Model Generation Theorem Proving with Interval Constraints*

Reiner Hähnle†
Institute for Logic, Complexity and Deduction Systems
Dept. of Comp. Sci., Univ. Karlsruhe, 76128 Karlsruhe, Germany
haehnle@ira.uka.de http://i12www.ira.uka.de/~reiner

Ryuzo Hasegawa
Dept. of Electronics, Kyushu Univ. 36, Fukuoka 812, Japan
hasegawa@ele.kyushu-u.ac.jp

Yasuyuki Shirai
Information Technologies Development Dept.
Mitsubishi Research Institute, Inc.
Ohtemachi 2-3-6, Chiyoda-ku, Tokyo 100, Japan
shirai@mri.co.jp

*This report has been submitted for publication elsewhere and will be copyrighted if accepted. It is published simultaneously as a Technical Report of the Department of Electronics, Kyushu University 36, Fukuoka 812, Japan.

†This co-author's contribution in the results described in the present paper was mainly done during a stay as visiting researcher at the Institute for New Generation Computer Technology (ICOT), Tokyo, Japan. The generous support provided by ICOT is gratefully acknowledged.

Abstract

We investigate how the deduction paradigm of *model generation theorem proving* can be enhanced with interval- and extraval-based constraints leading to more efficient model generation in for some finite domain problems.

1 Model Generation

Model generation (MG) is a sound and complete inference rule for first-order predicate logic for input in conjunctive normal form (CNF). One can view it as a positive literal restriction of clausal semantic tableaux. Manthey & Bry [9] gave a concise implementation of a variant of model generation in Prolog.

In logic programming and deductive databases it is common to impose the range-restrictedness condition on first-order clauses:[1]

Definition 1 *A first-order clause $C \rightarrow D$ is* **range-restricted** *if all variables that occur in D occur also in C.* ∎

Observe that a range-restricted positive clause must be ground. In the range-restricted case model generation can be defined as follows:[2]

Definition 2 *Let $S = \{C_1, \ldots, C_m\}$ be a first-order CNF formula. A set $\mathcal{M}$ of* **model candidates**[3] *for S is inductively defined as a minimal family of sets of ground atoms that obeys the following conditions:*

Initialization $\{\} \in \mathcal{M}$.

Model extension *If $M \in \mathcal{M}$ is not rejected, $(C \rightarrow D) \in S$, ($D = \{d_1, \ldots, d_k\}$, $k \geq 1$), σ is a substitution, $C\sigma \subseteq M$, and $D\sigma \cap M = \emptyset$, then $\{M \cup \{d_1\}, \ldots, M \cup \{d_k\}\} \subseteq \mathcal{M}$ ($C = \emptyset$ is allowed). Call M* **extendable**. *If $D\sigma \cap M \neq \emptyset$ then D is said to be* **subsumed** *by M. Whenever $C\sigma \subseteq M$ holds we say that $D\sigma$ is derived from M and S with* **conjunctive matching**.

Model rejection *If $C \rightarrow \perp \in S$[4], σ a substitution, and $C\sigma \subseteq M \in \mathcal{M}$, then mark M as* **rejected**.

A set of **models** *for S are those members of a set of model candidates for S that are neither extendable nor rejected.* ∎

[1] We employ rule notation $p_1, \ldots, p_n \rightarrow q_1; \ldots; q_m$ for first-order clauses of the form $\{\neg p_1, \ldots, \neg p_n, q_1, \ldots, q_m\}$.

[2] The following definition is a slightly more precise variant of the definition in [2].

[3] In connection with the implementation of model generation also the expression **term memory** is sometimes used for a model candidate.

[4] The symbol $\perp$ denotes falsity that is $D = \emptyset$.

Soundness and completeness of MG may now be formulated as follows: a range-restricted first-order CNF formula S is satisfiable iff the set of models for S is non-empty.

Restricting the input for model generation to range-restricted clauses has two important advantages that are being reflected in Definition 2 above: first, it is easy to formulate a backtracking-free MG proof search procedure, in other words an AND search tree is sufficient; second, matching of clauses is sufficient as opposed to unification. Therefore, an efficient implementation of MG called MGTP (*M*odel *G*eneration *T*heorem *P*rover) [2, 5] in KL1 is possible. KL1 is a committed choice concurrent logic programming language [11] developed at ICOT and has just the required features.

