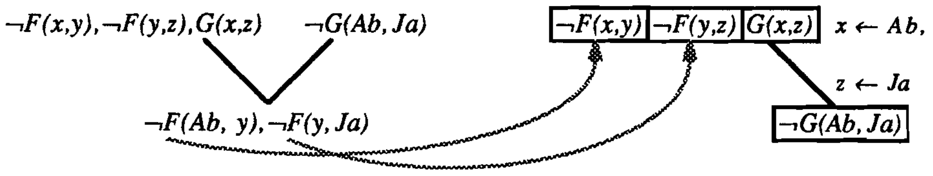
In this section we present an abstraction from the usual notion of a proof. The basic observation is that literals in resolvents are nothing but instances of literals appearing in the initial clause set. In principle a resolvent may therefore be represented by a set of pointers to literals in the initial clause set, along with a substitution. Yet, not even this is really necessary. Actually it suffices to mark the resolution literals in the initial clauses by an R-link. Such a link then signifies (as opposed to its meaning in the state transition systems discussed so far) that the corresponding resolution step is to be considered as executed and that thus both resolution literals have been "resolved away". The set of literals not incident with a link, or rather their instances, now corresponds to the resolvent. We demonstrate the idea with a simple example.

**Example:** *Abraham* is the father of *Isaac*, and *Isaac* is the father of *Jacob*. Therefore *Abraham* is the grandfather of *Jacob*. In clausal form the hypotheses and the negated conclusion are as follows:

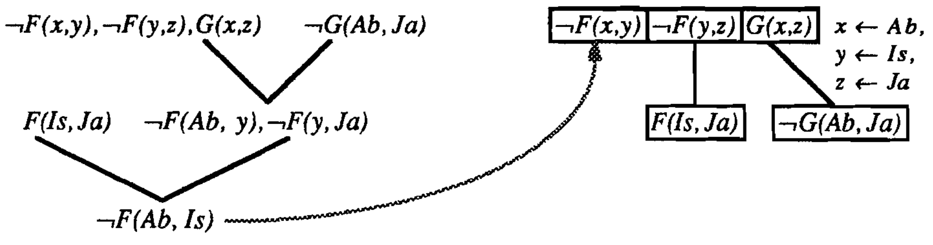
*Father(Abraham, Isaac)*
*Father(Isaac, Jacob)*
*¬Father(x, y), ¬Father(y, z), Grandfather(x, z)*
*¬Grandfather(Abraham, Jacob)*

Below we develop a resolution refutation on the left hand side, and on the right hand side a clause graph in which the successive resolution steps are represented by R-links joining the initial clauses. The shaded connections are pointers from the literals in the resolvents to their literals of origin. To save space, we abbreviate predicate symbols and constant symbols.

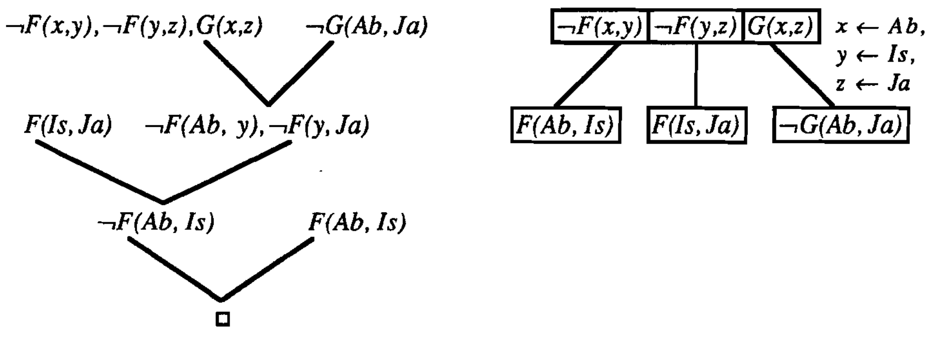
First resolution step:



Second resolution step:

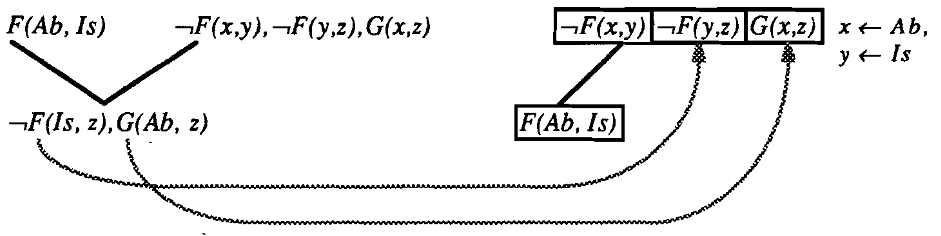


Third resolution step:

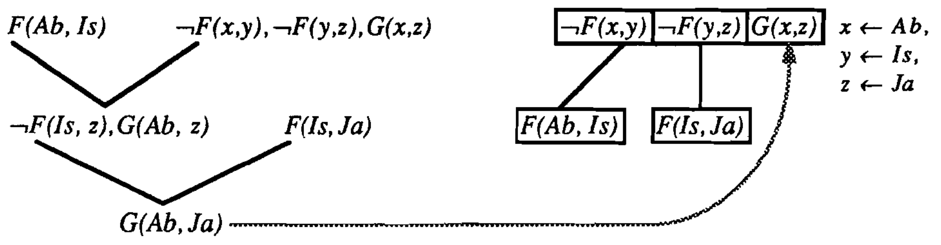


This was a proper refutation proof. The following three diagrams show a "positive" resolution proof, where the conclusion is explicitly derived as the second-to-last clause.

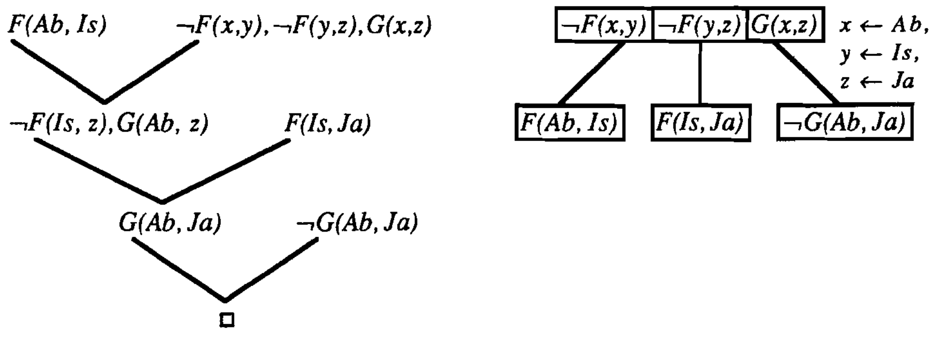
First resolution step:



Second resolution step:



Third resolution step:



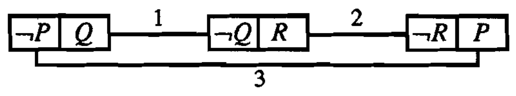
♦

Although the two resolution proofs are different, they result in the same clause graph. In either refutation, each of the three unit clauses is used as a parent clause exactly once, only in different orders. The clause graph representation abstracts from this irrelevant order of the steps and altogether represents six different refutations. These can be reconstructed by executing the steps represented by the R-links in any order, using the transition rules described before.

