FRIEDER STOLZENBURG AND BERND THOMAS

# CHAPTER 10

# ANALYZING RULE SETS FOR THE CALCULATION OF BANKING FEES BY A THEOREM PROVER WITH CONSTRAINTS

## 1. INTRODUCTION

We show that theorem proving, logic programming and constraint solving can be combined in a straightforward manner. This is shown not only by setting up a theoretical framework, but also by a real world application: the calculation of banking fees. We tackle the problem of deciding whether such a rule set is total and deterministic. Although these questions are undecidable in general, the restricted form of the investigated rule sets makes them decidable and even tractable in practise. Experiences with our system implemented in Prolog as well as some questions one may have with the application are discussed. The success of this application is due to the combination of constraint logic programming and first-order theorem proving, based on the model elimination calculus which is in the focus of the German research program on deduction. Each approach alone is not powerful enough to find the solution in reasonable time. The procedure proposed here can easily be generalized to analyzing arbitrary rule sets.

## 2. CONSTRAINT MODEL ELIMINATION

One of the most traditional disciplines in research on automated reasoning is theorem proving. In the early days, it concentrated mainly on developing general proof procedures for predicate logic. Because of a shift towards special domain-dependent systems, automated reasoning and logic programming nowadays aim at incorporating specialized and efficient modules which are suited for handling specialized knowledge. Therefore, it seems to be rather natural to combine theorem proving and constraint reasoning.

We will combine the model elimination procedure (Loveland, 1968) and constraint logic programming (Jaffar and Maher, 1994) here. Model elimination is a special tableau method, and constraint reasoning is closely related to theory reasoning. Both topics are discussed in great detail in this enterprise; see chapters I.1.2 and I.1.3 on clausal tableaux and chapter I.2.6 on

*W. Bibel, P. H. Schmitt (eds.), Automated Deduction. A basis for applications. Vol. III*
© 1998 *Kluwer Academic Publishers. Printed in the Netherlands*

theory reasoning. This may be traced back to the work of Bibel (1982) and Stickel (1985). It is interesting to notice that the success of the application to be presented here is due to the combination of both approaches.

## 2.1. *The PTTP Technique*

Constraint logic programming (CLP) is a field of active research. There, logic programming with Horn clauses is enhanced with an interface to efficient constraint solvers. There are also approaches that combine general, non-Horn theorem proving calculi with constraint processing. For example, Bürckert (1991) introduces the calculus of constrained resolution. However, efficient implementations for these approaches still have to be developed. Here we present a method which is also well-suited for implementation: constraint model elimination (CME). This method has been introduced by Baumgartner and Stolzenburg (1995) and Stolzenburg (1998).

Model elimination (Loveland, 1968) is a linear and goal-directed calculus, thus making it close to the logic programming language Prolog. This allows us to realize model elimination in a straightforward manner on top of existing Prolog systems by using the Prolog technology theorem prover (PTTP) technique proposed by Stickel (1988). The PTTP technique compiles a problem specified in first-order logic into a (constraint) logic program, hence ready to be executed by a Prolog system.

A theorem proving or logic programming system with constraints can be implemented rapidly and easily as will be shown here. For this purpose, the constraint solving mechanisms of the underlying Prolog dialect can be used. So we actually built a PTTP based theorem prover with a very modular architecture that exploits finite domain constraints (van Hentenryck, 1989). Such a cheap and rapid implementation does not seem to be available for constrained resolution and other related approaches.

## 2.2. *Applying Finite Domain Constraints*

In the last years, researchers in the fields of artificial intelligence, logic programming and deduction systems developed many new ideas and formalisms. They succeeded in showing that their approaches are adequate at least for small examples. But nowadays the following questions often arise:

1. What are the real-world applications of these ideas?
2. Which parts of the formalisms are needed in order to express realistic problems?
3. Are the approaches efficient enough to handle large data sets?

With these questions in mind, we tried to apply our model elimination system Protein (Baumgartner and Furbach, 1994) with finite domain constraints on rule sets for the calculation of banking fees. The underlying problem is as follows: a credit institution sells stocks and shares to its costumers; the fee for them depends on their value, the customer type and other parameters. The rule set is given as a natural language description. However, we will assume that it has already been translated into first-order predicate logic.

As an example, consider the following rule given in natural language: *For every transaction made for an office in Switzerland, a constant fee of 30 SFr is charged.* (Of course, the original rule set was given in German language.) The corresponding first-order logical representation is given as follows, where $A$ stands for the actual deal:

(1) $\forall A\ (country(A,ch) \rightarrow (customer(A,office) \leftrightarrow const(A,30)))$

This formula can be represented by two Prolog rules:

(2) $const(A,30) \leftarrow country(A,ch), customer(A,office).$

(3) $customer(A,office) \leftarrow country(A,ch), const(A,30).$

We translate them into rules with (arithmetical) constraints, i.e. with equations and inequations between parameters. Constraints always occur in rule bodies only. Hence they have to be negated and moved into the body, if they have to occur in the head. This is the case for rule (3), which is translated into (5). Note that $\underline{A}$ stands for the list of input parameters of the actual deal, e.g. $\underline{A} = (Country, Customer)$.

(4) $const(\underline{A},30) \leftarrow Country = ch, Customer = office.$

(5) $\leftarrow Customer \neq office, Country = ch, const(\underline{A},30).$

Schmid (1990) investigates how the following interesting problems concerning such rule sets can be tested: *Is the rule set total, i.e., does it permit calculation of a fee for every business deal? Is it deterministic, i.e., can only one fee be computed in each case?* We tried to tackle these questions by means of our theorem proving system with constraints, especially the problem of determinism. We succeeded in finding the answer.