One of the applications of MGTP is the search for models in finite domains, for instance in finite quasigroups [3]. Experiments in such domains showed that merely admitting positive ground atoms in the model candidates results in quite large models which in turn lead to a search space of infeasible size already for relatively small examples. It is, therefore, desirable to allow literals in the model candidates which, while still being ground, represent more than one domain element; in other words, one should work with atomic ground *constraints*. This idea is implemented in CMGTP (*C*onstraint MGTP) [6] for constraints that represent negative information as described in the following section.

2 Model Generation with Negative Constraints in CMGTP

The constraints that are elements of model candidates in CMGTP take on the following form:

- $p(X, Y, Z)$ meaning that the value of $X \circ Y$ is equal to Z, where $\circ$ is a binary operation on a finite domain;
- $\neg p(X, Y, Z)$ meaning that the value of $X \circ Y$ is unequal to Z.

Observe that *formally* a constraint with negative sign in CMGTP is still handled as a positive atom, that is, $\neg p(X, Y, Z)$ is just a name for the constraint expressing $X \circ Y \neq Z$ (to make this clearer one could use a predicate name like np instead of $\neg p$). In particular, conjunctive matching needs not be changed.

In order to make use of such constraints, several extensions must be made to the rules of standard MGTP:

1. A model candidate in which both $p(a, b, c)$ and $\neg p(a, b, c)$ are present must not be generated. When MGTP checks for rejection of a model-extending candidate it does this on both sides of the arrow anyway, so here no real need for a change arises.

2. The main point, however, is: newly generated disjuncts D in the model-extending candidate set as well as in the current model candidate may contain negative constraints. Even when the current model candidate is neither rejected nor the newly generated disjunct is subsumed, it might still be possible to *simplify* either. For instance, if $\neg p$ occurs in the current model candidate it can be used to simplify a disjunction (a model extending candidate) $D = D_1 \vee p \vee D_2$ to $D_1 \vee D_2$. As in the newly generated disjunctions themselves negative constraints may occur, the case where p and $\neg p$ are switched around must be checked as well. When D happens to be of length 1 then D itself may additionally be used to simplify other model-extending candidates.

The model generation process of CMGTP is displayed schematically in Figure 1. For each model candidate there exists one process. Besides the set of input clauses S there are four kinds of objects: (i) the current model candidate M: a set of atomic ground constraints; (ii) from M and S newly generated disjunctions some of which may be unit clauses (the non-units are immediately checked for simplification by M, see item (2) above, the units for possible refutation of M); (iii) the simplified newly generated disjunctions are added to the pool D of model extending candidates. When one of the D's is selected for model extension, the current process forks into $|D|$ many subprocesses in each of which (iv) one of the $\Delta \in D$ gets the status of extending literal. According to item (2) above, Δ can be used to simplify the D's or to refute M before being added to M.

3 IV-MGTP: Model Generation with Interval and Extraval Constraints

As we are working with finite domains let us assume the domain N is of the form $\{1, 2, \ldots, n\}$. The task of the constraints in the model candidates is to represent the possible values a function still can take on at a given argument at a certain point of the model generation process. For example, $\neg p(2,4,3)$ says that the value of $2 \circ 4$ is an element from the set $N \backslash \{3\}$. Arbitrary sets can be represented by conjunctively combining such constraints, but it is a natural idea to use a more efficient representation of finite sets than positive and negative constraints. Such an efficient representation would be *lists of intervals*. Essentially the same execution model as for negative constraints can be used.