A clause graph which in that way represents a class of derivations of the empty clause, is called a **refutation graph**.

Naturally, not every clause graph is also a refutation graph. The example above shows some conditions such graphs have to meet: every literal node is incident with exactly one R-link, and there exists a global substitution that unifies every pair of connected atoms. All clauses are members of the initial clause set to be refuted. If an initial clause is needed twice in the resolution derivation, the refutation graph contains two copies of this clause with renamed variables, so that the copies may be instantiated in different ways. A resolvent needed several times corresponds to copies of the subgraph representing this resolvent.

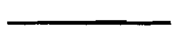
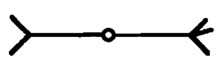
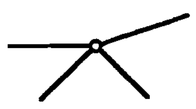
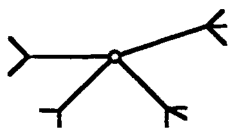
Finally, a refutation graph must not contain any cycles. A cycle is a path that starts from a clause and along one or more links returns to the same clause. Without this condition, the following satisfiable graph would also be a refutation graph:



The clause set corresponds to the formulae $P \Rightarrow Q$, $Q \Rightarrow R$, $R \Rightarrow P$. Resolution on R-links 1 and 2 results in the tautology $\{\neg P, P\}$, the empty clause cannot be derived. The link sequence 1, 2, 3 is a cycle. The example shows that cyclic paths represent derivation chains "biting their own tail": in order to prove $P$, one has to prove $R$; to prove $R$, one has to prove $Q$; to prove $Q$, one has to prove $P$; ...

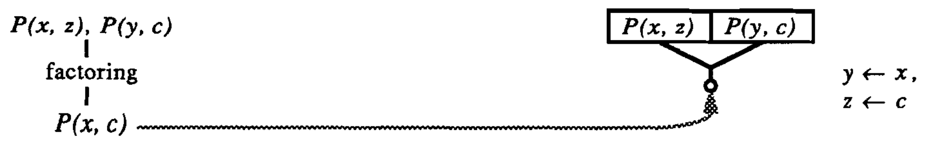
Refutation graphs were first examined by Robert E. Shostak [Shostak 76]. He was able to show that a clause set is unsatisfiable if and only if for a sufficient number of copies of these clauses there exists a refutation graph, that is a nonempty, noncyclic clause graph in which every literal node is incident with exactly one R-link, so that some global substitution unifies every pair of connected atoms.

In order to represent in a refutation graph the phenomenon of two literals being merged by factoring, we have to generalize the form of R-links for refutation graphs. Each side of an R-link may be incident not only with one, but with several literal nodes, and thus the link may fan out on either side.

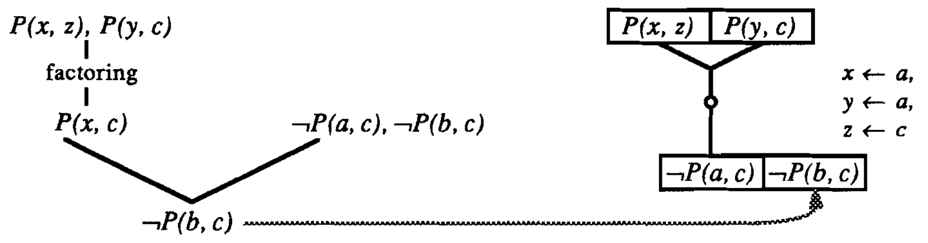
| Form of R-links | clause graph | refutation graph |
|---|---|---|
| simple resolution: |  |  |
| theory resolution: |  |  |

An R-link consists of as many major branches as there are clauses in the resolution step. For simple resolution, there are two, but for theory resolution, there may be several. Each major branch of an R-link fans out into one or more minor branches, which show which literals must merge before the resolution step can actually be executed.

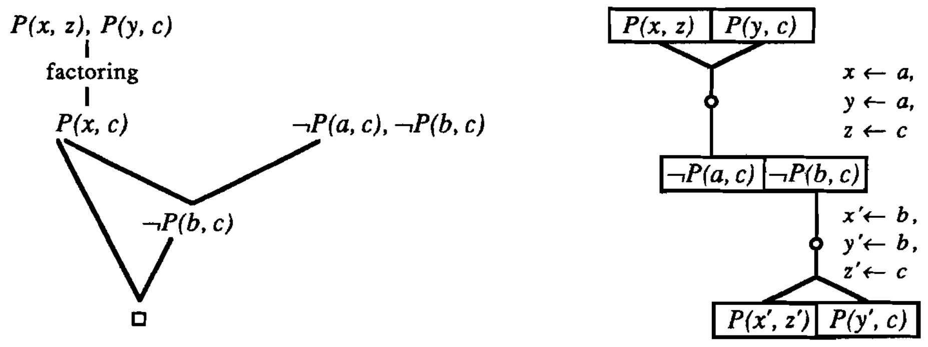
**Example:** First step:



Second step:



Third step:



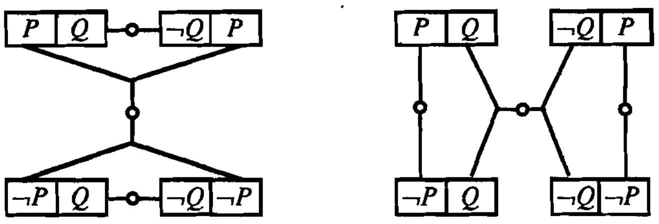
♦

It would be possible to generalize the definition of R-links and I-links (recall that R-links are just special cases of I-links) not only for refutation graphs, but for arbitrary clause graphs in the way above. Then a major branch of an R-link would correspond to the disjunction of the literals at its minor branches, while the entire R-link would still represent the conjunction of all its major branches. To carry out a resolution step, one has to select exactly one minor branch of each major branch, and use their literals as resolution literals. But we do not pursue this generalization.

In a refutation graph, a major branch does not have to fan out to a single clause, as it happens in the example above. Minor branches leading to different clauses indicate that not the initial clauses, but some resolvent of theirs has to be factored or merged. When no major branch of any R-link fans out to different clauses, this is a special kind of refutation graph; it represents a derivation factoring only initial clauses.

Even more special refutation graphs are such that the major branches do not fan out at all, representing derivations without factoring or merging. In this case, the refutation graph has a treelike structure and is also called a **refutation tree**. A theorem by M.C. Harrison and N. Rubin [Harrison & Rubin 78] states that there exists a refutation tree for a given clause set if and only if this clause set is unit refutable (for instance, if it is an unsatisfiable Horn clause set).