## 2.3. *Rule Set Verification and Software Engineering*

There is an interesting relationship between our application and automated reasoning in software engineering, which we want to discuss now. In the introduction to their book, Rich and Waters (1986) state that “the ultimate goal of artificial intelligence applied to software engineering is automatic programming”. We achieve this goal here to a certain extent, since we get a directly executable specification of a rule set, i.e. a logic program that calculates banking fees according to the given rule set. But, as said above, we want to achieve more than that, namely analyzing these rule sets wrt. critical properties. For this, it seems to be very unlikely to achieve a satisfying state of performance, due to the highly complex and domain-specific problems. One attempt to cut down the problems of automatic programming was to focus on very small and domain-specific problems. In fact, we concentrate here on a special domain: rule sets for banking fees. In addition, we exploit domain-specific knowledge by employing specialized constraint solving modules from the CLP field.

According to Rich and Waters (1986), theorem-proving techniques were used (a) for *deductive synthesis* and (b) for *program verification* in software engineering. Deductive program synthesis is based on the idea, that each step carried out by a constructive theorem prover can be seen as a computational step in a program. Assume that a program takes some input value $x$ and returns an output value $y$. Furthermore a program specification is given, and the assertions that for a precondition $P(x)$ dependent on the input value $x$ the postcondition $Q(x,y)$ holds. This can be formulated as a theorem $\forall x \exists y\ (P(x) \rightarrow Q(x,y))$ to be proven by the theorem prover, taking the given program specification into account. Let us focus on our approach for the calculation of banking fees and the connection to deductive synthesis. If we consider the rule set to be a program specification and the calculation of a specific transaction under the aspects of determinism and totality as the theorem to be proven, then the derivation or refutation tableau of this proof can be seen as an explanation for the calculation. Experiments with the tool ILF developed to generate natural language proofs from such derivations are very helpful in this context. For more information about ILF, see chapters III.1.1 and III.2.8 and the ILF manual (Dahn et al., 1995).

Program verification makes use of theorem provers in a very similar way. The main goal of program verification is to verify that a program satisfies its formal specification. For this, usually, a set of axioms describing the behavior of the statements, a precondition and a postcondition are required, where the pre- and postconditions are similar to those in program synthesis. The first step of the automatic verification is to construct one large formula by combining the precondition with the behavior of the statements contained in the program. After this formula has been constructed, a theorem prover is used to verify, whether the postcondition follows from this formula. The test for determinism of the rule set can be seen as a verification task, where the postcondition states that at most one fee is calculated for each transaction, and the precondition simply states that the transaction values are within the given domains.

In summary, our practical approach to solve the problems of determinism and totality for a special kind of rule sets is strongly related to basic areas in artificial intelligence and software engineering. Though we focus more on the logic programming and deduction based point of view, the previous mentioned aspects of this work underline the wide area of its research interest.

### 2.4. *Overview*

The plan for the rest of this chapter is as follows. At first, we will instantiate the CME framework with finite domain constraints in section 3. The reader will be able to see how simple and elegant the system can be implemented. After that, we will present our application of calculating fees and the problems we had to overcome in order to solve the interesting questions in section 4 in more detail. Finally we will give some conclusions in section 5. A preliminary version of the material presented here has been given in Stolzenburg and Thomas (1996).

## 3. THEOREM PROVING WITH CONSTRAINTS

We now introduce constraint clauses, finite domain constraints (van Hentenryck, 1989), and the CME framework, where ideas from Bürckert (1991) and Jaffar and Maher (1994) are used. Model elimination will be viewed as a tree calculus in this context. The calculus acts with constraint clauses on model elimination tableaux. For further information, the interested reader is referred to Baumgartner and Stolzenburg (1995) and Stolzenburg (1998). Readers not familiar with basic notions of logic programming should consult e.g. Lloyd (1987).

### 3.1. *Constraint Clauses and their Semantics*

DEFINITION 1. A *constraint clause* is a pair consisting of a disjunction of literals $L$ which is an ordinary clause (over a signature $\Sigma$, see below) and a constraint $R$. It is written as $L/R$ and can be understood as the implication $\forall(R \rightarrow L)$; all free variables are supposed to be universally quantified. The constraints stem from a constraint theory which consists of a set of constraint symbols $\Delta$ and a set of constraint models in which these symbols are interpreted. A constraint theory is nothing but a set of first-order structures over the signature $\Delta$.

Constraint theories will now be extended with further predicate symbols. Thus given a constraint theory and a set $\Sigma$ of predicate symbols—disjoint from $\Delta$—, we augment the constraint models by any possible denotation for these new symbols. A structure $\Im'$ for a constraint clause set $\mathcal{C}$ is given by a constraint model $\Im$ of the constraint theory for all symbols in $\Delta$, where in addition all symbols from $\Sigma$ occurring in left-hand sides $L$ of rules are interpreted by a relation with the respective arity. $\Im'$ is called $\Sigma$-*expansion* of $\Im$. The disjointness of the signatures (except for the variables) is crucial for the calculus. This allows a modular architecture of hybrid systems hosting constraint solvers.

DEFINITION 2. If an interpretation $\Im'$ together with a variable assignment $\alpha$ satisfies a literal set $L$ (read as disjunction) we write $\Im'_\alpha \models L$. See Lloyd (1987) for detailed definitions. A constraint clause $L/R$ is satisfied by $\Im'$ (written $\Im' \models L/R$) iff for all variable assignments $\alpha : V \to D$ (where $V$ is the set of the variables in $L/R$, and $D$ is the domain of the considered constraint theory) with $\Im_\alpha \models R$ it holds $\Im'_\alpha \models L$. We say that a constraint clause set $\mathcal{C}$ is satisfied by $\Im'$ iff $\Im' \models L/R$ for every clause $L/R \in \mathcal{C}$. The set $\mathcal{C}$ is called *satisfiable* iff such an $\Im'$ exists.

## 3.2. *Finite Domain Constraints*

A *constraint logic program* (also called *constraint clause set* in this context) consists of rules of the form $(L_1 \leftarrow L_2)/R$ where $L_1$ and $L_2$ are sets of positive literals (read as disjunction and conjunction of literals, respectively) and $R$ is a constraint. Quite often $L_1$ is a singleton. A rule is called *disjunctive* iff $L_1$ contains more than one literal. During a computation when goals are unfolded using constraint clauses, the constraints $R$ in their bodies are collected and tested for consistency. In the sequel, we will consider constraint solvers just as a black box.

