ICOT Technical Report :TR-911

TR- 911

Two Approaches for Finite-Domain Constraint Satisfaction Problems -CP and CMGTP-

Y. Shirai & R. Hasegawa

February, 1995

I C O T Mita Kokusai Bldg. 21F
4 28 1 Chome
Minato-ku Tokyo 108 JAPAN
Tel(03)3456-3191～5

Institute for New Generation Computer Technology

Two Approaches for Finite-Domain Constraint Satisfaction Problems – CP and CMGTP –

Yasuyuki Shirai
Ryuzo Hasegawa
Institute For New Generation Computer Technology
Mita Kokusai Building 21F, Mita 1-4-28, Minato-ku, Tokyo, JAPAN
{shirai,hasegawa}@icot.or.jp

Abstract

We have developed two types of systems; CP and CMGTP, for finite domain constraint satisfaction problems. CP is based on the constraint logic programming scheme, and is written in SICStus Prolog. CP has achieved high performance on quasigroup (QG) existence problems in terms of the number of branches and execution time. CP succeeded in solving a new open quasigroup problem. On the other hand, CMGTP is a slightly modified version of our theorem prover MGTP (Model Generation Theorem Prover), enabling negative constraint propagation. CMGTP has exhibited the same pruning ability as CP for QG problems. CMGTP can be used as a general constraint solver for finite-domains on which we can write down constraint propagation rules with CMGTP input clauses directly.

1 Introduction

Quasigroup (QG) existence problems[2] in finite algebra are typical finite-domain constraint satisfaction problems, which have a reputation for being combinatorially intensive.

Several attempts have been made to solve open quasigroup problems. Through this research it was shown that the theorem-proving approach was really useful for finite algebra.

In 1992, M. Fujita and J. Slaney[7] first succeeded in solving several open QG problems such as QG5.9, QG5.10, QG5.12, and other open problems in QG6, QG7 by using FINDER [3] on a Sparc workstation and MGTP[1] on a parallel inference machine PIM/m consisting of 256 processors.

In 1993, M. Stickel succeeded in solving some problems of higher order by using his DDPP prover, which is an efficient Davis-Putnam procedure augmented with discrimination trees for quick retrieval of clauses. DDPP solved QG3.12, QG4.12, and QG5.13–15.

Later in 1993, M. Wallace et.al. showed that a constraint logic programming system CHIP[4] can also solve QG problems efficiently. Remarkably, the problem description for solving QG problems was very simple and easy

∘	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 1: Latin square (order 5)

to write, and the number of failed branches were much smaller than with MGTP or FINDER.

Such research has revealed that the original MGTP lacks the ability to propagate negative constraints. This motivated us to develop an experimental system we called CP (Constraint Propagation) based on the CLP (constraint logic programming) scheme. CP succeeded in solving an open problem QG5.16 in 21 days using a Sparc-10[8].

Through experiments with CP, we found what kinds of constraint propagation mechanisms are required in solving QG problems (or other finite-domain constraint satisfaction problems), and that these constraint propagation mechanisms can be introduced into MGTP with slight modification.

The new modified MGTP system is called CMGTP (Constraint MGTP). CMGTP treats negative atoms, and has the ability to perform unit refutation and unit simplification with those negative atoms.

In this paper, we introduce these two systems and show their effectiveness in solving QG problems.

2 Definition of Quasigroup(QG) Problems

First, we briefly introduce quasigroup (QG) problems.

2.1 Quasigroup

A Quasigroup is a pair $\langle Q, \circ \rangle$ where Q is a finite set, $\circ$ a binary operation on Q and for any $a, b, c \in Q$,

$$a \circ b = a \circ c \Rightarrow b = c$$
$$a \circ c = b \circ c \Rightarrow a = b.$$

The multiplication table of this binary operation $\circ$ forms a Latin square (shown in Fig.1).