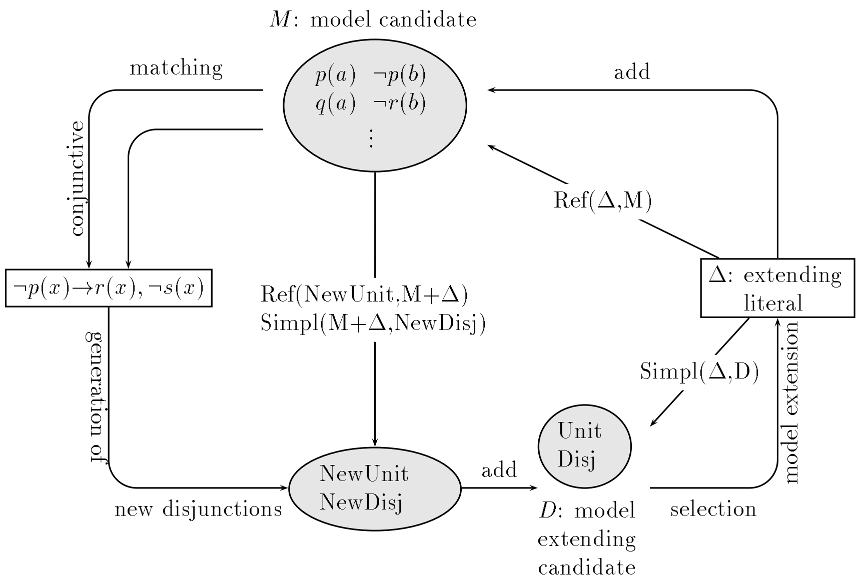

Figure 1: Model generation process of CMGTP.

In the following we give the details of a version of MGTP called **IV-MGTP** in which model candidates essentially are interval-based constraints. As before in CMGTP the following questions must be adressed:

1. Which constraints appear in input clauses?
2. How are constraints represented internally?
3. How and when is model rejection performed?
4. How is conjunctive matching done?
5. Which disjunctions are kept?
6. How and when is simplification performed?

3.1 Constraint Language of Input

In the following we assume that for each right-functional $m+1$-place predicate `p/(m+1)` over a finite domain N (where $p(X_1, \ldots, X_m, Z)$ typically expresses that the result of applying an m-ary operation to $X_1, \ldots, X_m$ is Z) there is an m-place *function* with the same name. Let α give the arity of each such function.

Definition 3 *A* **signed clause** *is a clause of the form*

$$S_1\, p_1(t_{11}, \ldots, t_{1\alpha(p_1)}), \ldots, S_k\, p_k(t_{k1}, \ldots, t_{k\alpha(p_k)}) \rightarrow \\ S_{k+1}\, p_{k+1}(t_{(k+1)1}, \ldots, t_{(k+1)\alpha(p_{k+1})}); \ldots; S_m\, p_m(t_{m1}, \ldots, t_{m\alpha(p_m)})$$

where $S_i \subseteq N$. It is **satisfiable** *iff there are $c_1, \ldots, c_m \in N$ such that*

$$p_1(t_{11}, \ldots, t_{1\alpha(p_1)}, c_1), \ldots, p_k(t_{k1}, \ldots, t_{k\alpha(p_k)}, c_k) \rightarrow \\ p_{k+1}(t_{(k+1)1}, \ldots, t_{(k+1)\alpha(p_{k+1})}, c_{k+1}); \ldots; p_m(t_{m1}, \ldots, t_{m\alpha(p_m)}, c_m)$$

is satisfiable in the usual sense. The constituents of signed clauses are called **signed literals**. ■

Thus a signed literal $S\, p(X_1, \ldots, X_n)$ simply abbreviates the disjunction $\bigvee_{i \in S} p(X_1, \ldots, X_n, i)$.

Example 4 Assume $p(X, Y, Z)$ says that the result of an operator $\circ$ over N on X and Y is Z. Injectivity of $\circ$ can be expressed as follows:

```
{I} p(X,Y), dom(X1), {X1\=X, S1=N\{I}} -> S1 p(X1,Y).
```

Here, `dom` is a unary predicate saying that its argument is in the domain of p; the expressions in curly brackets are primitives over finite sets which, for the moment, can assumed to be part of KL1.

There are many ways of expressing the same fact. For instance, using only intervals and extravals as signs one may equivalently write:

```
[I,I] p(X,Y), dom(X1) -> ]I,I[ p(X1,Y).
```
or
```
[I,I] p(X,Y), dom(X1) -> ]1,I[ p(X1,Y); ]I,n[ p(X1,Y).
```
■

Although the declarative meaning of all formulations in the above example is identical the operational semantics is quite different: in the last formulation the search for a model is split into two cases while in the first two formulations it is not. This gives the programmer means to control the structure of the search space.

We chose to admit intervals and extravals as admissible constraints (or signs) in the input. Recall that any finite set can be represented as an intersection of finitely many extravals, see also the next section. The constraints themselves may contain variables which must observe the range-restrictedness condition.