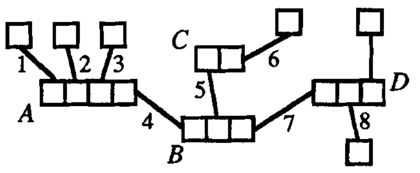
The clause set *{{P, Q}, {¬P, Q}, {¬Q, P}, {¬Q, ¬P}}* is unsatisfiable and has only refutation graphs that are no refutation trees. Thus it is not unit refutable, and any refutation requires at least one merging step. The reader may wish to try and construct the resolution refutations represented by the following two refutation graphs for this set:



## 4.5 Extracting Refutation Graphs from Clause Graphs

The study of refutation graphs leads to a better understanding of the topological structure of resolution proofs and thus of the interdependencies of consequences. Refutation graphs are also quite useful in examining theoretical properties of deduction systems. So far we did not tell whether they are also useful in searching for a proof; in the examples we always translated complete resolution refutations into corresponding refutation graphs. How to find these refutations, remained open.

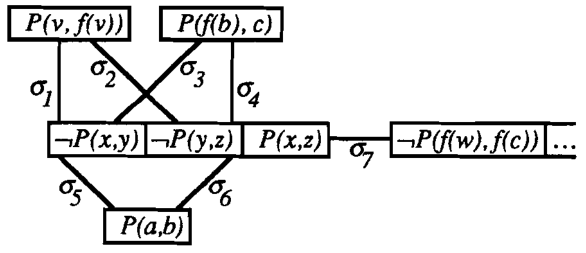
In this section we introduce a procedure that enables us to extract a refutation tree for a unit refutable clause set directly from the initial clause graph. If the procedure succeeds, a proof has been found without generating a single resolvent. The basic idea leading to the extraction procedure can best be seen in an example involving a somewhat more complex refutation tree for a propositional logic clause set:



In this refutation tree there are three clauses, *A*, *C*, and *D*, with the property that all but one of their literals resolve with a unit clause. Let's take, say, *A* and successively resolve with all three of the unit clauses. The final resolvent consists of just the fourth literal node *A4* and is a new unit clause linked to *B* by a descendant of link 4. Proceeding likewise with *C*, we obtain another unit clause *C1*, connected to *B* by a descendant of link 5. Combined, *A4* and *C1* enable us to resolve away the first two literal nodes in *B*, leaving *B3*. At this point every literal node in *D* is linked to a unit clause and the empty clause can be derived.

Thus in a clause graph containing all possible R-links, one simply has to look for a subgraph in which each literal of a clause is connected to a unit clause by an R-link. If this succeeds, one has found a refutation tree and is done. Otherwise, one considers the subgraphs in which all but one of the literals of a clause are connected to a unit clause by an R-link. A subgraph of this kind represents a unit resolvent consisting of a successor of the exempted literal. This literal can now be treated as if it were a "proper" unit clause, such that an appropriate subgraph may be found for other clauses, for which this was not possible before.

When working with predicate logic clause sets, the unifiers of the R-links have to be considered as well. Our next example, which contains a transitivity axiom, shows the effects of that.



$\sigma_1: x \leftarrow v, \quad y \leftarrow f(v)$
$\sigma_2: y \leftarrow v, \quad z \leftarrow f(v)$
$\sigma_3: x \leftarrow f(b), y \leftarrow c$
$\sigma_4: y \leftarrow f(b), z \leftarrow c$

$\sigma_5: x \leftarrow a, \quad y \leftarrow b$
$\sigma_6: y \leftarrow a, \quad z \leftarrow b$
$\sigma_7: x \leftarrow f(w), z \leftarrow f(c)$

Here it is of course not the third literal *P(x, z)* of the transitivity axiom itself that can be turned into a unit clause, but at best some instance of it. Possible instances are determined by finding compatible combinations of unifiers of the R-links to the unit clauses. In order to see which ones these are in our example, we organize the unifiers in a table, showing the instantiations for the variables *x, y, z* in a fixed order one beneath the other. The four compatible combinations are now found by unifying pairs of term lists, namely those given by the entries for the respective substitutions in the two tables. (The variable *v* in $\sigma_2$ is renamed in order to simulate that another copy of the clause *P(v,f(v))* must be used.)

| | x | y | z |
|---|---|---|---|
| $\sigma_1$ | v | f(v) | z |
| $\sigma_3$ | f(b) | c | z |
| $\sigma_5$ | a | b | z |

| | x | y | z |
|---|---|---|---|
| $\sigma_2$ | x | v' | f(v') |
| $\sigma_4$ | x | f(b) | c |
| $\sigma_6$ | x | a | b |

| | x | y | z |
|---|---|---|---|
| $\sigma_1, \sigma_2$ | v | f(v) | f(f(v)) |
| $\sigma_1, \sigma_4$ | b | f(b) | c |
| $\sigma_3, \sigma_2$ | f(b) | c | f(c) |
| $\sigma_5, \sigma_2$ | a | b | f(b) |

From the four compatible combinations, four new unit clauses can now be derived by instantiating the third literal *P(x, z)*. These new unit clauses are: *{P(v,f(f(v)))}*, *{P(b,c)}*, *{P(f(b),f(c))}*, and *{P(a,f(b))}*. By looking at the unifier $\sigma_7$ one realizes that only those instances can be useful later on, in which *x* is replaced by *f(...)* and *z* by *f(c)*. With this constraint only the combination $\sigma_3, \sigma_2$ remains, resulting in the unit clause *{P(f(b),f(c))}*. However, the corresponding instance of $\sigma_7$ can also be computed directly from $\sigma_7$ and the combination $\sigma_3, \sigma_2$ by a **merging algorithm** for substitutions; the result is $\{x \leftarrow f(b),\ z \leftarrow f(c),\ w \leftarrow b\}$, and the detour of computing the instantiated literal has been saved.

The whole procedure for the extraction of refutation trees from clause graphs now works as follows [Antoniou & Ohlbach 83]:

Select a clause whose literals are connected to unit clauses by R-links, with the exception of at most one literal, say *K*. Compute the set of compatible combinations of the unifiers of the R-links to unit clauses. For each unifier of each R-link incident with literal *K* and for each compatible combination, apply the merging algorithm. Mark the R-links of *K* with the instances of their unifiers thus obtained. These instances correspond to the unifiers of the R-links to the unit clauses potentially derivable from *K*. They can, in subsequent search steps, be used as if the unit clauses were indeed "properly" present in the graph. The algorithm terminates if a clause is found in which all literals are connected to proper or potential unit clauses by R-links with compatible substitutions. In order to construct the tree and, if required, to translate it into a resolution proof, one simply has to gather all used links and substitutions.

This approach is based on the observation that refutations are characterized by a topological structure and a compatibility property of the substitutions involved. By extracting the topologically suitable subgraphs, whole classes of refutations are treated at once, such that an improvement of the overall cost can be expected.