A quasigroup $\langle Q, \circ \rangle$ is called *idempotent* iff $\forall x \in Q. x \circ x = x$.

2.2 Inverse Operations

For any x and y in a Latin square, there exist unique values of z_1 and z_2 such that $x \circ z_1 = y$ and $z_2 \circ x = y$. With these features (the *distinctness property* for each row and column on the Latin square), we can define the following inverse operations $\circ_{ijk}$:

$$\begin{array}{lcl} x \circ_{123} y = z & \Longleftrightarrow & x \circ y = z \\ x \circ_{231} y = z & \Longleftrightarrow & y \circ z = x \\ x \circ_{312} y = z & \Longleftrightarrow & z \circ x = y \end{array}$$

Inverse operation $\circ_{ijk}$ is called *(ijk)-conjugate*. The (123)-conjugate is the same as the original quasigroup. Multiplication tables of the inverse operations defined above also form Latin squares. Inverse operations play a significant role in solving QG problems.

2.3 QG Problems

We have been trying to solve 7 categories of QG problems (called QG1, QG2,..., QG7), each of which is defined by adding some constraints to the original quasigroup constraints. QG problems are existence problems in Latin squares which satisfy some specific constraints. The additional constraints for each QG problem are shown below:

QG1 : $\forall abcd \in Q.\ (ab = cd \wedge a \circ_{321} b = c \circ_{321} d) \Rightarrow (a = b \wedge c = d)$
QG2 : $\forall abcd \in Q.\ (ab = cd \wedge a \circ_{312} b = c \circ_{312} d) \Rightarrow (a = b \wedge c = d)$
QG3 : $(\forall abcd \in Q.\ (ab)(ba) = a) \wedge$
$(\forall abcd \in Q.\ (ab = cd \wedge a \circ_{213} b = c \circ_{213} d) \Rightarrow (a = b \wedge c = d))$
QG4 : $\forall ab \in Q.\ (ab)(ba) = b$
QG5 : $\forall ab \in Q.\ ((ba)b)b = a$
QG6 : $\forall ab \in Q.\ (ab)b = a(ab)$
QG7 : $\forall ab \in Q.\ a(ba) = (ba)b$

Since the Latin square shown in Fig.1 satisfies the condition of QG5, QG5 has at least one idempotent solution for an order of 5.

There are still many open QG problems; for example, QG5 of order n ($n \geq 17$) has never been solved.

The characteristics of QG problems are summarized as follows:

- QG problems can be defined as finite-domain constraint propagation problems.
- QG problems bring serious combinatory explosions. A simple generate-and-test strategy needs N^{N^2} size search space for order N.
- A complete search is needed because in QG we need to find all solutions if any, otherwise we need to show that no solutions exist.

3 CP

CP is a CLP based constraint propagation system written in SICStus Prolog. In this section, we first discuss approaches using ordinary CLP languages, and then explain the constraint propagation mechanism in CP. We also show the results of our experiments with CP.

3.1 Approaches using CLP languages

It seems to be very natural to represent the constraint propagation rules for QG problems in CLP languages, where domain variables are assigned to each cell in a Latin square to store a result of multiplication. The constraint propagation rules for QG5 can be represented as follows :

```
constraint(Y,X,Sq1,Sq2,Sq3) :-
        mult(Y,X,Sq1,A), mult(A,Y,Sq1,B),
        mult(Y,X,Sq2,B), mult(Y,B,Sq2,A).
```

where `Sq1`, `Sq2`, `Sq3` denote the (123)-, (231)- and (312)-conjugate Latin squares respectively. Using the conjugate Latin squares enables constraint propagation when either `A` or `B` is instantiated.

`mult(X1,X2,Sq,X3)` calculates `X1` $\circ$ `X2` on the square `Sq` when both `X1` and `X2` are instantiated, unifying the result with `X3`, which is a domain variable.

