Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints

Frieder Stolzenburg, Bernd Thomas
Universität Koblenz, Institut für Informatik
Rheinau 1, D-56075 Koblenz, Germany
Phone: +49+261–9119–426, Fax: +49+261–9119–496
Email: {stolzen,bthomas}@informatik.uni-koblenz.de

Abstract

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 problems of deciding whether such a rule set is total and deterministic, i.e. does it permit calculation of a fee for every business deal, and can only one fee be computed in each case. 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.

Keywords: banking applications; constraint logic programming; finite domain constraints; theorem proving.

1 Introduction

One of the most traditional disciplines in artificial intelligence research is theorem proving. In the early days, it was 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.

This led to the very active research field of constraint logic programming [Jaffar and Maher, 1994]. Here, 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. [Bürckert, 1991] for example introduces the calculus of constrained resolution. However, efficient implementations for this approach still have to be developed.

1.1 Constraint Model Elimination

[Baumgartner and Stolzenburg, 1995] presents a framework for constraint model elimination along the lines of [Bürckert, 1991]. *Model elimination* [Loveland, 1968] is a linear and goal-directed calculus, thus making it close to the logic programming language Prolog. That allows us to realize model elimination in a straightforward manner on top of existing Prolog systems by using the so-called *Prolog technology theorem prover* (PTTP) technique proposed in [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 shown in this paper. To 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 available for constrained resolution.

1.2 An Application with Finite Domain Constraints

During recent decades, researchers in the fields of artificial intelligence and logic programming 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 really 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 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.

[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 out the answer.

1.3 Overview of the Rest

The plan for the rest of the paper is as follows: At first, we will instantiate the framework of constraint model elimination with finite domain constraints. 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. Finally we will give a résumé.

2 Theorem Proving with Finite Domain Constraints

Next we will introduce constraint clauses, finite domain constraints [Van Hentenryck, 1989], and the framework for constraint model elimination along the lines of [Bürckert, 1991]. Model elimination will be viewed as a tree calculus. The calculus acts with constraint clauses on model elimination tableaux. For full details the interested reader is referred to [Baumgartner and Stolzenburg, 1995]. Readers not familiar with basic notions of logic programming should consult e.g. [Lloyd, 1987].

2.1 Constraint Clauses and their Semantics

Definition 1 (constraints) A *constraint clause* is a pair consisting of a disjunction of literals L which is an ordinary clause and a constraint R. It is written as L/R and can be understood as the implication $R \Rightarrow L$. All variables are supposed to be universally quantified. The constraints stem from a *constraint theory* which consists of a set of constraint symbols and a set of constraint models in which these symbols are interpreted. Up to now a constraint theory is nothing else but a set of first-order structures over a signature Δ.

Constraint theories will now be extended with further predicate symbols. Thus given a constraint theory and a set Σ of predicate symbols – disjoint from Δ – we augment the constraint models by any possible denotation for these new symbols. A structure $\mathcal{A}^*$ for a constraint clause set $\mathcal{C}$ is given by a constraint model $\mathcal{A}$ for all constraints R, where in addition all symbols from Σ occurring in left-hand sides L are interpreted by a relation with the respective arity. – The disjointness of the signatures (except for the variables) is crucial for the calculus. This is why constraint systems are modular.

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

2.2 Finite Domain Constraints

Constraint logic programming is often described as logic programming with unification replaced by constraint solving over a computation domain. A *constraint logic program* 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 disjunctions) and R is a constraint. Quite often L_1 is a singleton. A rule is called *disjunctive* iff L_1 contains more than on literal. During a computation when goals are unfolded using constraint clauses, the constraint(s) R in their body are collected and tested for consistency.

In the case of *finite domain constraints* [Van Hentenryck, 1989] used e.g. in the Chip system, we consider the domain of finite sets of constants. Possible 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. Mostly, only *linear integer constraints* of the form $a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq b$ are considered where the x_i $(1 \leq i \leq n)$ 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.

The constraint theory consists of only one single model for the finite domain case. (All symbols shall be interpreted as expected by their usual meaning here.) This is an important property because it simplifies our proof procedure as we will see later. Since we assume that all variables have only finite domains, the model is even finite. We will specify the domain of a variable x like this: $x : 0..99$ which means $x \in \{0, 1, \ldots, 99\}$.

Implementations of finite domain constraints [Van Hentenryck, 1989] exploit the implication

$$\sum_{i=1}^{n} a_i \cdot x_i = b \Longrightarrow b - \sum_{\substack{i=1 \\ i \neq k}}^{n} a_i \cdot \max(D_i) \leq a_k \cdot x_k \leq b - \sum_{\substack{i=1 \\ i \neq k}}^{n} a_i \cdot \min(D_i) \qquad (+)$$

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

However it is not correct to use this implication as the only simplification rule, since not all $(x, y) \in \{4, \ldots, 14\} \times \{1, \ldots, 5\}$ are solutions, but only $(4, 5)$, $(9, 3)$ and $(14, 1)$. These solutions could 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.