In the case of finite domain constraints, we consider the domain of finite sets of constants. Constraint predicate symbols in this context are $=$, $\neq$, $\leq$, $>$ etc. In the case of finite integer domains, operations such as $+$ and $\cdot$ are also useful. All symbols will be interpreted as expected by their usual meaning. Mostly, only linear integer constraints of the form $a_1x_1 + \cdots + a_nx_n \leq b$ are considered where the $x_i$ with $1 \leq i \leq n$ are integer variables and the $a_i$ and $b$ are integer constants. But in general, any relation between two or more variables may be taken into consideration. Without loss of generality, we will assume that all $a_i$ with $1 \leq i \leq n$ are positive. We will specify the domain of a variable $x$ like this: $x :: 0..99$, which means $x \in \{0, 1, \ldots, 99\}$.

The constraint theory consists of only one single model for the finite domain case. This is an important property, because it simplifies our proof procedure. Every constraint theory consisting of only one model enjoys the so-called independence of negated constraints property (Maher, 1993). Because of this, it suffices to consider only one derivation (refutation) for computing answers. This means, the completeness result for CME has a simple structure in this restricted case (see section 3.3). Since we assume that all variables have only finite domains, the only model is even finite. Implementations of finite domain constraints often exploit the implication

$$\text{(6)} \qquad \text{if } \sum_{i=1}^{n} a_i x_i = b \text{ then } b - \sum_{\substack{i=1\\ i\neq k}}^{n} a_i \max(D_i) \leq a_k x_k \leq b - \sum_{\substack{i=1\\ i\neq k}}^{n} a_i \min(D_i)$$

for all $1 \leq k \leq n$ where $D_i$ is the domain for the variable $x_i$, i.e. its value range (van Hentenryck, 1989). Thus, e.g. for $x :: 0..99 \wedge y :: 0..99 \wedge 2x + 5y = 33$, we can infer the domain restrictions $x :: 4..14 \wedge y :: 1..5$ by applying (6) repeatedly. This is left as an exercise to the reader.

However it is not correct to use (6) as the only simplification rule, since not all $(x,y) \in \{4,\dots,14\} \times \{1,\dots,5\}$ are solutions, but only $(4,5)$, $(9,3)$ and $(14,1)$. These solutions can be computed by means of the (extended) Euclidean algorithm. Linear integer programming techniques are also applicable here. But for finite domain constraints, usually an algorithm which achieves only local consistency is completed by a backtrack scheme, because this seems to be cheaper than only one monolithic algorithm. In general there is a trade-off between the effectiveness and efficiency of constraint simplification. Hence constraints may be evaluated eagerly or lazily, i.e., the consistency check should be delayed, if it cannot be done efficiently.

## 3.3. *The Extended Model Elimination Calculus*

Now we want to treat the overall framework in a formal way. This is done in a tableau-like presentation. We have two inference rules in our calculus, namely (constraint) extension and reduction steps. They are used in constraint derivations. Mostly we are interested in derivations ending with an *empty tableau* $\square/R$ where $R$ is a constraint. We will represent the two basic inference rules of the model elimination extended by constraints. For a more detailed description, see Baumgartner and Stolzenburg (1995) and Stolzenburg (1998).

DEFINITION 3. Let $\mathcal{C}$ be a constraint clause set. A pair $\mathcal{P}/R$ is called *constraint tableau* iff $\mathcal{P}$ is a tableau, i.e. a finite set of finite sequences $p = \langle p_1, \dots, p_n \rangle$ of $\Sigma$-literals (which are called paths) and $R$ is a constraint. We will call a sequence of tableaux starting with the initial one $L/r$—which must be a clause in $\mathcal{C}$ where $L$ is interpreted as a set of paths of length 1—*constraint derivation* in $\mathcal{C}$ with *goal clause* $L/r$ iff in each step one of the following inference rules has been applied:

**extension.** $\dfrac{(\mathcal{P} \uplus \{p\})/R}{\mathcal{P}'/R'}$ if $(L \vee l)/r \in \mathcal{C}$

where

– $(L \vee l)/r$ is a copy of a clause in $\mathcal{C}$ with fresh variables;
– $R' = R \wedge r \wedge (p_n = \neg l)$ (see explanation below);
– $\mathcal{P}' = \mathcal{P} \uplus (p \circ L)$ where $\uplus$ denotes (multi-set) union;
– $p \circ L$ denotes the set of paths appending each literal in $L$ to $p$, i.e. $p \circ L = \{\langle p_1, \dots, p_n, l\rangle \mid l \in L\}$ provided $p = \langle p_1, \dots, p_n\rangle$.

**reduction.** $\dfrac{\mathcal{P} \uplus \{p\}/R}{\mathcal{P}/R'}$ if $l \in p$

where

– $R' = R \wedge (p_n = \neg l)$;
– $p_n = \neg l$ denotes the fact that the last literal in $p = \langle p_1, \dots, p_n\rangle$ is complementary with $l$.

The practical combination of model elimination with (finite domain) constraints is quite easy. We had a PTTP-implementation of model elimination already at hand—the Protein system (Baumgartner and Furbach, 1994)—implemented in ECLiPSe-Prolog (ECRC, 1995). Only small modifications were necessary to build the whole system because ECLiPSe features constraint solving mechanisms. The constraint predicates simply can be passed directly to the finite domain constraint solver while the other predicates are treated by our Protein theorem prover. There is a strong interaction between both components. The architecture is modular and simple. We do not need any additional control component. Only one derivation of an empty tableau is necessary in the finite domain case for computing an answer, because the constraint theory consists of only one single model as mentioned before. The next theorem states *soundness and completeness* of CME with finite domain constraints; it is a specialization of the main theorem in Baumgartner and Stolzenburg (1995).