CLP can propagate constraints between domain variables when these variables are unified. For example, the constraints for the variable `A` in the above program can be propagated not only from `mult(Y,X,Sq1,A)` but also from `mult(Y,B,Sq2,A)`. This means the domains of two variables must be equal if they are unified. CLP, however, cannot propagate some negative constraints between different conjugate squares, such as :

$$\forall abc.\ (a \circ_{123} b \neq c \Leftrightarrow b \circ_{231} c \neq a \Leftrightarrow c \circ_{312} a \neq b) \quad (1),$$

because domain variables representing $a \circ_{123} b$, $b \circ_{231} c$, $c \circ_{312} a$ are different and CLP has no way to propagate negative constraints between different domain variables. CLP can propagate constraints between different squares when at least one of these 3 variables is instantiated as follows:

$$\forall abc.\ (a \circ_{123} b = c \Leftrightarrow b \circ_{231} c = a \Leftrightarrow c \circ_{312} a = b) \quad (2).$$

Negative constraint propagation as shown in (1) is significant because it can reduce the number of candidates for each domain variable. To do this efficiently, CP introduces *domain element variables* w.r.t. each domain variable. Domain element variables in different conjugate squares can be linked (unified) with each other.

S–square :
(123)-conjugate

o	1	2	3
1	V_{11} $\{A_1\ A_2\ A_3\}$	V_{12} $\{B_1\ B_2\ B_3\}$	V_{13} $\{C_1\ C_2\ C_3\}$
2	V_{21} $\{D_1\ D_2\ D_3\}$	V_{22} $\{E_1\ E_2\ E_3\}$	V_{23} $\{F_1\ F_2\ F_3\}$
3	V_{31} $\{G_1\ G_2\ G_3\}$	V_{32} $\{H_1\ H_2\ H_3\}$	V_{33} $\{I_1\ I_2\ I_3\}$

I–square :
(231)-conjugate

o	1	2	3
1	W_{11} $\{A_1\ D_1\ G_1\}$	W_{12} $\{A_2\ D_2\ G_2\}$	W_{13} $\{A_3\ D_3\ G_3\}$
2	W_{21} $\{B_1\ E_1\ H_1\}$	W_{22} $\{B_2\ E_2\ H_2\}$	W_{23} $\{B_3\ E_3\ H_3\}$
3	W_{31} $\{C_1\ F_1\ I_1\}$	W_{32} $\{C_2\ F_2\ I_2\}$	W_{33} $\{C_3\ F_3\ I_3\}$

R square :
(312) conjugate

o	1	2	3
1	U_{11} $\{A_1\ B_1\ C_1\}$	U_{12} $\{D_1\ E_1\ F_1\}$	U_{13} $\{G_1\ H_1\ I_1\}$
2	U_{21} $\{A_2\ B_2\ C_2\}$	U_{22} $\{D_2\ E_2\ F_2\}$	U_{23} $\{G_2\ H_2\ I_2\}$
3	U_{31} $\{A_3\ B_3\ C_3\}$	U_{32} $\{D_3\ E_3\ F_3\}$	U_{33} $\{G_3\ H_3\ I_3\}$

Figure 2: The variables in conjugate Latin squares

3.2 Variable Maintenance in CP

Let V be a domain variable whose domain is $\{1,\ldots,n\}$ and $\{A_1,\ldots,A_n\}$ be a vector of domain element variables w.r.t. V. Each domain element variable A_k represents the possibility of V taking the value k. That is, if a domain element variable A_k is bound to *yes*, V's value is fixed to k; if bound to *no*, V should not take the value of k; and if it remains unbound, V may take the value of k. From the finite-domain property, if $n-1$ variables of $\{A_1,\ldots,A_n\}$ excluding A_k are bound to *no*, A_k is bound to *yes* and variable V is bound to k.

Fig.2 shows the variables of (123)-, (231)- and (312)-conjugate Latin squares for order 3, where $V_{ij}(1 \le i,j \le 3)$ are domain variables with the range $\{1,2,3\}$ and $A_k, B_k, \ldots, I_k$ $(1 \le k \le 3)$ are domain element variables.