In the version presented, the method applies to clause sets that are unit refutable with simple resolution. The extension to theory resolution is unproblematic. A generalization of this method to non-unit-refutable clause sets has not yet been worked out, but there are a number of approaches with the same underlying idea [Sickel 76, Chang & Slagle 79, Shostak 79]. The matrix method of Andrews and Bibel also belongs to that category [Andrews 68, Bibel 81,82].

# 5 Control

In principle, all one needs to do in order to find a proof with a logical state transition system, is to systematically enumerate all states reachable from the respective initial state – assuming, of course, appropriate system properties. However, deduction systems become feasible only if they avoid "bad" steps and prefer "good" steps as much as possible.

The selection of "good" steps actually requires domain specific knowledge about the field to which the statements to be proven refer. Unfortunately, so far little is known about what exactly constitutes such knowledge and how a deduction procedure can be controlled by it. But there are also purely syntactic criteria, which exploit only the structure of the formulae and can therefore be applied independent of the domain. Although they are necessarily of limited power, they do help to avoid gross inefficiencies. At the present state of the art, such syntactic criteria are the decisive factor for a tolerable functioning of deduction systems.

For the resolution calculus two types of syntactic criteria have been studied. The emphasis has long been on **restriction strategies**, which prohibit some of the possible steps altogether. Essentially they result in a smaller branching rate of the search space, but compared to the unrestricted system they often increase the lengths of the proofs in return, such that their overall benefit becomes questionable. Some restriction strategies empirically turned out to be quite useful, though. In contrast to restriction strategies, **ordering strategies** do not prohibit any steps but dictate the order in which the possible steps are chosen. Syntactic heuristics can play a rôle in determining this order.

In some cases the difference between restriction strategies and ordering strategies becomes somewhat blurred. Whenever an ordering strategy prefers an infinite number of steps over certain others, it has the effect of a restriction strategy.

## 5.1 Restriction Strategies

One of the most simple restriction strategies for resolution is called **unit resolution**. It prohibits the generation of resolvents from two parent clauses if both of them contain more than one literal. Worded positively, every resolvent must have at least one unit parent clause. This restriction greatly reduces the number of successor states of a clause set. It always leads to resolvents with fewer literals than the larger parent clause. Moreover, it is easy to implement. Unit resolution has also proved rather successful in practice.

However, sometimes this strategy is too restrictive. A restriction strategy ought to preserve as many of the properties of the underlying state transition system as possible. While soundness properties cannot be affected by restriction strategies, completeness and confluence properties might be lost. Unit resolution is not in general unsatisfiability complete; that is, with this strategy the empty clause cannot be derived from each unsatisfiable clause set. Still, unsatisfiability completeness is guaranteed for an important class of clause sets, which includes the class of Horn clause sets and which for lack of a syntactic characterization is called the **unit refutable** class. This is the same class of clause sets for which refutation trees can be constructed (see sections 4.4 and 4.5).

Independent of completeness, one also has to ensure the confluence of restriction strategies for the relevant classes of formulae. For example, we could define a restriction strategy for simple resolution by prohibiting that any clause be used more than once as a parent clause. This restriction strategy would be unsatisfiability complete for propositional Horn clause sets, but not unsatisfiability confluent. Compare the remarks on unsatisfiability completeness and confluence of state transition systems at the end of section 4.3.

For the class of unit refutable clause sets, another important restriction strategy, **input resolution,** is unsatisfiability complete. It prohibits the generation of resolvents whose parent clauses are both resolvents. Worded positively, every resolvent must have at least one parent clause from the initial clause set. The major advantage of this restriction is that for each admissible resolution step, one of the resolution literals is known a priori. In particular, any unification involves some arbitrary atom and an atom from the initial clause set. For each of the latter a specific unification algorithm can be "compiled", which computes the most general unifiers for "its" initial atom and an arbitrary atom it takes as an argument. Such an algorithm is usually much more efficient than one capable of unifying two arbitrary atoms.

In an input derivation each resolvent has a "far parent" from the initial set and a "near parent" that may be any clause. The resolvents and factors can be partially ordered by the ancestor relation, which imposes a tree structure upon them. Many restrictions weaken the condition on the far parent in order to cover arbitrary clause sets, while trying to preserve as much as possible of the flavour of the input restriction.

The **merging restriction** [Andrews 68] accepts as far parent either an input clause or a "merge". Such a clause results from a resolvent by merging or factoring two literals descending from different parent clauses.

Another relaxation characterizes **linear resolution.** Beside input resolution steps, this strategy also permits resolution steps between two resolvents in cases where one is an "ancestor" of the other. By admitting ancestor steps only when a factor of their resolvent subsumes the near parent and by making this factoring compulsory, we obtain the **s-linear restriction** [Loveland 78]. The **t-linear restriction** [Kowalski & Kuehner 71] further limits ancestor steps to so-called "A-ancestors", all of whose literals excepts for the one resolved upon have descendants in all intermediate clauses down to the near parent. Moreover, near parents resolvable with an A-ancestor where the resolvent subsumes the parent clause may not participate in input steps. An additional qualification calling for a single most recently introduced literal of the near parent to be the only one ever resolved upon in input steps leads to **SL resolution** [Kowalski & Kuehner 71].

Some restrictions are more concerned about the factoring rule. For example, the **half-factoring** restriction excludes resolution steps between two factors [Noll 80]. Finally, factoring can be restricted in a new way. Let "kindred literals" be such as have a common ancestor literal. For instance, resolution between $\{P(x), Q(z)\}$ and $\{\neg Q(a), \neg Q(b)\}$ produces $\{P(y), \neg Q(b)\}$ which again resolves with the first clause giving $\{P(x'), P(y')\}$. This resolvent consists of kindred literals. The **kindred factoring restriction** confines factoring to kindred literals.

Finally, the **set-of-support** restriction strategy is widely used and on the whole rewarding. It distinguishes between clauses stemming from the hypotheses and clauses obtained from the (negated) conclusion. Assuming that the hypotheses are not contradictory in themselves, a contradiction must involve the conclusion. Therefore the strategy prohibits the generation of resolvents from two hypothesis clauses. Somewhat more general, one can distinguish any satisfiable subset of the initial clause set, and prohibit resolution between members of this distinguished subset; only the other clauses are "supported".

## 5.2 Ordering Strategies

The most simple ordering strategy for resolution is the **level saturation** strategy. Each clause is associated with its "depth": initial clauses from the original clause set have depth 0, factors are of the same depth as their parent clauses, and the depth of a resolvent is one more than the larger of the parent clauses' depths. The level saturation strategy orders the possible steps simply by the depths of the clauses generated, so that a clause of depth *n* may be added to a clause set only if all clauses with a smaller depth have already been derived. The order in which clauses of the same depth are generated is not specified, but left up to the particular implementation. A possible condition might be to prefer clauses with the smallest number of literals from among all clauses of the same depth.