3.2 Representation of Constraints

In principle one could simply take signed ground literals as members of model candidates. This is not very efficient, however, because KL1 has no built-in primitives for manipulation of finite sets. We chose to represent the constraint S of $S\ p$ as a finite, minimal, ordered list of extravals that is

$$S = \{\,]i_1, j_1[, \ldots,]i_{m_p}, j_{m_p}[\, \} \ ,$$

where $1 \leq i_1$, $i_k \leq j_k$, $j_k < i_{k+1} + 1$, $j_{m_p} \leq |N|$ for all $1 \leq k \leq m_p$. Thus a model candidate M is a partial function from the set of ground atoms to ordered lists of extravals. If $M(p)$ is undefined p is assumed to have the constraint $N = [1, n]$.

We chose to use lists of extravals rather than lists of intervals, because during the model generation process the constraint S associated with a ground atom p is conjunctively combined with constraints S', S'' etc, when M is updated with new information. Thus extravals are efficient when the term memory is updated frequently, because the intersection of two extravals again is an extraval.[5]

$M(p)$ corresponds to the *set of domain elements which p can assume in the current model candidate.* It is given by

$$M(p) = \bigcap_{k=1}^{m_p}]i_k, j_k[\ .$$

Depending on which is the more convenient we sometimes prefer to conceive $M(p)$ as a list of extravals and sometimes as a set of domain elements. In the actual implementation, however, $M(p)$ is always a list of extravals.

[5]Below we will see, however, that for certain operations such as conjunctive matching or simplification interval representations are more natural. As extraval-interval conversion induces a certain overhead, it is specific to the problem domain which kind of representation should be actually chosen depending on whether model candidate updates or conjunctive matching and simplification dominates. See Section 4 for an example, where this consideration becomes relevant.

From a minimal extraval representation $\emptyset \neq M(p) = \bigcap_{k=1}^{m_p}]i_k, j_k[$ it is trivial to compute an interval representation denoted with $M'(p)$:

$$M'(p) = [1, i_1{-}1] \cup \left(\bigcup_{k=2}^{m_p} [j_{k-1}{+}1, i_k{-}1] \right) \cup [j_{m_p}{+}1, m] \tag{1}$$

If either $i_1 = 1$ or $j_{m_p} = m$ then the first, respectively, the last disjunct is dropped.

3.3 Model Rejection

Given a model candidate M and a signed literal (that is a unit clause) $S\,p$, where S is either an interval or an extraval, we say that M **rejects** $S\,p$ iff $M(p)$ is defined and $M(p) \cap S = \emptyset$.

In view of the equality[6] $[i, j] =]1, i{-}1[\cap]j{+}1, n[$ (for $1 < i, j < n$; if $i = 1$ or $j = n$ drop the extraval involving i, respectively, j) it suffices to conjoin extraval lists with extraval lists in order to check for model rejection.

3.4 Conjunctive Matching

Assume we have a rule of the form $S_1\,p_1, \ldots, S_m\,p_m \rightarrow C$ in our database. Then one derives $C\sigma$ with **conjunctive matching** provided that the current model candidate M defines constraints for ground atoms $\{p'_1, \ldots, p'_m\}$ such that

1. there is a substitution σ with $\{p'_1, \ldots, p'_m\} = \{p_1, \ldots, p_m\}\sigma$ and
2. $M(p'_i) \subseteq S_i$ for all $1 \leq i \leq m$

and the S_i are ground. If one admits variables in the constraints for reasons of efficiency and flexibility, then it is slightly more complicated to compute the disjunctions, because uninstantiated constraints represent disjunctive information: in general, one derives $C\sigma$ with *conjunctive matching* provided that the current model candidate M defines constraints for ground atoms $\{p'_1, \ldots, p'_m\}$ such that

1. there is a substitution σ with $\{p'_1, \ldots, p'_m\} = \{p_1, \ldots, p_m\}\sigma$ and
2. there is a substitution ρ with $M(p'_i) \subseteq S_i\rho$ for all $1 \leq i \leq m$

[6]This is essentially the dual of (1).

For each literal $[I', J']\,p$ in the antecedent the variables I', J' must be instantiated such that all intervals in $M'(p') = \bigcup_{k=1}^{m'_p} [i'_k, j'_k]$ are covered. The obvious choice is to set $I' = i'_1$ and $J' = j'_{m'_p}$.