Domain element variables can be shared by the 3 squares according to the definitions of conjugate Latin squares. For example, let the domain variables V, W, U be defined as :

$$\begin{aligned} a \circ_{123} b &= V : Dom_V \\ b \circ_{231} c &= W : Dom_W \\ c \circ_{312} a &= U : Dom_U \end{aligned}$$

where $Dom_{\mathcal{V}}$ $(\mathcal{V} \in \{V,W,U\})$ is a vector of domain element variables corresponding to domain variable $\mathcal{V}$. The variable of the k-th argument of a

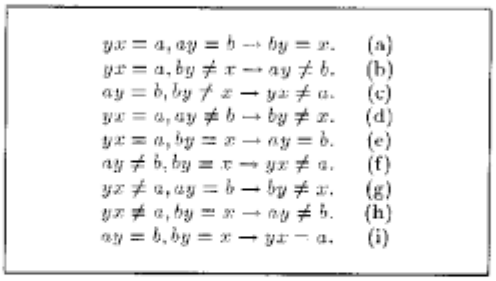


Figure 3: Constraint Propagation Rules for QG5

vector Dom_V ranges over $\{yes, no\}$. The assignment of this variable determines whether or not $V = k$. Here let us consider the following 3 variables X, Y and Z defined as :

$$arg(c, Dom_V, X)$$
$$arg(a, Dom_W, Y)$$
$$arg(b, Dom_U, Z)$$

where $arg(N, F, A)$ unifies the Nth argument of a vector F with A.

The unification $X = Y = Z$ enables the constraint propagation shown in (1) and (2). That is, when X, Y, Z are assigned to no, the relation shown in (1) holds, and when they are assigned to yes, the relation shown in (2) holds.

In this way, negative constraints between different conjugate squares can be propagated by the unification between linked domain element variables.

Fig.3 shows the constraint propagation rules [1] needed to fully propagate QG5 constraints. Rule (a) says that for any value x, y, a, b, if $yx = a, ay = b$ then by must be x. Rules (d) and (g) can be derived from (a) because of the distinctness property for each row and column in the Latin square. Rules (b)(c), (e)(f), and (h)(i) are logically equivalent to (contrapositives of) (a), (d), (g) respectively.

To implement these propagations, negative information must be completely propagated. The 9 constraint propagation rules in Fig.3 are implemented by the CP description shown in the next section. By contrast, in ordinary CLP, since propagation of negative information is not complete, some negative information may be overlooked.

[1] These rules can be considered as *forward-checking* rules w.r.t. the domain element variables appearing in the antecedent parts of the rules. In this case, constraints are propagated only when all domain element variables in the antecedent are bound. On the other hand, *lookahead* enables constraint propagation even if a domain element variable is unbound by calculating constraints with a possible value which the variable may take. We discuss *lookahead* propagation in Section 5.

3.3 Problem Description in CP

Fig.4 shows the problem description for QG5 in CP. `init_set` generates the initial data structure, `link` defines the relationship between domain element variables of each square, `idempotent` defines the idempotent condition, `iso_check` is for removing isomorphic proofs.

`qg5_constraint` defines the QG5 constraints for every `X` and `Y`. `Sq`, `ISq`, `RSq` are squares corresponding to (123)-, (231)-, and (312)-conjugates respectively. `cons1`, `cons2` and `cons3` defines the QG5 constraints for given `X` and `Y` :

$$\forall xy.\ ((yx)y)y = x.$$
$$\forall xy.\ y((xy)y) = x.$$
$$\forall xy.\ (y(xy))y = x.$$

The last 2 equations are equivalent to the first one.

`mult(X,Y,Sq,Z,Zdomain)` calculates `X` $\circ$ `Y` on the Square `Sq` and returns the value `Z` and the vector of its domain element variables `Zdomain`. `mult` is coded using the *freeze* predicate in SICStus Prolog.