A frequently used modification of the level saturation strategy is obtained by subtracting some constant positive integer *c* from the depth of each resolvent having a unit parent clause. Then, all derivable factors and resolvents will still be systematically generated, but those with a unit parent occur *c* levels earlier than others. This variant of the level saturation strategy has become known as the **unit preference** strategy.

At least for the trivial logical state transition system of the resolution calculus, both strategies are **exhaustive**: for any clause appearing in any state of the search space, a state containing this clause will be reached after a finite number of steps (unless a final state is reached before that). If the underlying state transition system and the combination of restriction strategies used are complete and confluent with respect to some class of final states, then an exhaustive ordering strategy always reaches a final state of this class after finitely many steps.

In some systems, however, it is impossible for ordering strategies to be exhaustive. Then one has to guarantee at least their **fairness**: no step must be postponed relative to infinitely many others.

The preference of certain steps need not be confined to resolvents with unit parent clauses in order to preserve fairness. When a given fair ordering strategy is modified such that steps producing resolvents with fewer literals than either parent clause are given priority over all other steps, another fair ordering strategy is obtained. The highest priority can in principle be given to any class of steps considered useful, provided that no infinite succession of steps of that class is applicable from any state. For example, in the cg state transition system one might give highest preference to the resolution steps that cause the purity of both parent clauses, or to the steps that render one parent clause pure and derive a clause that is shorter than this parent clause.

Many reduction rules actually also contribute to an ordering strategy. Strictly speaking, the look-ahead link removal (section 4.2.4) enforces that a link whose factor or resolvent can be eliminated by some clause reduction rule be processed before any other links. Even subsumption factoring and subsumption resolution, which only eliminate single literals from clauses, really simulate the derivation of new clauses. If they get highest priority, this defines a new ordering strategy. In general, one cannot combine arbitrary reduction rules with arbitrary ordering strategies without jeopardizing properties of the strategy.

For those cases in which certain steps cannot be generally preferred, one can define a heuristic priority. To ensure fairness, the depth of the clauses has to enter into this priority. But the priority may also depend on further syntactic features, such as the number of literals or the term complexity. It is a common technique for heuristic search to compute the priority value as a weighted sum of these feature values, where the weights can be adjusted by the user in order to influence the system's behaviour.

In principle, the heuristic values might also be based on domain specific knowledge, although there is the standard objection that such knowledge can hardly be encoded in a simple priority value.

And come to that, the user is often endowed with control knowledge that ought to be made available to the system in an appropriate way. However, at present the structure of such control knowledge and the mechanisms to bring it in are largely unknown.

## 5.3 Filters

For a formal description of strategies we have to return to the general level of logical state transition systems with a set S of states and a transition relation $\rightarrow$. Here the difference between restrictions and orderings disappears.

A **filter** for a state transition system is a unary predicate $\Phi$ on the set of finite sequences of states. The notation $S \overset{*}{\underset{\Phi}{\rightarrow}} S'$ stands for a derivation $S \overset{*}{\rightarrow} S'$ where $\Phi(S\ldots S')$ holds. For an infinite derivation, $S_0 \underset{\Phi}{\rightarrow} \ldots \underset{\Phi}{\rightarrow} S_n \underset{\Phi}{\rightarrow} \ldots$ means that $\Phi(S_0\ldots S_n)$ holds for each n.

The name "filter" is due to Smolka [Smolka 82]. Intuitively, a filter $\Phi$ conceals all derivations for which the predicate does not hold, i.e. $\Phi$ retains a sub-portion of the state transition system's original search space. While it is not impossible to apply some tentative control regime in the remaining search space, there is a canonical way to associate with $\Phi$ a class of irrevocable control strategies. Having derived from some initial state $S_0$ some non-final state $S_n$, such a strategy may choose as successor any state S with $S_n \rightarrow S$ and $\Phi(S_0\ldots S_n S)$, thus pursuing one single derivation with $\Phi$. The strategy may freely exploit any nondeterminism left by the filter, and one may not make any assumption about its choices. Under these conditions the system's behaviour depends entirely on the properties of the filter $\Phi$.

Traditional strategies for resolution are often described by means of an auxiliary structure called **deduction tree** [Chang & Lee 73, Loveland 78], which is an upward growing tree whose nodes are clauses and whose arcs connect resolvents or factors with their parent clauses (we used deduction trees in the Abraham-Isaac-Jacob example in section 4.4). One can think of deduction trees as being embedded in clause sets or clause graphs or other kinds of states composed of clauses in the following way: Given $S_0 \overset{*}{\rightarrow} S_n$, associate with each clause *C* in $S_n$ a deduction tree made up according to the steps of the derivation. Its nodes are clauses from the union of all $S_i$, with leaves from $S_0$ and root *C*. Loosely speaking, $S_n$ contains a set of such deduction trees. A transition step $S_n \rightarrow S$ producing a new clause introduces a new deduction tree, which is contained in G and not contained in $S_n$, but its immediate subtrees are.

A restriction strategy essentially depends upon a predicate that admits only certain deduction trees. The corresponding **restriction filter** $\Phi$ is defined as $\Phi(G_0\ldots G_n G)$ iff $G_n \rightarrow G$ and the deduction tree introduced by G is admitted by this predicate. An ordering strategy relies on a **merit ordering** of deduction trees (or of linearizations thereof [Kowalski 70]). Given a merit ordering one obtains the corresponding **ordering filter** $\Phi$ as $\Phi(G_0\ldots G_n G)$ iff $G_n \rightarrow G$ and none of the deduction trees introduced by any other potential successor of $G_n$ has better merit than the deduction tree introduced by G.

**Example**: Let $\Phi_{INPUT}(S_0 \ldots S_n S)$ iff the root of any deduction tree introduced by S is adjacent to a leaf from $S_0$. Further let a deduction tree $T_1$ have better merit than $T_2$, if the depth of $T_1$ is less than the depth of $T_2$, and let $\Phi_{LEVEL}$ be the ordering filter corresponding to this merit ordering. For the trivial state transition system for the resolution calculus an irrevocable control strategy based on $\Phi_{INPUT} \wedge \Phi_{LEVEL}$ performs a level saturation derivation in the search space left by the input restriction. The remaining nondeterminism leaves it up to the strategy to sequentialize at pleasure the generation of clauses of the same level. ♦

Reduction steps can also be treated by both kinds of filters. An ordering filter might postpone the removal of subsumed clauses to certain resolution steps. A restriction filter might preclude backward subsumption. Furnished with appropriate node attributes indicating if and why the clause is not present in the respective state, deduction trees may serve as a definitional aid for such cases too.