Extraval constraints $Sp =]I, J[p$ in the antecedent are more complicated: if $M(p') = \bigcap_{k=1}^{m'_p}]i_k, j_k[$ then *for each* $1 \leq k \leq m'_p$ letting $I = i_k$ and $J = j_k$ is sufficient for $M(p'_i) \subseteq S\rho$ to hold.

This means that when extraval constraints $S_1\,p_1, \ldots, S_r\,p_r$ appear in the antecedent of a clause C up to $\Pi_{k=1}^{r} m_{p_k}$ disjuncts can be generated (where $m_p \leq \frac{n}{2}$ for all p), but typically most of them are subsumed or simplified, see the following sections. Also (for example, in the quasi-group problems) antecedents tend to be short, typically $r \leq 2$, and often $m_p \ll n$.

3.5 Subsumption

Only disjunctions that are not subsumed by the current model candidate are kept in the lists `NewUnit` and `NewDisj`. For negative constraints Definition 2 needs not to be changed. For intervals and extravals, however, it must be adapted as follows:

Let D be a disjunction of signed literals. Define

$$d(D, p) = \bigcup_{\substack{S \subseteq N \text{ and } S\,p \\ \text{occurs in } D}} S$$

Then a model candidate M **subsumes** D iff M defines $M(p)$ such that $M(p) \subseteq d(D, p)$.

This definition takes into account that the same atom can occur multiply with different signs within the same disjunction.

3.6 Simplification

One possibility of simplification is a direct generalization of the case of negative constraints described in Section 2:

Given a signed literal $S\ p$ and a disjunction $D = D_1 \vee S'\ p \vee D_2$, then D can be simplified to $D' = D_1 \vee D_2$ provided that $S \cap S' = \emptyset$. $S\ p$ can come from two sources (cf. Figure 1): either the current model candidate defines $M(p) = S$ or $S\ p$ is an extending literal Δ from the selected model extending candidate. Thus a similar functionality as for model rejection is required for this possibility and its integration into the model generation process is exactly as in Section 2.

There is, however, a second possibility for simplification which can only occur in the more general case: assume, for example, $M(p) = \{1,2,4\}$ in the current model candidate M and $\Delta = S\,p = \{2,3\}\,p$. Then Δ is neither subsumed by M nor does it reject M. In this case $M(p)$ must be updated to $M(p) \cap S = \{2\}$. Similarly, newly derived disjunctions $D = D_1 \vee S'\,p \vee D_2$ can be simplified to $D' = D_1 \vee (S \cap S')\,p \vee D_2$ provided that neither $S \cap S' = \emptyset$ nor D is subsumed by M. Finally, the current model candidate M itself can be used to simplify newly derived disjunctions in exactly the same way.

We call this second possibility of simplification **reduction** after a similar rule used in annotated logic programming [7]. The result of the reduction operation, namely $M(p) \cap S$, respectively, $(S \cap S')\,p$ is called a **residue.**[7]

The overall control schema must be augmented with additional possibilities for reduction denoted with `Red/2`. This is depicted in Figure 2. The simplification (or updating) of model candidates by extending literals is achieved by the call Red(Δ,M) while the simplification of newly derived disjunctions by extending literals or the current model candidate is simultaneously achieved by the call Red(M+Δ,NewDisj).

4 Results from Experiments

We have developed an IV-MGTP prototype system by modifying CMGTP, and made experiments on finite domain constraint satisfaction problems.

4.1 IV-MGTP Prototype System

The model generation cycle of IV-MGTP is basically the same as that of CMGTP. For IV-MGTP, however, conjunctive matching, simplification, reduction, and subsumption processes are extended in order to manipulate signed literals. The term memory for quick retrieval of literals is also extended to maintain the signed literals in M using extraval lists.

In CMGTP, the criterion for selecting Δ from D is defined so as to prefer atoms to disjunctions. If no atom is in D, CMGTP selects a disjunction of the least length. In contrast to this IV-MGTP selects a literal such that

[7]Observe that in contrast to the subsumption test, where all occurrences $d(D,p)$ of an atom in D are considered simultaneously, residues are computed separately for each literal in D. This reflects the rationale that the user of IV-MGTP should have control over the branching degree of disjunctions. A different approach would merge the occurrences of p to $d(D,p)$, simplify $d(D,p)$ and then split it up again according to a fixed and complete splitting strategy.