For every domain element variable, `var_mainte` creates a frozen process (`var_constraint`) in SICStus Prolog. Each frozen process is unblocked only when the domain element variable is assigned to *yes* or *no*.

`make_Sq` is the main routine for solving QG problems, which chooses a variable having the smallest number of candidates, and assigns a value to that variable. This process is controlled by fail backtrack in Prolog.

3.4 Experimental Results on CP

Table 1 compares the experimental results for QG problems on CP and other systems. The numbers of failed branches generated by CP are almost equal to DDPP and less than those from FINDER and MGTP. In fact, we confirmed that CP has the same pruning ability as DDPP by comparing the proof trees generated by CP and DDPP for QG5. The slight differences in the number of failed branches were caused by the different selection functions (according to which a disjunction for case splitting is selected from the set of disjunctions) used.

For general performance, CP was superior to the other systems in almost every case. In particular we found that no model exists for QG5.16 by running CP on a Sparc-10 for 21 days in October 1993. It was the first new result we obtained.

As for the effect of propagating negative constraints, CP without unification between domain element variables in different conjugate squares generates 3456 branches for QG5.12 (198 sec), while CP with unification generates only 372 branches (19sec) as shown in Table 1. For problems of higher orders, this gap becomes much wider. This shows the considerable effect of negative constraints propagation in reducing the number of branches.

Table 1: Comparison of experimental results using CP and other systems

Problem		Models	Failed Branches			
			DDPP*	FINDER*	MGTP*	CP
QG1	7	8	353	628		354
	8	16	97521	129258	180446	78079
QG2	7	14	364	808	1128	642
	8	2	83987	119141		167397
QG3	8	18	1037	801	399	251
	9	0	46748	35473	312321	11030
QG4	8	0	970	989	3516	338
	9	178	58711	68550	315100	20295
QG5	9	0	15	40	239	16
	10	0	50	356	7026	46
	11	5	136	1845	51904	117
	12	0	443	13527	2749676	372
	13	0				6466
	14	0				34835
	15	0				130425
	16	0				19382469
QG6	9	4	13	97	164	13
	10	0	65	640	2881	53
	11	0	451	4535	50888	474
	12	0	5938	73342	2429467	5573
QG7	9	4	9	62	37027	10
	10	0	40	289	1451992	37
	11	0	321	1526		236
	12	0	2083	10862		1973
	13	64	61612	141513		34206

Problem		Models	Run Time(sec)			
			DDPP*	FINDER*	MGTP*	CP
QG1	7	8	87	3		12.26
	8	16	10260	853	1894	3323
QG2	7	14	80	4	48	23
	8	2	8109	816		8177
QG3	8	18	76	4	28	6
	9	0	5221	244	1022	283
QG4	8	0	71	5	23	7
	9	178	6115	478	1127	456
QG5	9	0	25	2	12	1
	10	0	66	8	66	2
	11	5	228	26	224	6
	12	0	883	158	13715	19
	13	0				467
	14	0				2308
	15	0				11218
	16	0				1831452
QG6	9	4	19	1	14	1
	10	0	51	4	43	2
	11	0	299	25	248	13
	12	0	5180	501	8300	134
QG7	9	4	17	2	90	1
	10	0	47	4	2803	2
	11	0	304	6		11
	12	0	2291	146		113
	13	64	99066	1984		2394