With this background the behaviour of control strategies can be described in terms of filters and their properties. There is a natural way to accommodate to filters the notions introduced in section 4 for the state transition system:

A filter $\Phi$ for a logical state transition system with initial state $INIT(\mathcal{F})$ for a formula $\mathcal{F}$ is called:

| | | |
|---|---|---|
| **unsatisfiability sound** | iff | whenever $INIT(\mathcal{F}) \xrightarrow[\Phi]{*}$ some final unsatisfiability state then $\mathcal{F}$ is unsatisfiable; |
| **unsatisfiability complete** | iff | whenever $\mathcal{F}$ is unsatisfiable then $INIT(\mathcal{F}) \xrightarrow[\Phi]{*}$ some final unsatisfiability state |
| **unsatisfiability confluent** | iff | whenever $\mathcal{F}$ is unsatisfiable, and $INIT(\mathcal{F}) \xrightarrow[\Phi]{*} S_1$ and $INIT(\mathcal{F}) \xrightarrow[\Phi]{*} S_2$ then $INIT(\mathcal{F}) \xrightarrow[\Phi]{*} S_1 \xrightarrow[\Phi]{*} S$ and $INIT(\mathcal{F}) \xrightarrow[\Phi]{*} S_2 \xrightarrow[\Phi]{*} S$ for some S; |
| **unsatisfiability Noetherian** | iff | whenever $\mathcal{F}$ is unsatisfiable then there is no infinite derivation $INIT(\mathcal{F}) \xrightarrow[\Phi]{} S_1 \xrightarrow[\Phi]{} \ldots \xrightarrow[\Phi]{} S_n \xrightarrow[\Phi]{} \ldots$ |

Note that $\xrightarrow[\Phi]{}$ needs not be transitive, hence the special form of confluence. Again, the concepts for satisfiability and other properties read correspondingly. ♦

The logical state transition system's soundness properties are obviously not affected by any strategy, i.e. each filter is unsatisfiability sound and satisfiability sound if the state transition system is. Unsatisfiability completeness of a filter $\Phi$ only signifies the existence of a refutation with $\Phi$ from any unsatisfiable initial state. For the trivial state transition system this is the central and usually the only property of restrictions ever investigated. Unsatisfiability confluence of a filter is necessary to avoid the need for backups to earlier states of a derivation, like the same property of the sate transition system. Most of the restrictions mentioned in section 5.1 are unsatisfiability complete for the trivial state transition system for resolution and many of them can even be combined without destroying unsatisfiability completeness. Their unsatisfiability confluence is usually rather obvious for the trivial state transition system, but hardly ever mentioned.

Unsatisfiability completeness and unsatisfiability confluence of a restriction filter do not exclude infinite branches in the search space. It requires additional properties of the filter to ensure that an existing refutation will actually be found by an irrevocable control regime. The normal thing would be to call for unsatisfiability Noetherianness. Alternatively, one might squeeze a restriction to a point where it is unique, i.e. it accepts at most one successor state in each situation. Then its irrevocable application is deterministic and unsatisfiability completeness alone is sufficient. While for certain state transition systems, e.g. in syntax analysis, filters with such qualities do exist, none of the traditional restriction filters for resolution comes anywhere near uniqueness or unsatisfiability Noetherianness.

Instead, termination is usually considered a problem to be handled by an ordering strategy. The definition of an appropriate merit ordering yields an ordering filter. For each of them one has to ascertain unsatisfiability completeness, unsatisfiability confluence, and unsatisfiability Noetherianness, and ideally these qualities should be maintained in conjunction with any unsatisfiability complete and unsatisfiability confluent restriction filter.

Now in fact all orderings proposed for resolution belong to the same class of **exhaustive** orderings. A merit ordering is exhaustive, if each derivation admitted by the corresponding filter potentially, i.e. if sufficiently continued to non-final states, reaches every deduction tree in the search space. Note the repugnance of this property with the very nature of restriction filters. Exhaustive ordering filters trivially enjoy all the properties we strive for.

Unfortunately exhaustiveness heavily depends on the state transition system's commutativity. For non-commutative systems exhaustive ordering filters do not in general exist. Then we must attempt to capture the intention of exhaustiveness with a weaker notion: **fairness**, which means that *each* possible operation has a finite chance to be performed and none is infinitely postponed. Depending on the state transition system, the precise definition of fairness may vary. The goal is that each fair ordering filter, when combined with any unsatisfiability complete and unsatisfiability confluent restriction filter, is unsatisfiability complete, unsatisfiability confluent and unsatisfiability Noetherian.

## 5.4 Results on Control Strategies for the CG State Transition System

In this section we summarize the most important findings on control strategies for the cg state transition system [Eisinger 89]. Section 4.3 described the properties of this state transition system itself.

Allowing a single application of a copy rule[1] at the beginning, the combination and thus each subset of the following restrictions is unsatisfiability complete: set-of-support, linear, s-linear, half-factoring, kindred factoring. With the same proviso the t-linear and the SL-restriction are unsatisfiability complete, and so is the input restriction for the unit refutable class. Without the copy rule the unit restriction is unsatisfiability complete for the unit refutable class and the input and the SL restriction are for the Horn class.

The conjunction of the set-of-support and the merging restriction is not unsatisfiability complete (although it is for the trivial system [Andrews 68]). This demonstrates that completeness results for strategies for the trivial state transition system do not simply carry over to other state transition systems.

[1] The copy rule allows insertion of additional variants of a clause in the graph. The copy rule is needed in the completeness proof. There is a strong conjecture that this rule is an artifact of the completeness proof technique and actually not necessary.

Neither of the linear, s-linear, t-linear, SL, or the merging restriction is unsatisfiability confluent, even for the Horn class. The half-factoring restriction is not unsatisfiability confluent. For the unit refutable class the input restriction is not unsatisfiability confluent, but the unit restriction is. So is the SL restriction for the Horn class. The set-of-support and the kindred factoring restriction, single or combined, are unsatisfiability confluent in the general case.

Fairness of an ordering filter has been defined by the concept of "covering". An ordering filter $\Phi$ for the cg state transition system is called covering, if the following holds: Let $G_0$ be an initial clause graph, let $G_0 \overset{*}{\underset{\Phi}{\Rightarrow}} G_n$ be a derivation, and let $\lambda$ be an R-link in $G_n$. Then there is a number $n(\lambda)$, such that for any derivation $G_0 \overset{*}{\underset{\Phi}{\Rightarrow}} G_n \overset{*}{\underset{\Phi}{\Rightarrow}} G$ extending the given one by at least $n(\lambda)$ steps, $\lambda$ is not in G.

With a covering filter no link may be infinitely delayed. It seems plausible that in combination with an unsatisfiability complete and unsatisfiability confluent restriction filter (or at least with the null restriction) *each* covering ordering filter is unsatisfiability complete, unsatisfiability confluent and unsatisfiability Noetherian. This presumption is a precise wording of what has been known as the **strong completeness conjecture** for the cg state transition system.