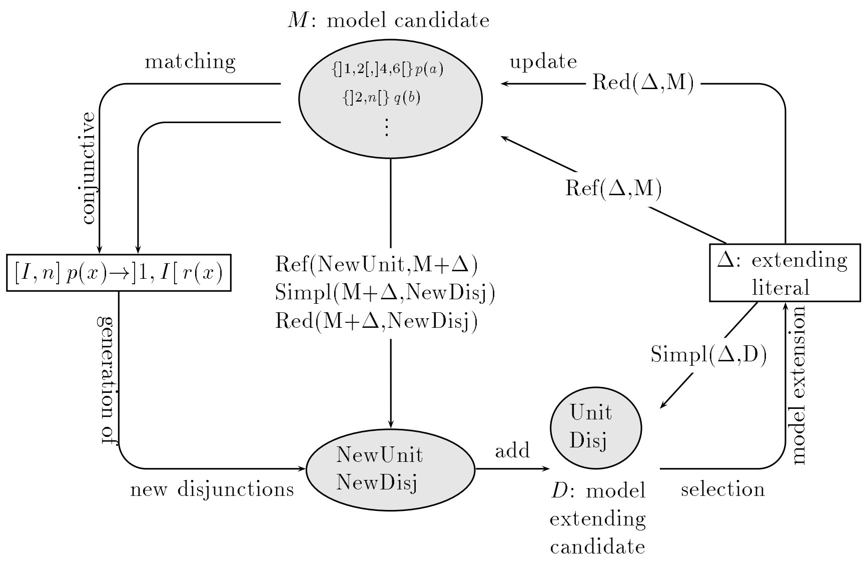

Figure 2: Model generation process of IV-MGTP.

the sum of its interval lengths is the least. This criterion can be changed according to the problem to be solved.

Both systems are written in KL1 and run on a KLIC system (portable KL1 running on a UNIX machine). The experiments described in this paper were made on a SPARCstation20.

4.2 Problems and Results

4.2.1 Quasigroup Existence Problems

Quasigroup existence problems (QG problems) [1, 10] in finite algebra are typical finite-domain constraint satisfaction problems, which attract a world-wide attention for being combinatorially explosive yet good benchmarks.

In 1992, MGTP succeeded in solving several open QG problems on a parallel inference machine PIM/m consisting of 256 processors [3]. Later, other theorem provers or constraint solvers such as DDPP, FINDER, Eclipse, CMGTP solved other new open problems more efficiently than the original MGTP.

In order to solve QG problems it is essential to prune as many redundant

$\circ$	1	2	3	4	5
1	1	3	2	5	4
2	5	2	4	3	1
3	4	5	3	1	2
4	2	1	5	4	3
5	3	4	1	2	5

Figure 3: Latin square (order 5).

branches as possible using the constraint propagation mechanisms. For this, CMGTP provides negative constraint propagation by introducing negative atoms and incorporating simplification processes.

QG problems can be defined as finding models or showing that no model exists for Latin squares which satisfy some additional constraints. The multiplication table of a binary operation $\circ$ defined on a quasigroup QG forms a Latin square. Figure 3 shows an example of a Latin square of order 5.

QG problems are classified according to the additional constraints. One usually uses codes for classes of QG problems: $QG1, QG2, \ldots, QG7$, each of which is defined by adding some constraints to the bare Latin square constraints.

For instance, the additional constraint for $QG5$ is as follows:

QG5 : $\forall XY \in Q.\ ((YX)Y)Y = X$

Since the Latin square shown in Figure 3 satisfies the QG5 constraint, QG5 has at least one idempotent solution for an order of 5.

The QG5 constraint, $((YX)Y)Y = X$, can be expressed with rules as follows:

$$\begin{array}{lll}
Y \circ X = A, \quad A \circ Y = B & \rightarrow & B \circ Y = X. \\
Y \circ X = A, \quad B \circ Y \neq X & \rightarrow & A \circ Y \neq B. \\
A \circ Y = B, \quad B \circ Y \neq X & \rightarrow & Y \circ X \neq A.
\end{array}$$

The last two rules are contrapositives of the first rule. We can write the above rules in IV-MGTP syntax as follows:

```
[A,A] p(Y,X), [B,B] p(A,Y) -> [X,X] p(B,Y).
[A,A] p(Y,X), ]X,X[ p(B,Y) -> ]B,B[ p(A,Y).
[B,B] p(A,Y), ]X,X[ p(B,Y) -> ]A,A[ p(Y,X).
```

Table 1: The experimental results for QG problems

		IV-MGTP		CMGTP	
order	models	runtime(sec)	branches	runtime(sec)	branches
7	3	25.9	2	3.9	2
8	1	35.8	8	5.3	9
9	0	66.1	15	9.9	16
10	0	225.7	52	33.4	54
11	5	2102.1	167	192.6	173
12	0	5562.3	320	455.5	324

where `p(X,Y)` denotes $X \circ Y$.