DDPP: Sparc2, FINDER: Sparc2, MGTP: PIM/m-256, CP: Sparc10 (*: [6])

```
run(Size) :-
        init_set(Size,Sq,Label), link(Size,Sq,ISq,RSq,Label,ILabel,RLabel),
        idempotent(Size,Label), iso_check(Size,Sq),
        qg5_constraint(Size,1,1,Sq,ISq,RSq).
        var_mainte(Size,Label,Sq,ISq,RSq),
        make_Sq(Size,Label,ILabel,RLabel,Sq,ISq,RSq),
        fail.
run(_) :- !.

qg5_constraint(Size,N,M,Sq,ISq,RSq) :- ....
        ..... % cons1(Y,X), cons2(Y,X), cons3(Y,X) for all X,Y

cons1(Y,X,Sq,ISq,RSq) :-
        mult(Y,X,Sq,A,Adomain),mult(A,Y,Sq,B,Bdomain),
        mult(Y,X,ISq,B,Bdomain),mult(Y,B,ISq,A,Adomain).
cons2(X,Y,Sq,ISq,RSq) :-
        mult(X,Y,Sq,A,Adomain),mult(A,Y,Sq,B,Bdomain),
        mult(X,Y,RSq,B,Bdomain),mult(Y,B,ISq,A,Adomain).
cons3(X,Y,Sq,ISq,RSq) :-
        mult(X,Y,Sq,A,Adomain),mult(Y,A,Sq,B,Bdomain),
        mult(Y,X,ISq,B,Bdomain),mult(B,Y,RSq,A,Adomain).

var_mainte(Size,[],Sq,ISq,RSq) :- !.
var_mainte(Size,[v(N,M,Var,Cons)|Rest],Sq,ISq,RSq) :-
        var_mainte1(Size,1,v(N,M,Var,Cons),Sq,ISq,RSq),
        var_mainte(Size,Rest,Sq,ISq,RSq).
var_mainte1(Size,A,v(N,M,Var,Cons),Sq,ISq,RSq) :- A > Size,!.
var_mainte1(Size,A,v(N,M,Var,Cons),Sq,ISq,RSq) :- !, arg(A,Cons,Element),
        freeze(Element,var_constraint(Size,Element,A,v(N,M,Var,Cons),Sq,ISq,RSq)),
        A1 is A+1, var_mainte1(Size,A1,v(N,M,Var,Cons),Sq,ISq,RSq).

make_Sq(Size,Label,ILabel,RLabel,Sq,ISq,RSq) :-
        choice(Size,Label,ILabel,RLabel,v(N,M,Var,Cons)),
        guess(v(N,M,Var,Cons)),
        make_Sq(Size,Label,ILabel,RLabel,Sq,ISq,RSq).
```

Figure 4: The problem description for QG5 in CP

4 CMGTP

4.1 Key Features of CMGTP

MGTP is a full first-order theorem prover [2] based on the model generation method[5]. One merit of solving QG problems by MGTP is that they can be described in very short first-order forms. This enables concise problem descriptions. In the case of QG5, MGTP only requires seven input clauses. However, MGTP has the demerit that it cannot propagate negative constraints since it is based on forward reasoning and only uses positive atoms.

To overcome this, we developed CMGTP (Constraint MGTP) in SICStus Prolog with a slight modification to the original MGTP. CMGTP introduces the following key features:

- negative atoms can be used to represent negative constraints propaga-

[2] Although MGTP imposes a condition called *range-restrictedness* to a clause set, any first order predicate can be made range-restricted by introducing the *dom* predicate.