For the unit refutable class this conjecture turns out to be true. In general, however, it is false, even if no restriction filter is involved: There is a ground case example which derives from an unsatisfiable initial graph G a graph G', such that all links and all clauses in G disappear during the derivation, but G and G' are isomorphic. Define an ordering filter that enforces the steps leading from G to G' and subsequently the analogous steps, thus admitting of only one derivation, which is infinite. This filter is covering and is neither unsatisfiability complete nor unsatisfiability Noetherian.

Apart from attempts at proving strong completeness conjecture, unsatisfiability Noetherianness has not been definitely established for any particular ordering filter for the general case, not even for the obviously covering $\Phi_{LEVEL}$. An irrevocable strategy based on this filter, called **level monotonicity strategy**, cannot enumerate all deduction trees in the search space, but comes as close to a level saturation strategy as one can get with the cg state transition system.

Another ordering filter is founded on the idea to select some positive literal occurrence $L = P(\ldots)$ and then to successively resolve on all links incident with $L$. If (a) $L$ is not incident with an internal R-link, it must eventually become pure. Provided that (b) no positive literal with predicate symbol $P$ was introduced into any resolvent, the number of positive occurrences of $P$ decreases with the subsequent purity removal. For classes of formulae where (a) and (b) hold a priori, a systematic elimination of all predicate symbols can be developed by taking advantage of these observations. The resulting **predicate cancellation** filter is trivially Noetherian. In general, however, the conditions are not met and a predicate cancellation filter does not exist. Even if it does, it is incompatible with almost every restriction filter.

At first glance these results on strategies for the cg state transition system are very disappointing. Practice has shown, however, that the traditional control strategies are not that significant for the overall efficiency and that heuristics exploiting the topological properties of the graph are much more important. Furthermore the practical incompleteness caused by time and memory limitations totally outweigh any theoretical problems.

## 6 Conclusion

The ability to draw logical conclusions is of fundamental importance to intelligent behaviour. For this reason, deduction components are an integral part of numerous Artificial Intelligence systems. Even though the original motivation for developing these systems was to automatically prove mathematical theorems, their applications now go far beyond. Logic programming languages such as PROLOG have been developed from deduction systems, and these systems are used within natural language systems and expert systems as well as in intelligent robot control. In addition, this field's logic-oriented methods have influenced the basic research of almost all areas of Artificial Intelligence.

In this paper we have presented a general theory of deduction systems. The theory has been illustrated with deduction systems based on the resolution calculus, in particular using clause graphs. In this theory there are four levels making up an entire deduction system. The first level is constituted by a logic, which establishes the syntax and the semantics of a formal language and thus defines the permissible structure and meaning of statements. The logics of interest to deduction systems define a notion of semantic entailment which, however, does not provide any means to determine algorithmically whether or not a given statement entails another one. This is accomplished by a calculus, the second level of a deduction system. A calculus defines syntactic derivations as operations on formulae. The third level of a deduction system, the logical state transition system, determines the description of formulae or sets of formulae together with their interrelationships, and also the representation of the various states of the derivation chains. Often, additional operations are introduced at this level, for instance the removal of redundant statements, rendering impossible some derivations allowed by the calculus – hopefully unnecessary ones. Finally, the control, the fourth level a deduction system is composed of, comprises the strategies and heuristics used to choose the most promising among all applicable derivation steps.

For the last two levels appropriately adjusted notions of soundness, completeness, confluence and Noetherianness have been introduced in order to characterize the properties of particular deduction systems. For more complex deduction systems, where logical and topological phenomena interleave, these properties can be far from obvious. We discussed these properties in particular for the system underlying Kowalski's connection graph proof procedure and listed the present knowledge about this system.

We presented only a very small fraction of the activity going on in the fascinating field of the automation of reasoning. We hope that the developed framework proves useful in other areas of the field as well.

## 7 References

Abbreviations:

| | |
|---|---|
| CADE | Conference on Automated Deduction |
| CSLI | Center for the Study of Language and Information |
| IEEE | Institute for Electrical and Electronics Engineers |
| IJCAI | International Joint Conference on Artificial Intelligence |
| JACM | Journal of the Association of Computing Machinery |
| JAR | Journal of Automated Reasoning |
| JCSS | Journal for Computer and System Sciences |
| LNAI | Lecture Notes in Artificial Intelligence |
| SIAM | Special Interest Group on the Automation of Mathematics |

| | |
|---|---|
| Aït-Kaci & Nasr 86 | H. Aït-Kaci, R.Nasr: *LOGIN: A Logic Programming Language with Built-In Inheritance.* Journal of Logic Programming, No. 3, 1986. |
| Aït-Kaci & Smolka 87 | H. Aït-Kaci, G. Smolka: *Inheritance Hierarchies: Semantics and Unification.* MCC Technical Report, 1987. |
| Andrews 68 | P.B. Andrews: *Resolution with Merging.* JACM, Vol. 15, No. 3, pp 367-381, 1968. |
| Antoniou & Ohlbach 83 | G. Antoniou, H.J. Ohlbach: *Terminator.* Proc. of 8th IJCAI, Karlsruhe 1983. |
| Bibel 81 | W. Bibel: *On Matrices with Connections.* JACM, Vol. 28, No. 4, pp 633-645, 1981. |
| Bibel 82 | W. Bibel: *Automated Theorem Proving.* Vieweg Verlag, 1982. |
| Bläsius 87 | K.H. Bläsius: Equality Reasoning Based on Graphs. SEKI Report SR-87-01, FB Informatik, University of Kaiserslautern, 1987. |
| Brachman & Schmolze 85 | R. Brachman, J. Schmolze: *An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science Vol. 9, No. 2, p. 171-216, 1985. |
| Book 82 | R.V. Book: *Confluent and other Types of Thue Systems.* JACM, Vol. 29, No. 1, pp. 171-182, 1982. |
| Brand 75 | D. Brand: *Proving Theorems with the Modification Method.* SIAM Journal of Comp., vol 4, No. 4, 1975 |
| Bundy 83 | A. Bundy: *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, 1983. |
| Chang & Lee 73 | C.-L. Chang, R.C. Lee: *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics Series (W. Rheinboldt, ed.), Academic Press, New York, 1973. |
| Chang & Slagle 79 | C.-L. Chang, J.R. Slagle: *Using Rewriting Rules for Connection Graphs to Prove Theorems.* Artificial Intelligence, Vol. 12, No. 2, pp. 159-178, 1979. |
| Church 41 | A. Church: *The Calcule of Lambda Conversion.* Princeton University Press, Princeton 1941. |
| Cohn 87 | A.G. Cohn: *A More Expressive Formulation of Many Sorted Logic.* JAR Vol. 3, No. 2, pp. 113-200, 1987. |
| Digricoli 79 | V.J. Digricoli: *Resolution by Unification and Equality.* Proc. 4th Workshop on Automated Deduction, Texas, 1979 |
| Dixon 73 | J.K. Dixon: *Z-Resolution: Theorem Proving with Compiled Axioms.* JACM, 20,1, 1973. |