THEOREM 4. *Let $\mathcal{C}$ be a clause set with finite domain constraints attached. $\mathcal{C}$ is unsatisfiable iff there exists one derivation of an empty tableau $\square/R$ such that $\exists R$, i.e., the existential closure of $R$ is valid in the theory of finite domain constraints.*

The definitions stated here slightly differ from those in Stolzenburg (1998), because we establish only refutational completeness of CME here, which is less than answer-completeness. Completeness can be proven by employing the lifting technique to ground proofs. The notion of a ground proof has to be adapted in our context to derivations on ground instances of constraint tableaux. In contrast to Baumgartner and Stolzenburg (1995), the proof of the lifting lemma—relating CME and corresponding ground derivation—is more difficult in the answer-completeness case. Again, the interested reader is referred to Baumgartner et al. (1997) and Stolzenburg (1998) for more details.

## 4. BANKING APPLICATIONS

At this point we are ready to come back to our application: the calculation of banking fees. Remember the problem: a credit institution sells stocks and shares to its costumers and makes a charge for them. We distinguish different kinds of data in the specification of the problem:

**input parameters:** value, country, currency, type of customer, etc.;

**background facts:** Skolem constants or mappings, e.g. conversion table for different currencies;

**output value:** the customer's fee.

The rule set will be translated into a constraint clause set. In the following we will only refer to the case of finite domain constraints. We are interested in the questions whether the rule set is total and deterministic. At first we want to discuss these problems for unrestricted rule sets and show their undecidability. After that we will introduce some reasonable restrictions which leads to tractability in practice. Finally we will compare two approaches that cope with the restricted rule sets and state some complexity results.

### 4.1. *Theoretical Problems*

In the sequel we will consider rule sets that compute fees $y$ (the output value) via a binary relation $G(x,y)$ where $x$ comprises all input parameters and background facts. We will assume that the problem specification is always given as a clause set with finite domain constraints. Although the rest of this chapter is mainly concerned with the determinism problem, we will now define two notions, which are interesting in our context.

DEFINITION 5. A binary relation $G$ is called (left) *total* iff for all possible $x$ there exists at least one $y$ with $G(x,y)$. It is called (weakly) *deterministic* iff for all $x$ there exists at most one $y$ with $G(x,y)$. $G$ is *strongly* deterministic iff the rule set allows exactly one way to compute a $y$ for each $x$. Of course, the latter notion depends on the chosen computation model.

Both kinds of determinism are interesting. In the practical application the strong variant may be desirable, whereas we will prefer the weak one because then we can tackle the problem by means of a Prolog-based theorem proving system quite easily, as we will see next. The following table gives an overview about the complexity of the problems. We will explain it in a while.

| problem | semi-decidable | strictly undecidable |
|---|---|---|
| $G$ total | yes[1] | yes[2] |
| $G$ non-total | no | yes |
| $G$ deterministic | no | yes |
| $G$ non-deterministic | yes[3] | no |

Since predicate logic is undecidable in general, we are able to conclude that the interesting problems here (totality and determinism) are undecidable too. Not both a problem and its complement can be semi-decidable. As we will see, totality may even be strictly undecidable, i.e. not even semi-decidable. Some remarks seem to be necessary to clarify things. They relate to the annotated numbers given in the table.

1. Totality is semi-decidable if we consider arbitrary models. Then we only have to show the theorem $\forall x \exists y\, G(x,y)$. This can be done by means of an ordinary theorem prover.
2. If the elements of $x$ stem from an inductive set, e.g. the Herbrand domain or the natural numbers, then the problem becomes more difficult. This can be seen by the following argument. Logic programs and Turing machines are equivalent in their computational power, and we can reduce our problem to the complement of the halting problem for Turing machines, which is known to be strictly undecidable.
3. Non-determinism turns out to be semi-decidable by the following procedure: enumerate all answers to the query $G(x,y)$ until some $x,y,y'$ are found with $G(x,y) \wedge G(x,y') \wedge y \neq y'$. This works quite well with our theorem proving environment.

$$
\begin{array}{lll}
G(A,K) & \leftarrow & const(A,K). \\
G(A,M) & \leftarrow & \neg const(A,K), min(A,M), value(A,V), \\
 & & V \leq 100 \cdot M. \\
G(A,M) & \leftarrow & \neg const(A,K), \neg country(A, oversea), max(A,M), \\
 & & value(A,V), V \geq 100 \cdot M. \\
G(A,G) & \leftarrow & \neg const(A,K), \neg min(A,Min), \neg max(A,Max), \\
 & & value(A,V), G = V/100. \\
 & & \\
const(A,0) & \leftarrow & country(A,ch), customer(A, office). \\
const(A,30) & \leftarrow & \neg country(A,ch), customer(A, office). \\
const(A,50) & \leftarrow & customer(A, personal). \\
 & & \\
max(A,10000) & \leftarrow & country(A,ch), currency(A,ch). \\
max(A,20000) & \leftarrow & country(A,ch), value(A,V), V > 1000000. \\
 & & \\
min(A,10) & \leftarrow & country(A,ch), type(A,share), value(A,V), V < 100. \\
min(A,10) & \leftarrow & country(A,ch), type(A,oblig), value(A,V), V < 100. \\
min(A,20) & \leftarrow & \neg country(A,ch), value(A,V), V < 200. \\
min(A,50) & \leftarrow & \neg country(A,ch), value(A,V), V \geq 200, V < 500. \\
min(A,80) & \leftarrow & value(A,V), V > 500.
\end{array}
$$

*Figure 1.* Small rule set.

## 4.2. *The Treatment of Restricted Rule Sets*

These undecidability results do not sound very encouraging. So it might be a good idea to lower the expressiveness of the rules. But at the same time we want to be able to state all rules for the calculation of banking fees that occur in practice. The following restrictions—see Schmid (1990) for a full list of restrictions—meet both requirements in fact, although they seem to be very strong. The interesting problems then become decidable.

1. All input parameters have finite domains. That allows us to employ finite domain constraints.
2. Recursive loops which cause infinite computations are forbidden.
3. There are no interdependencies, i.e. constraints, among any input parameters (including background facts). Hence only binary constraints between one input parameter and the output value are possible.

Incidentally, the restrictions 1 and 2 alone already suffice for achieving decidability. In fact our system is able to manage rule sets even if none of the restrictions holds, whereas the approach of Schmid (1990) is not. So these restrictions need not to be taken too serious. But our algorithm may not terminate without them.