2.3 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. Due to the lack of space we will just introduce both inference rules. For a more detailed description see [Baumgartner and Stolzenburg, 1995].

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

extension. $\dfrac{(P \uplus \{p\})/R}{P'/R'}$ 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;
- $R' = R \wedge r \wedge (\mathcal{L}(p) = \bar{l})$ (see explanation below);
- $P' = P \uplus (p.L)$ where $\uplus$ denotes (disjoint) union;
- $p.L$ denotes the set of paths appending each literal in L to p,
  i.e. $p.L = \{\langle p_1, \ldots, p_n, l \rangle \mid l \in L\}$ provided $p = \langle p_1, \ldots, p_n \rangle$.

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

- $R' = R \wedge (\mathcal{L}(p) = \bar{l})$;
- $\mathcal{L}(p) = \bar{l}$ denotes the fact that the last literal in p is complementary with l,
  i.e. p_n is complementary with l.

The combination of model elimination plus 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 only have to 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 for the finite domain case because the constraint theory consists of only one single model as mentioned before. The next theorem states the completeness of constraint model elimination with finite domain constraints; it is a specialization of the main theorem in [Baumgartner and Stolzenburg, 1995].

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

3 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 shall be given as 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.

3.1 Theoretical Problems

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

Definition 5 (total ; deterministic) A 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.

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 afterwards.

problem	recursively enumerable	essentially undecidable
G total	yes*	yes†
G non-total	no	yes
G deterministic	no	yes
G non-deterministic	yes‡	no

Since predicate logic is undecidable in general, it is clear that the interesting problems are undecidable too. That implies in addition that not both a problem and its complement can be recursively enumerable. Some remarks seem to be necessary to clarify things:

(*) Totality is recursively enumerable 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.

(†) If the elements of x stem from an inductive set, e.g. the Herbrand domain or the natural numbers, then the problem becomes much 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 essentially undecidable.

(‡) Non-determinism turns out to be recursively enumerable by the following procedure: enumerate all answers to the query $G(x,y)$ until some x, y, y' are found with $G(x,y) \wedge G(x,y') \wedge y \neq y'$. That works quite well with our theorem proving environment.

3.2 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 – set up in [Schmid, 1990] – meet both requirements in fact, although they seem to be very strong. The interesting problems then become decidable.

- All input parameters have finite domains. That allows us to employ finite domain constraints.
- Recursive loops which cause infinite computations are forbidden.
- 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.
- In [Schmid, 1990] other technical restrictions are mentioned which we will not discuss here.

Incidentally, one of the former two restrictions alone already suffices for achieving decidability. In fact our system is able to manage rule sets even if none of the restrictions holds, whereas the approach in [Schmid, 1990] is not. So these restrictions need not to be taken too harshly. But our algorithm may not terminate without them.

Let us consider now the fragment of a rule set shown further down as an example. It is given in a Prolog-like notation. As one can see, negation in the body of rules is allowed. We can interpret it as disjunctive rules; but another view is possible, as we will see soon. – The variable A serves as an identifier for the deal in question.

$$\begin{array}{lcl}
G(A,K) & \leftarrow & const(A,K). \\
G(A,M) & \leftarrow & \neg const(A,K), min(A,M), value(A,W), \\
 & & W \leq 100 \cdot M. \\
G(A,M) & \leftarrow & \neg const(A,K), \neg country(A, oversea), max(A,M), \\
 & & value(A,W), W \geq 100 \cdot M. \\
G(A,G) & \leftarrow & \neg const(A,K), \neg min(A, Min), \neg max(A, Max), \\
 & & value(A,W), G = W/100.
\end{array}$$

$$\begin{array}{lcl}
const(A,0) & \leftarrow & country(A,ch), customer(A, office). \\
const(A,30) & \leftarrow & \neg country(A,ch), customer(A, office). \\
const(A,50) & \leftarrow & customer(A, personal).
\end{array}$$

$$\begin{array}{lcl}
max(A,10000) & \leftarrow & country(A,ch), currency(A,ch). \\
max(A,20000) & \leftarrow & country(A,ch), value(A,W), W > 1000000.
\end{array}$$

$$\begin{array}{lcl}
min(A,10) & \leftarrow & country(A,ch), type(A, share), value(A,W), W < 100. \\
min(A,10) & \leftarrow & country(A,ch), type(A, obligation), value(A,W), W < 100. \\
min(A,20) & \leftarrow & \neg country(A,ch), value(A,W), W < 200. \\
min(A,50) & \leftarrow & \neg country(A,ch), value(A,W), W \geq 200, W < 500. \\
min(A,80) & \leftarrow & value(A,W), W > 500.
\end{array}$$

The reader may look on his own at the rule set for a while. The first four rules specify alternative 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. – 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 and 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. Examples are G, $const$ and min.