| | |
|---|---|
| Eisinger 89 | N. Eisinger: *Completeness, Confluence, and Related Properties of Clause Graph Resolution.* SEKI Report SR-88-07, FB. Informatik, Univ. of Kaiserslautern, 1989. |
| Harrison & Rubin 78 | M.C. Harrison, H. Rubin: *Another Generalization of Resolution.* JACM, Vol. 25, No. 3, pp. 341.-351, July 1978. |
| Hopcroft & Ullman 79 | J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA 1979. |
| Huet 80 | G. Huet: *Confluent Reductions: Abstract Properties and Applications to Term Rewriting.* JACM, Vol. 27, NO. 4, pp. 797-821, 1980. |
| Huet & Oppen 80 | G. Huet, D.C. Oppen: *Equations and Rewrite Rules.* in: Formal Language Theory: Perspectives and open Problems (Ed. R. V. Book), Academic Press, New York, 1980. |
| Joyner 73 | W. Joyner: *Automatic Theorem Proving and the Decision Problem.* Report 7/73, Center Research Comp. Tech. Havard University, 1973. |
| Knuth & Bendix 70 | D. Knuth, P.Bendix: *Simple Word Problems in Universal Algebras.* in: Computational Problems in Abstract Algebra. (I. Leech, ed.), Pergamon Press, pp. 263 - 297, 1970. |
| Kowalski 70 | R. Kowalski: *Search Strategies for Theorem Proving.* Machine Intelligence (B. Meltzer, D. Michie eds.), Vol. 5, Edinburgh University Press, Edinburgh, pp. 181-201, 1970. |
| Kowalski 75 | R. Kowalski: *A Proof Procedure Using Connection Graphs.* JACM, Vol. 22, No. 4, 1975. |
| Kowalski & Kuehner 71 | R. Kowalski, D. Kuehner: *Linear Resolution with Selection Function.* Artificial Intelligence, Vol. 2, No. 3-4, pp. 227-260, 1971. |
| Lim & Henschen 85 | Y. Lim, L.J. Henschen: *A New Hyperparamodulation Strategy for the Equality Relation.* Proc. IJCAI-85, Los Angeles, 1985. |
| Lindström 69 | P. Lindström: *On Extensions of Elementary Logic.* Theoria 35, 1969. |
| Loveland 78 | D. Loveland: *Automated Theorem Proving: A Logical Basis.* Fundamental Studies in Computer Science, Vol. 6, North-Holland, New York, 1978. |
| Morris 69 | J.B. Morris: *E-Resolution: An Extension of Resolution to include the Equality Relation.* Proc. IJCAI, pp. 287-294, 1969. |
| Nilsson 80 | N. Nilsson: *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980. |
| Noll 80 | H. Noll: *A Note on Resolution: How to Get Rid of Factoring without Losing Completeness.* Proc. of 5th CADE, Springer LNCS, Vol 87, pp. 250-263, 1980. |
| Ohlbach 87 | H.J. Ohlbach: *Link Inheritance in Abstract Clause Graphs.* JAR, Vol. 3, No. 1, pp. 1-34, 1987. |
| Ohlbach 88 | H.J. Ohlbach: *A Resolution Calculus for Modal Logics.* Proc. of 9th CADE, Springer LNCS 310, pp. 500-516, 1988. SEKI Report SR-88-08, FB. Informatik, Univ. of Kaiserslautern, 1988. |
| Ohlbach 89 | H.J. Ohlbach: *Context Logic.* SEKI Report SR-89-08, FB. Informatik, Univ. of Kaiserslautern, 1989. |
| Ohlbach & Siekmann 89 | H.J. Ohlbach, J.H. Siekmann: *The Markgraf Karl Refutation Procedure.* SEKI Report SR-89-20, FB. Informatik, Univ. of Kaiserslautern, 1989. |
| Ohlbach 90 | H.J. Ohlbach: *Compilation of Recursive Two-Literal Clauses into Unification Algorithms.* Proc. of AIMSA-90. Albena-Varna, Bulgaria, 1990. |
| Paterson & Wegman 78 | M.S. Paterson, M.N. Wegman: *Linear Unification.* JCSS 16, pp. 158-167, 1978. |

| | |
|---|---|
| Plotkin 72 | G. Plotkin: *Building in Equational Theories.* Machine Intelligence 7, 1972. |
| Richter 78 | M. Richter: *Logikkalküle.* Leitfäden der angewandten Mathematik und Mechanik, Band 43, Teubner, Stuttgart, 1978. |
| Robinson 65 | J.A. Robinson: *A Machine-Oriented Logic Based on the Resolution Principle.* JACM, Vol. 12, No. 1, pp. 23-41, 1965. |
| Robinson 65b | J.A. Robinson: *Automated Deduction with Hyper-Resolution.* Intern. Journal of Comp. Mathematics 1, pp. 227-234, 1965. |
| Robinson & Wos 69 | G. Robinson, L.Wos: *Paramodulation and TP in First Order Theories with Equality.* Machine Intelligence 4, pp. 135-150, 1969. |
| Schmidt-Schauß 89 | M. Schmidt-Schauß: *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer LNAI 395, 1989. |
| Shieber 86 | S.M. Shieber: *An Introduction to Unification-Based Approaches to Grammar.* Stanford University, CSLI Lecture Notes, 1986. |
| Shostak 76 | R.E. Shostak: *Refutation Graphs.* Artificial Intelligence 7, pp. 51-64, 1976. |
| Shostak 78 | R.E. Shostak: *An Algorithm for Reasoning about Equality.* JACM, Vol. 21, No. 7, 1978. |
| Shostak 79 | R.E. Shostak: *A Graph-Theoretic View of Resolution Theorem Proving.* Report SRI International, Menlo Park, Ca, 1979. |
| Sibert 69 | E.E. Sibert: *A Machine-Oriented Logic Incorporating the Equality Axiom.* Machine Intelligence 4, pp. 103-133, 1969. |
| Sickel 76 | S. Sickel, *A Search Technique for Clause Interconnectivity Graphs.* IEEE Trans. on Computers C-25(8), pp. 823-835, 1976. |
| Smolka 82 | G. Smolka: *Completeness and Confluence Properties of Kowalski's Clause Graph Calculus.* Univ. Karlsruhe, Techn. Report 31/82, 1982. |
| Stickel 85 | M.E. Stickel: *Automated Deduction by Theory Resolution.* JAR, Vol. 1, No. 4, pp. 333-356, 1985. |
| Walther 87 | Ch. Walther: *A Many-Sorted Calculus Based on Resolution and Paramodulation.* Research Notes in Artificial Intelligence, Pitman Ltd., London 1987. |
| Wos et al 67 | L. Wos. D. Carson, G. Robinson, L. Shallar: *The Concept of Demodulation in Automated Theorem Proving.* JACM, Vol. 14, No. 4, pp. 698-709, 1967. |

# 8 Index