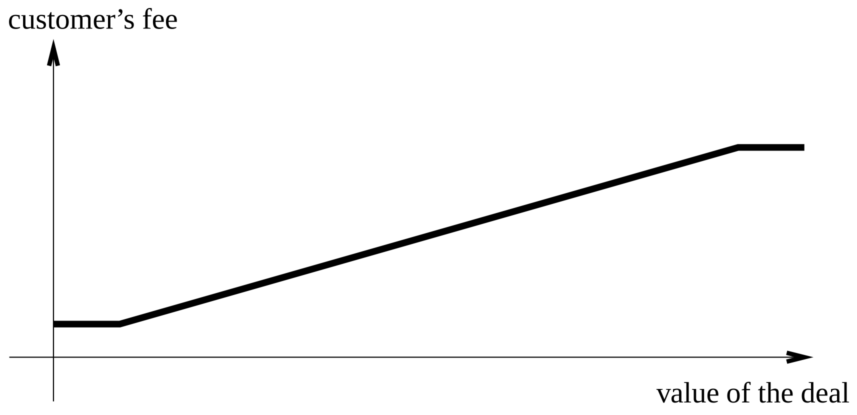

*Figure 2.* Intended meaning of the rule set.

Let us consider as an example the fragment of a rule set shown in figure 1. It is given in Prolog-like notation. As one can see, negation in the body of rules is allowed. We can interpret it as disjunctive rules, by rewriting e.g. a rule of the form $P \leftarrow \neg Q$ into $P \vee Q$. The variable $A$ serves as an identifier for the deal in question. The first four rules specify different fees depending on special features concerning a transaction $A$. The most simple calculation e.g. would be a derivation of the first rule of $G$, followed by an application of the third rule of *const*, which will lead to a calculated fee of 50, if the condition claimed in the *const* rule, namely that the customer of transaction $A$ is a staff member, holds. The intended meaning of this rule set for an "average" customer is shown in figure 2. The reader may look on his own at the rule set for a while. We may distinguish several kinds of predicates:

**features:** they are binary predicates and specify properties for the parameters of the deal in question, e.g. *country*, *currency* and *value*; these parameters have finite domains which have to be defined elsewhere.

**constraints:** these predicates, e.g. $W \geq 200, W < 500$ are the arithmetical conditions which may be treated by a finite domain constraint solver; without loss of generality, we will assume that we only have to deal with non-negative integer numbers.

**ordinary predicates:** they may be computed by further rules in the rule set; examples are $G$, *const* and *min*.

One solution to our problems is sketched by Schmid (1990). He proposes an implementation in Prolog that uses meta-programming techniques. In this approach, a rule set is processed as follows. Firstly, the program is completely unfolded with its definitions. This procedure terminates because recursive loops are forbidden. Secondly, the semantics of each ordinary predicate is computed as a cross product of the features used in its definition. Of course the constraints must be taken into consideration during this step. In addition, the negation of a predicate may be reduced to the computation of a set complement which is not difficult because of the use of finite domains. Finally we get a set-based representation for the semantics of $G$ (and all other predicates too) which can be investigated further. In order to test (strong) determinism, the intersection of the semantics of alternative rules has to be empty for each predicate.

## 4.3. *Compiling Rule Sets*

The idea here is to make use of the abilities of our Prolog-based theorem proving systems more directly. Instead of a meta-programming approach we compile the rule set into clauses with finite domain constraints. This leads to a very rapid implementation of the problem exploiting constraint techniques. Our implementation performs a transformation whose main task is to extract the constraints and features to build a constraint clause set description of the rule set. Then the theorem proving system processes it in a top-down manner while the approach of Schmid (1990) works bottom-up. Each predicate call is transformed as described below. The operators with prefix # denote constraint predicates which are processed by the constraint solver.

**features:** for example, $\neg country(A, ch)$ is rewritten as a constraint condition $Country \mathbin{\#\#} ch$, where ## means $\neq$, i.e. inequality; background facts must ensure that the value of the variable *Country* is taken from the corresponding finite domain.

**constraints:** for example, $value(A,W) \land W < Min * 100$ is replaced by the constraint condition $W \mathbin{\#<} Min * 100$.

**ordinary predicates:** they have to be transformed in a special way; their parameter list consists of a all possible features, more exactly those variables the feature predicates have been transformed into, e.g. *Country*.

We also have to define the domains of the features by background facts and the query to investigate the desired problem, e.g. determinism. The ordinary predicates in the premise are negated and added to the conclusion of the implication, this may lead to a set of disjunctive rules. Figure 3 shows the overall architecture of our application.

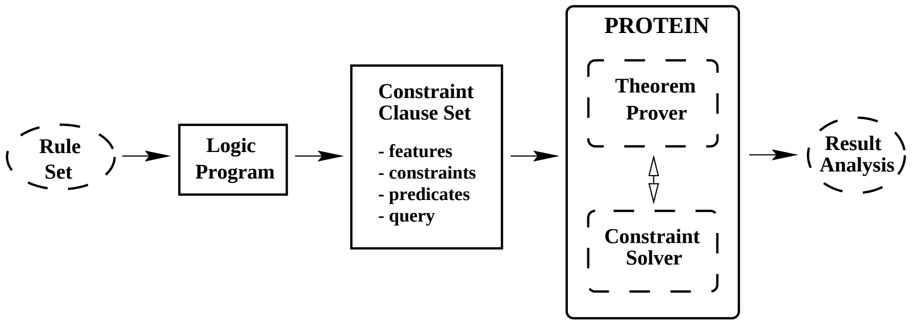

*Figure 3.* System architecture.

Let us have a closer look at our example after the transformation into a constraint clause set, which is given in figure 4. The parameter list is always abbreviated; the full list is

$$(Min, Max, Value, Country, Type, Customer, Currency, F)$$

where $F$ is the output value, i.e. the fee. Now we just have to define the finite domains of the background facts, and then we are ready to ask a query. This is the last step of the transformation, namely building the query, which is stated in figure 5 (a). The use of constraints helps us to bundle answers and to avoid backtracking. The transformed clause set can now be processed by our Prolog-based theorem prover with constraints giving the answer concerning non-determinism of the rule set, which is shown in figure 5 (b). It says that transactions with the stated features have an ambiguous rate value.