One solution to our problems is sketched in [Schmid, 1990]. An implementation in Prolog is proposed that uses meta-programming techniques. In this approach, a rule set is processed as follows: Firstly, the program is unfolded completely 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 only finite domains. Finally we get a set-based representation for the semantics of G (and all other predicates) which can be investigated further. To test determinism, the intersection of the semantics of alternative rules has to be empty for each predicate.

3.3 Compiling Rule Sets

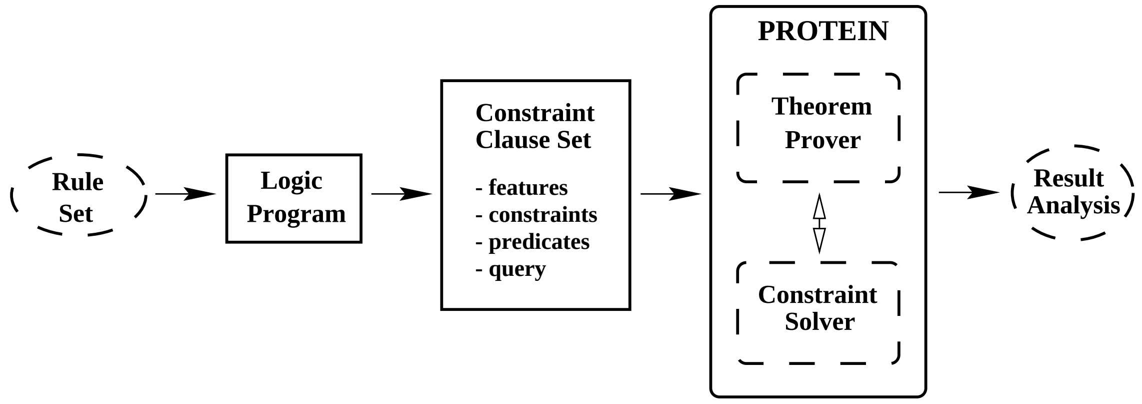
The idea in this paper is to make use of the abilities of our Prolog-based theorem proving systems more directly. Instead of a meta-programming approach we compile the rule set into clauses with finite domain constraints. This leads to a very rapid implementation of the problem exploiting constraint 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: $\neg country(A, ch)$ e.g. is rewritten as a constraint condition $Country \#\# ch$. (Here $\#\#$ means $\neq$, i.e. inequality.) Background facts must make sure that the value of the variable $Country$ is taken from the corresponding finite domain.

constraints: For example, $value(A, W) \wedge W < Min * 100$ e.g. is replaced by the constraint condition $W \#< Min * 100$.

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. The following figure shows the overall architecture of our application:



Let us now have a closer look at our example after the transformation into a constraint clause set. The parameter list is always abbreviated; the full list is

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

where F is the output value, i.e. the fee.

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

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

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

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

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:

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

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

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

The use of constraints helps us to bundle answers, and they avoid backtracking. The transformed clause set can now be processed by our Prolog-based theorem prover with constraints giving the following answer concerning non-determinism of the rule set. It says that transactions with the stated features have an ambiguous rate value.

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

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. For the latter approach, it is planned to collect all answers which requires that the theorem prover is answer-complete [Baumgartner *et al.*, 1995] in order to test for totality.

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

In order to analyse determinism of a rule set, we could combine our approach which essentially does search with methods from (approximate) determinism analysis, i.e. 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.

3.4 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: the analysis of the number of possible (top-down) computations with non-recursive rules, and 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 $G(x, y)$. Now, can the cost of this computation then 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 stated below 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).

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

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 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 can not be more output values than that number.

Let us now draw our attention to the constraints processing. The finite domain solver of ECLiPSe that we use only applies rules equivalent to the implication (+) stated earlier, obviously.

However in general, this does not always achieve satisfactory consistency. The constraints may not even be arc-consistent. (See below for a definition of the notions just used.) But in the examples we tried, the constraints are so simple such that arc-consistency can always be assured by the solver.

Definition 6 (consistency ; graphs) A *constraint satisfaction problem* requires us to instantiate the set of variables $x_1, \ldots, x_n$ subject to a set of constraints, i.e. relations involving the variables. k*-consistency* implies the following: 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. *arc-consistency* is synonymous with 2-consistency.
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 (constraints) that lead back from a node to previous ones.