Table 1 compares the experimental results (runtime and the number of failed branches) for the QG5 problem on IV-MGTP and CMGTP systems.

4.2.2 Cryptarithmetics

We experimented with the well-known cryptarithmetic problem $SEND + MORE = MONEY$. This problem is to find models for the variables $\{D, E, M, N, O, R, S, Y\}$ which satisfy the following equation:

$$\begin{array}{rccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

To solve this problem, for example, we need the following rule:

$$D + E = 10 \times r + Y$$

where r ranges over $\{0, 1\}$, while D, E, Y range over $\{0, 1, \ldots, 9\}$. Constraint propagation from D, E, r to Y can be represented as an IV-MGTP rule:

```
[D1,D2] v(D), [E1,E2] v(E), [r1,r2] v(r) ->
                                [D1+E1-r2,D2+E2-r1] v(Y).
```

As can be seen here, the domain of `Y` can be calculated with the minimum and maximum values of the variables `D,E,r`. IV-MGTP can handle such domain calculation defined by the above rule using intervals/extravals

Table 2: The experimental results for Cryptarithmetics

	IV-MGTP		CMGTP	
models	runtime(sec)	branches	runtime(sec)	branches
1	3.69	13	62.05	1735

and extended conjunctive matching. For CMGTP, however, since the system lacks the notion of a variable domain, one has to represent possible values which a variable may take with different literals. Thus, in CMGTP constraint propagation cannot be implemented with domain calculation. Table 2 compares IV-MGTP and CMGTP for cryptarithmetic.

4.3 Discussion

For QG problems, IV-MGTP has almost the same pruning (constraint propagation) ability as CMGTP. Positive and negative atoms in CMGTP are represented in IV-MGTP as intervals and extravals, respectively. This is because, unlike as for cryptarithmetic, for QG problems no additional constraints are propagated by domain calculation using an *interval* or an *extraval*. The slight difference in the number of failed branches is caused by the different case splitting strategy used.

For cryptarithmetics, however, the comparison of the number of failed branches generated by CMGTP and IV-MGTP exhibits that the pruning effect for IV-MGTP, which supports the domain calculation, is considerable. As CMGTP (or MGTP) cannot propagate constraints by domain calculation, they generate a large number of redundant branches.

Regarding performance, as IV-MGTP as yet is a prototype, there is ample room for improvements. The main reason why IV-MGTP is less *time* efficient for QG problems is that in the current implementation, the constraint representation with intervals and/or extravals is only permitted on one argument of a predicate, denoting a value of a function. Thus, we have to use extra predicate symbols to represent constraint propagation done through the inverse functions.

For example, in CMGTP $p(a, b, c)$ means $a \circ b = c$, but also $b \bullet c = a$ and $c \diamond a = b$, where $\bullet$ and $\diamond$ are the inverse operations of $\circ$. This is represented in IV-MGTP using three different predicates f, g, h, such as in $[c, c]\ f(a, b)$, $[a, a]\ g(b, c)$, $[b, b]\ h(c, a)$, where g, h denote the inverse functions of f. This renders the size of term memory needed for QG problems in IV-MGTP much bigger than that in CMGTP, thus lengthening the model generation cycle.

Another determining factor of performance concerns manipulation of intervals and extravals. The currently implemented version of IV-MGTP retains extraval lists in the term memory so that interval-extraval conversion is needed frequently in conjunctive matching, subsumption testing, etc. For cryptarithmetics, however, no extraval occurs in the problem description, hence such conversions should be needless if the term memory maintains literals in interval form.

From the viewpoint of parallelization, IV-MGTP can potentially suppress redundant case-splitting, because it can maintain the set of candidates for a variable using a single signed literal, while in MGTP these candidates are represented by a disjunction which causes case-splitting when it is selected as Δ. Whenever decreasing the number of case-splits for a variable is essential for solving a problem, the signed literal-representation would prevent the disjunction from being blindly case-split.