The next table lists the test procedures available in Schmid (1990) (upper half) and our approach (lower half). The totality problem seems to be more difficult than the other one. This is related to the fact that the theoretical complexity of the totality problem is higher than the one of the determinism problem. In order to test for totality, it is planned to collect all answers, which requires that the theorem prover is answer-complete (Baumgartner et al., 1997) for the latter approach.

| | |
|---|---|
| $G$ non-total | complement of a parameter non-empty |
| $G$ strongly deterministic | intersection of rule alternatives empty |
| $G$ is total | collect answers $G(x,y)$ with constraints |
| $G$ weakly deterministic | query $G(x,y), G(x,y'), y \neq y'$ fails |

$G(\ldots,F) \lor \neg const(\ldots,F)\ /\ true.$
$G(\ldots,F) \lor const(\ldots,F) \lor \neg min(\ldots,F)\ /\ Value\ \#\leq Min * 100.$
$G(\ldots,F) \lor const(\ldots,F) \lor \neg max(\ldots,F)\ /\ Country\ \#\#\ oversea,$
$\quad Value\ \#\geq Max * 100.$
$G(\ldots,F) \lor const(\ldots,F) \lor min(\ldots,F) \lor max(\ldots,F)\ /$
$\quad Value\ \#= G * 100.$

$const(\ldots,0)\ /\ Country\ \#=\ ch, Customer\ \#=\ office.$
$const(\ldots,30)\ /\ Country\ \#\#\ ch, Customer\ \#=\ office.$
$const(\ldots,50)\ /\ Customer\ \#=\ personal.$

$max(Min,10000,\ldots,F)\ /\ Country\ \#=\ ch, Currency\ \#=\ ch.$
$max(Min,20000,\ldots,F)\ /\ Country\ \#=\ ch, Value\ \#>\ 1000000.$

$min(10,\ldots,F)\ /\ Country\ \#=\ ch, Type\ \#=\ share, Value\ \#<\ 100.$
$min(10,\ldots,F)\ /\ Country\ \#=\ ch, Type\ \#=\ obligation, Value\ \#<\ 100.$
$min(20,\ldots,F)\ /\ Country\ \#\#\ ch, Value\ \#<\ 200.$
$min(50,\ldots,F)\ /\ Country\ \#\#\ ch, Value\ \#\geq\ 200, Value\ \#<\ 500.$
$min(80,\ldots,F)\ /\ Value\ \#>\ 500.$

*Figure 4.* Transformed rule set.

(a) ?- $Country :: [ch, d, usa, gb, nl, oversea],$
$\quad Customer :: [office, personal, other],$
$\quad Type :: [share, obligation],$
$\quad Currency :: [ch, d, usa],$
$\quad Value :: 0..1000000,$

$\quad F1\#\#F2,$
$\quad G(\ldots,F1),$
$\quad G(\ldots,F2),$

$\quad print\_answer([\ldots,F1,F2]).$

(b)

```
Following rates are possible : [0] or [10]
Value-Range                  : [0..99]
Country                      : [ch]
Transaction Type             : [share]
Customer                     : [office]
Currency                     : [d,ch,usa]
```

*Figure 5.* A query and its answer.

(a)

$$\begin{tikzcd}
 & G(x) \leftarrow p_1(x) \arrow[dl] \arrow[dr] & \\
p_1(c_1) \leftarrow p_2(x) & \cdots & p_1(c_z) \leftarrow p_2(x) \\
\vdots & & \vdots \\
p_{r/z}(c_1) \leftarrow q(x) \arrow[dr] & \cdots & p_{r/z}(c_z) \leftarrow q(x) \arrow[dl] \\
 & q(c) &
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
 & y & \\
x_1 \arrow[ur] & \cdots & x_n \arrow[ul]
\end{tikzcd}$$

*Figure 6.* Rule set and constraint graph.

In order to analyze determinism of a rule set, we can combine our approach (which is a strict search) with methods from (approximate) determinism analysis. These methods give an abstract interpretation of (constraint) logic programs. If the latter methods fail—and this will often be the case—, then our algorithm has to be applied. However our procedure does not perform brute-force search since the search is guided by the constraints, and computations are bundled. We also tried to tackle the problem without using finite (integer) domain constraints, but there was no chance of solving the problem within reasonable time even for small examples.

## 4.4. *Complexity Examinations*

We will now try to estimate the complexity of the procedures to detect determinism and ambiguity. We split the considerations into two parts: (a) the analysis of the number of possible (top-down) computations with non-recursive rules, and (b) the effort that must be spent to propagate the constraints provided that there are no interdependencies among the input parameters.

Remember, for the analysis of determinism we have to explore all answers to the query $G(x,y)$ in the worst case. Now, can the cost of this computation be exponential in the number of rules? The answer to this question is yes, even if we forbid recursive loops. To see this look at the rule set in figure 6 (a) where the output value is left out for the ease of notation. For each $p_k$ we have $z$ alternative rules with $1 \leq k \leq r/z$. Thus, we have $(z \cdot r/z) = r$ rules in total (ignoring both outermost rules). For each $k$ we can choose among $z$ alternatives, such that there are $z^{r/z} = (z^{1/z})^r$ possible computations. The expression becomes maximal at $z = e \approx 2.71$, or $z = 3$ for integer numbers. Hence the worst-case complexity is $(\sqrt[3]{3})^r \approx 1.44^r$, i.e. exponential in the number of rules. In real rule sets this worst-case complexity should not be approached too often. Anyway, ambiguity will be detected after at most $d^n$ computations where $n$ is the number of parameters and $d$ the domain size (assumed to be constant for all input variables), since there cannot be more output values than that number.

Let us now draw our attention to the constraint processing. Before discussing its complexity in more detail, we have to introduce some new notions.