Because of the restriction that there are no interdependencies among the input variables, the constraint graph forms a tree with root y, i.e. the output value. This can be illustrated by the following figure.

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

Theorem 7 (backtrack-freeness) *Given a constraint satisfaction problem, a 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 (taken 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. That means, we can do it without the built-in constraint predicate $indomain$ [Van Hentenryck, 1989] provided we instantiate the output value first. This justifies the bundling of answers we do. In addition we are only concerned with binary constraints. But of course, in principle we are not bound to these restrictions. It just speeds up our procedure.

4 Final Remarks

4.1 Results

In this paper, it is shown how theorem proving, logic programming and constraint processing can be combined in a straightforward and easy manner. 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 with 15 rules (14 clauses plus 1 query) and 8 parameters sketched in this paper, 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. Look at the following table.

Example	rules	parameters	time	depth
small	15	8	0.05s	3
big	169	20	7.2s	4

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.*, 1995] on a Sparc station 2. 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.

4.2 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 that cause 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.

Tablewise generates English language documentation describing that function. That 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. This 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. Thus the performance looks quite good.

However the domain where Tablewise is used slightly differs from ours: Tablewise was developed on a joint project with an aircraft company and to our knowledge not yet 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 amounts to 1% of the value of the deal in question. This should be 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.

[Colomb and Chung, 1995] emphasizes 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 systemic 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.

4.3 Résumé

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

(1) 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. Perhaps tax laws are better suited.
However we must represent the rule set in some formalism anyway, if we want to analyse it by means of computers, and it surely is not obvious why the translation into logic should be harder than into any other computation formalism.

(2) 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. However one rule is discussed in [Schmid, 1990] which does not meet the latter condition, but on the same page in this paper a reformulation is proposed which fits into the framework. But again, we are not really bound to this restriction.

(3) We are happy that we could solve the examples we had at hand. But it has been shown that the worst case complexity is exponential. Thus for sufficient large examples, our procedure might not work. Maybe another algorithm has to be developed for that purpose. But the worst case probably will never occur in practice.

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. It is dubious whether credit institutes are really interested in examining their rule set.

Nevertheless, from another point of view we are content. We are able to solve some problems by means of artificial intelligence 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.

Acknowledgements

Many thanks to Jürgen Dix, Douglas Hoover, Walter Hower, Hans-Jürgen Ohlbach and Graham Wrightson for comments on this paper and helpful discussions.

References

[Baumgartner and Furbach, 1994] Peter Baumgartner and Ulrich Furbach. PROTEIN: A PROver with a Theory Extension INterface. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction, Nancy, France, June/July 1994*, pages 769–773. Springer, Berlin, Heidelberg, New York, 1994. LNAI 814.

[Baumgartner and Stolzenburg, 1995] Peter Baumgartner and Frieder Stolzenburg. Constraint model elimination and a PTTP-implementation. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 201–216. Springer, Berlin, Heidelberg, New York, 1995. LNAI 918.

[Baumgartner *et al.*, 1995] Peter Baumgartner, Ulrich Furbach, and Frieder Stolzenburg. Model elimination, logic programming and computing answers. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 335–340, Montréal, August 1995. Volume 1.

[Bürckert, 1991] Hans-Jürgen Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*. LNAI 568. Springer, Berlin, Heidelberg, New York, 1991.

[Colomb and Chung, 1995] Robert M. Colomb and Charles Y. C. Chung. Strategies for building propositional expert systems. *Journal of Intelligent Systems*, 10:295–328, 1995.

[ECRC, 1995] ECRC GmbH, München. *ECLiPSe 3.5: User Manual – Extensions User Manual*, December 1995.

[Freuder, 1982] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[Hoover and Chen, 1995] Douglas N. Hoover and Zewei Chen. Tablewise, a decision table tool. In *Proceedings of the 10th Annual Conference on Computer Assurance*, pages 97–108, Washington, 1995. IEEE.

[Jaffar and Maher, 1994] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[Lloyd, 1987] John Wylie Lloyd. *Foundations of Logic Programming*. Springer, Berlin, Heidelberg, New York, 1987.

[Loveland, 1968] Donald W. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15(2), 1968.

[Schmid, 1990] Klaus-Martin Schmid. Eindeutigkeit und Vollständigkeit zweistelliger Relationen – Entscheidungsverfahren für eingeschränkte Fälle. Diplomarbeit, Universität Kaiserslautern, 1990. Supervised by Hans-Jürgen Ohlbach.

[Stickel, 1988] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Van Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, London, England, 1989.