5 Related Work

It is possible to interpret functions over finite domains as many-valued predicates. From this point of view there are considerable similarities between IV-MGTP and annotated logic programs [7], signed formula logic programs [8] as well as signed CNF formulas [4]. Then the main differences between IV-MGTP and the mentioned formalisms are: (i) IV-MGTP input is range-restricted; (ii) we consider intervals and extravals whereas the others either consider only upsets and downsets or totally arbitrary signs; (iii) we admit variables in the signs, this is also investigated in [7], but under quite different premisses; (iv) we use model generation, a bottom-up inference rule, whereas the others either use variants of SLD-resolution or unrestricted resolution.

The automatic simplification and reduction of model candidates and disjunctions we employ is much more sophisticated than similar stratagems in the resolution procedures mentioned in the previous paragraph. On the other hand, with the exception of the straightforward instantiation of constraint variables described in Section 3.4 all computations over constraints in IV-MGTP are performed on the ground level. In other words, we do not constraint solving, but merely constraint simplification. In languages such as Eclipse or CLP($\mathcal{FD}$) constraints may contain variables, and the used deduction paradigm is not model generation, but SLD-resolution plus constraint solving. The bulk of work is often done by the constraint solver and not by the logical inference machine. In our case, all can be done by conjunctive matching and relatively simple calls to KL1's built-in arithmetic. The advantage is an efficient, highly parallel implementation which admittedly aims at less generality than CLP.

6 Conclusion and Further Work

We have proposed IV-MGTP, in which interval and extraval constraints are effectively handled within the CMGTP framework. IV-MGTP manipulates domains for variables as the attributes of predicates. Extended conjunctive matching for signed literals makes it possible to enhance the constraint processing ability significantly.

The main advantage of this approach is that various kinds of constraint propagation mechanisms can be implemented just by giving different constraint propagation rules in forward reasoning style.

It depends on the problem domain how beneficial the effect of interval/-extraval constraints on performance is. For problems, where the ordering of the domain elements has no significance, such as the elements of a QG problem (whose numeric elements are considered strictly as symbolic values, not arithmetic values), CMGTP and IV-MGTP have essentially the same pruning effect. However, where reasoning about the arithmetic relationship between the elements is important, such as in cryptarithmetics, IV-MGTP outperforms CMGTP.

Currently, the most severe restriction in IV-MGTP is that interval/extraval constraints are limited to one argument of a predicate. If we want to describe more complicated constraints, we may need an extended framework in which one can admit interval/extraval constraints for the other arguments as well.

In order to improve time efficiency, we have to investigate the term memory which maintains signed literals using lists of extravals. As we saw in section 4, the most efficient internal expression depends on the description of the problem to be solved.

References

[1] F. Bennett. Quasigroup identities and Mendelsohn designs. *Canadian Journal of Mathematics*, 41:341–368, 1989.

[2] H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In K. Furukawa, editor, *Proceedings 8th International Conference on Logic Programming, Paris/France*, pages 535–548. MIT Press, 1991.

[3] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. Int. Joint Conf. on Art. Intelligence*, 1993.

[4] R. Hähnle. Exploiting data dependencies in many-valued logics. *Journal of Applied Non-Classical Logics, to appear*, 1995.

[5] R. Hasegawa, M. Koshimura, and H. Fujita. MGTP: A parallel theorem prover based on lazy model generation. In D. Kapur, editor, *Proc. 11th International Conference on Automated Deduction*, pages 776–780. Springer LNAI 607, 1992.

[6] R. Hasegawa and Y. Shirai. Constraint propagation of CP and CMGTP: Experiments on quasigroup problems. In *12th Conference on Automated Deduction CADE, Nancy/France, Proc. of Workshop on Automated Reasoning in Algebra*, 1994.

[7] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Jornal of Logic Programming*, 12:335–367, 1992.

[8] J. J. Lu. Logic programming with signs and annotations. Technical report, Bucknell University, Lewisburg/PA, USA, 1995.

[9] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proceedings 9th Conference on Automated Deduction*, pages 415–434. Springer LNCS, New York, 1988.

[10] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 1993.

[11] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, Dec. 1990.