DEFINITION 6. An instance of the *constraint satisfaction problem* requires us to instantiate a set of variables $x_1, \ldots, x_n$ subject to a set of constraints, i.e. relations involving these variables. We define *k-consistency* as follows: choose any set of $k-1$ variables along with values for each that satisfy all the constraints among them and any $k$-th variable, then there exists a value for the $k$-th variable such that the $k$ values taken together satisfy all constraints among the $k$ variables. Note that, the notion *arc-consistency* is synonymous with 2-consistency. Strong $k$-consistency means $i$-consistency for all $i \leq k$.

The finite domain solver of ECLiPSe (ECRC, 1995), used for our implementation, only applies rules similar to (6), mentioned in section 3.2. However in general, this does not always achieve global consistency. The constraints may not even be arc-consistent. But in our examples, the constraints are so simple that arc-consistency is always assured by the solver.

DEFINITION 7. A *constraint graph* is a graph where the nodes represent variables and two nodes are linked to represent the existence of a constraint that involve these variables. An ordered constraint graph arranges the nodes in a linear order. The *width* of such a graph is the maximal number of links, i.e. the number of constraints, that lead back from a node to previous ones.

Because of the restriction in our application that there are no interdependencies among the input variables, the constraint graph forms a simple tree with root $y$, i.e. the output value, with binary constraints only. This is illustrated by figure 6 (b). We want to order the variables such that we avoid as much backtracking as possible. We distinguish two different kinds of orderings: the vertical and the horizontal search order. An ordered constraint graph determines the vertical search order, i.e. the order in which variables are chosen for instantiation, whereas the horizontal order, i.e., the order in which values are tested for the given variables, is not determined. Backtracking is necessary, if an instantiation chosen during the search, which is consistent with all previous choices, must be discarded later in the search when no consistent instantiation can be made for a variable at a lower level in the backtrack tree. A search sometimes is accidently backtrack-free. But we only say that a given vertical search order is *backtrack-free* iff it guarantees a backtrack-free search regardless of the horizontal search order. The following theorem can be stated:

THEOREM 8. *For a given constraint satisfaction problem, a vertical search order is* backtrack-free*, if the level of strong consistency is greater than the width of the corresponding ordered constraint graph. There always is an ordering such that tree-structured constraint graphs have width* 1.

The theorem stated above—adapted from Freuder (1982)—allows us to compute solutions without backtracking during the constraint-processing phase, because the constraint graph has width 1, while 2-consistency is achieved. This means, we can do it without the built-in constraint predicate *indomain* (van Hentenryck, 1989) that tests possible values for variables by instantiating them, provided that we instantiate the output value $y$ first. This means, $y, x_1, \ldots, x_n$ is an optimal vertical search ordering for the constraint graph. This justifies the bundling of answers as we do in figure 5 (b). Therefore, we can avoid the expensive labeling phase, i.e. guessing values for the variables, in our application. We are only concerned with binary constraints. But of course, in principle we are not bound to this restriction. It simply speeds up our procedure.

## 5. Final Remarks

### 5.1. *Results*

It has been shown how theorem proving, logic programming and constraint processing can be combined in a straightforward and easy manner for the specification and analysis of (logic) programs. The rapid implementation of a theorem proving system using finite domain constraints is possible. We applied it to the calculation of banking fees and succeeded in testing some real examples for ambiguity within seconds. We investigated a small example (sketched in this chapter) with 15 rules (14 clauses plus 1 query) and 8 parameters sketched in figure 1, and a bigger one with 169 rules and 20 parameters including 10 disjunctive rules. The latter one was a rule set of the Swiss credit institution SKA.

| Example | rules | parameters | time |
|---|---|---|---|
| small | 15 | 8 | 0.1s |
| big | 169 | 20 | 3.5s |

The given answers concerning ambiguity were calculated by the Protein system (Baumgartner and Furbach, 1994) with the support of definite answer calculation (Baumgartner et al., 1997) on a Sparc station 4. These proof time results are very encouraging, but unfortunately we have not yet been able to tackle the problem whether a clause set is total or not in a really satisfactory way.

## 5.2. *Related Work*

There are other approaches treating similar problems, e.g. Colomb and Chung (1995) and Hoover and Chen (1995). We will discuss one of them in greater detail, namely the *Tablewise* system, a decision table tool (Hoover and Chen, 1995). This tool tests decision tables for consistency and completeness, i.e. determinism and totality according to our terminology. It also supports a special form of structural analysis that localizes flaws causing decision tables to be inconsistent or incomplete. Tablewise can generate Ada code implementing the function defined by the decision table. In our system we get the implementation for free, simply as a by-product. We just have to run the transformed Prolog code we produced with the query $G(x,y)$ and an instantiated input parameter set $x$. This means, a rule set can be directly considered as an executable specification (i.e., a restricted kind of automatic programming is possible).

Tablewise generates English language documentation describing that function. This means, the rules are first specified by means of decision tables and then translated automatically into a natural language form. This idea, namely to use formalisms like decision tables for specifying complex rules sets, seems to be very promising. The procedure may be transferred to logic programming as formalism. One of the largest tables Tablewise dealt with so far had 17 input variables, mostly boolean, with 13 operational procedures, comprising 20 to 25 different output values in all. Consistency and completeness analyses took 1 to 2 minutes depending on the system, structural analysis longer depending on the number of variables. All details are according to e-mail correspondence with Douglas Hoover.

However the domain where Tablewise is used slightly differs from ours: Tablewise was developed in a joint project with an aircraft company and to our knowledge has not yet been used for banking applications. We have a bigger number of different output values. For example, there may be a rule saying that the banking fee $y$ amounts to 1% of the value $x$ of the deal in question. This is difficult to express with decision tables whose semantics is naturally defined in terms of a (generalized) propositional logic, not in terms of full first-order predicate logic as in our approach, since in propositional logic it is impossible to express binary constraints such as

(7) $y = 0.01 \cdot x.$

One solution would be to introduce propositional variables with the meaning $x = d$ or $y = d$, respectively, where $d$ stands for a concrete value in the respective domains. Then, (7) can be expressed by the set of all instances of the rule

(8) $y = d_1 \leftrightarrow x = d_2$ where $d_1 = 0.01 \cdot d_2$.

But this is ponderous, if the domain sizes are large. Although it is possible—and also applied in practice—to make use of propositional variables meaning $x \geq d$ or a similar unary constraint, this again does not help us to encode (7), because it is a binary constraint.

Colomb and Chung (1995) emphasize that expert systems, i.e. propositional Horn clause systems can be mechanically transformed into a decision table. The transformed systems execute faster and in bounded time. The compact decision table presentation simplifies the problem of checking rule sets for totality and ambiguity, and it makes a systematic analysis of an expert system quite convenient. But since the underlying calculus is propositional the argument stated above holds here too. In addition, our approach can also cope with disjunctive, i.e. non-Horn rules.

### 5.3. *Summary*

Let us now return to the questions set up in section 2.2 and try to give some answers:

1. *What are the real-world applications of these ideas?*
   The application of calculating banking fees seems to be realistic. But the effort for coding the natural language rule set into logic with constraints may be too expensive, unless the rules change frequently so that repeated automatic tests of the rule sets are appropriate.
2. *Which parts of the formalisms are needed in order to express realistic problems?*
   There is only sparse use of disjunctive rules in our examples—less than 6%. But they do occur. It is interesting that we can dispense with recursive loops and even interdependencies among variables in practice.
3. *Are the approaches efficient enough to handle large data sets?*
   We are happy that we could solve the examples we had at hand. But it can be shown that the worst-case complexity is exponential. Thus, for

sufficient large examples, our procedure might not work. In this case, it is necessary to develop more efficient procedures, which e.g. are able to identify redundant computations or which make use of incomplete heuristics.

Last but not least the question remains whether, despite the costs involved, credit institutes or other companies will accept this approach. Many people do not want computers to solve tasks usually done by humans, because they are afraid of passing the decision-control to machines.

Nevertheless, we are able to solve some problems by means of deduction techniques exploiting constraint technology with a Prolog-based theorem proving system within reasonable time. At the moment we do not yet have more than a prototype of a system. We are currently planning to contact other partners in industry interested in our system. In summary, analyzing rule sets is a field with great potential for applying tableau-based deduction techniques and constraint solving.

## REFERENCES

Baumgartner, P. and U. Furbach: 1994, 'PROTEIN: A PROver with a Theory Extension INterface'. In: A. Bundy (ed.): *Proceedings of the 12th International Conference on Automated Deduction*. Nancy, pp. 769–773, Springer, Berlin, Heidelberg, New York.

Baumgartner, P., U. Furbach, and F. Stolzenburg: 1997, 'Computing answers with model elimination'. *Artificial Intelligence* **90**(1-2), 135–176.

Baumgartner, P. and F. Stolzenburg: 1995, 'Constraint Model Elimination and a PTTP-Implementation'. In: P. Baumgartner, R. Hähnle, and J. Posegga (eds.): *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*. pp. 201–216, Springer, Berlin, Heidelberg, New York.

Bibel, W.: 1982, *Automated Theorem Proving*. Braunschweig, Wiesbaden: Vieweg.

Bürckert, H.-J.: 1991, *A Resolution Principle for a Logic with Restricted Quantifiers*, LNAI 568. Berlin, Heidelberg, New York: Springer.

Colomb, R. M. and C. Y. C. Chung: 1995, 'Strategies for Building Propositional Expert Systems'. *Journal of Intelligent Systems* **10**, 295–328.

Dahn, B. I., J. Gehne, T. Honigmann, L. Walther, and A. Wolf: 1995, 'Integrating Logical Functions with ILF'. Humbold-Universität, Berlin. Manual.

ECRC: 1995, 'ECLiPSe 3.5: User Manual – Extensions User Manual'. ECRC GmbH, München. Two volumes.

Freuder, E. C.: 1982, 'A Sufficient Condition for Backtrack-Free Search'. *Journal of the ACM* **29**(1), 24–32.

Hoover, D. N. and Z. Chen: 1995, 'Tablewise, a Decision Table Tool'. In: *Proceedings of the 10th Annual Conference on Computer Assurance*. Washington, pp. 97–108, IEEE.

Jaffar, J. and M. J. Maher: 1994, 'Constraint Logic Programming: a Survey'. *Journal of Logic Programming* **19/20**, 503–581.

Lloyd, J. W.: 1987, *Foundations of Logic Programming*. Berlin, Heidelberg, New York: Springer.

Loveland, D. W.: 1968, 'Mechanical Theorem Proving by Model Elimination'. *Journal of the ACM* **15**(2).

Maher, M. J.: 1993, 'A Logic Programming View of CLP'. In: D. S. Warren (ed.): *Proceedings of the 10th International Conference on Logic Programming*. Budapest, pp. 737–753, MIT Press, Cambridge, MA, London.

Rich, C. and R. C. Waters (eds.): 1986, *Artificial Intelligence and Software Engineering*. Los Altos, CA: Morgan Kaufmann.

Schmid, K.-M.: 1990, 'Eindeutigkeit und Vollständigkeit zweistelliger Relationen – Entscheidungsverfahren für eingeschränkte Fälle'. Diplomarbeit, Universität Kaiserslautern. Supervised by Hans-Jürgen Ohlbach.

Stickel, M. E.: 1985, 'Automated Deduction by Theory Resolution'. *Journal of Automated Reasoning* **1**(3), 333–355.

Stickel, M. E.: 1988, 'A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler'. *Journal of Automated Reasoning* **4**, 353–380.

Stolzenburg, F.: 1998, *Disjunctive Logic Programming with Constraints and its Applications*, Koblenzer Schriften zur Informatik. Koblenz: Fölbach. Dissertation.

Stolzenburg, F. and B. Thomas: 1996, 'Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints'. In: *Proceedings of the 2nd International Conference on Practical Application of Constraint Technology*. London, pp. 269–282, Practical Application Company.

van Hentenryck, P.: 1989, *Constraint Satisfaction in Logic Programming*. Cambridge, MA, London: MIT Press.