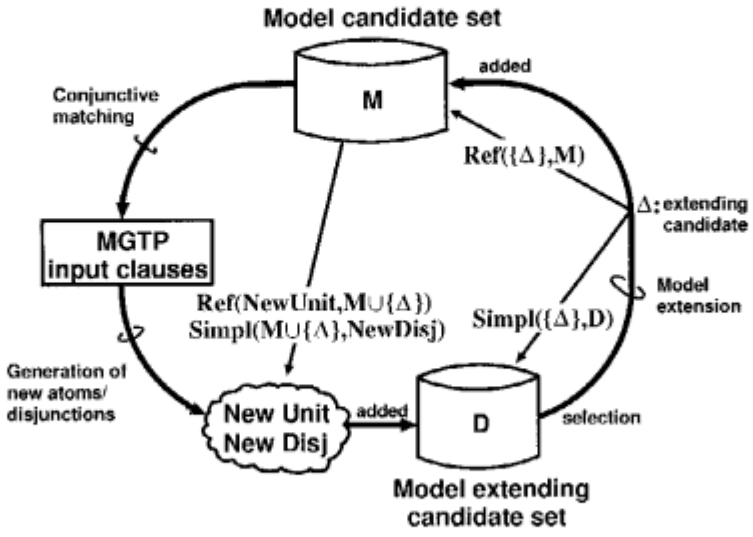


Figure 5: CMGTP model generation processes

tion,

- the integrity constraint (unit refutation), $A, \neg A \rightarrow false$ is introduced,
- unit simplification is performed between atoms in the model candidate set and disjunctions in the model-extending candidate set.

4.2 The Algorithm of CMGTP

Fig.5 shows the CMGTP model generation processes, whose structure is basically the same as MGTP. We first note the algorithm of MGTP briefly.

In Fig.5, MGTP input clauses are represented in an implicational form:

$$A_1, \ldots, A_n \rightarrow C_1; \ldots; C_m$$

where A_i $(1 \leq i \leq n)$ and C_j $(1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, \ldots, A_n$; and the consequent is a disjunction of $C_1, \ldots, C_m$.

Model Extension Rule: If there is a clause $\mathcal{A} \rightarrow \mathcal{C}$ and a substitution σ such that $\mathcal{A}\sigma$ is satisfied in the *model candidate set* M, then $\mathcal{C}\sigma$ (which corresponds to NewUnit if $\mathcal{C}\sigma$ is an atom, or to NewDisj if $\mathcal{C}\sigma$ is a disjunction in Fig.5) is added to the *model extending candidate set* D. We call the matching process between the antecedent and elements in M, *conjunctive matching*.

M retains atoms that have been used in model extension. D is a set of model extending candidates which are generated as a result of application of the model extension rule.

Δ is called an *extending candidate*, which is selected from D according to some criterion (basically we use the unit preference strategy). If an atom is selected as Δ, that atom is added to M unless it is subsumed in M. Conjunctive matching is performed using Δ and M for the next model extension. If a disjunction $C_1; \ldots; C_m$ is selected as Δ, MGTP performs case-splitting unless some C_i is subsumed in M, creating m new branches : $\langle M, D \cup \{C_1\}\rangle, \ldots, \langle M, D \cup \{C_m\}\rangle$.

If *false* is derived for a branch $\langle M, D\rangle$, that branch is terminated with *unsat*. If D becomes empty, that branch is terminated with *sat* and M is returned as a model. The detailed algorithm of the original MGTP is described in [1].

The differences between CMGTP and MGTP lie in the unit refutation processes and the unit simplification processes. If there exist A and $\neg A$ in M then *false* is derived by the unit refutation mechanism as follows:

$$\frac{\begin{array}{cc}(\mathrm{M}) & (\mathrm{M}) \\ \vdots & \vdots \\ \neg A & A\end{array}}{false}$$

If for an atom $\neg A \subset M (A \in M)$, there exists a disjunction which includes $A(\neg A)$, then $A(\neg A)$ is removed from that disjunction by the unit simplification mechanism as follows:

$$\frac{\begin{array}{cc}(\mathrm{M}) & (\mathrm{D}) \\ \vdots & \vdots \\ \neg A & \mathcal{D}_1 \vee A \vee \mathcal{D}_2\end{array}}{\mathcal{D}_1 \vee \mathcal{D}_2} \qquad \frac{\begin{array}{cc}(\mathrm{M}) & (\mathrm{D}) \\ \vdots & \vdots \\ A & \mathcal{D}_1 \vee \neg A \vee \mathcal{D}_2\end{array}}{\mathcal{D}_1 \vee \mathcal{D}_2}$$

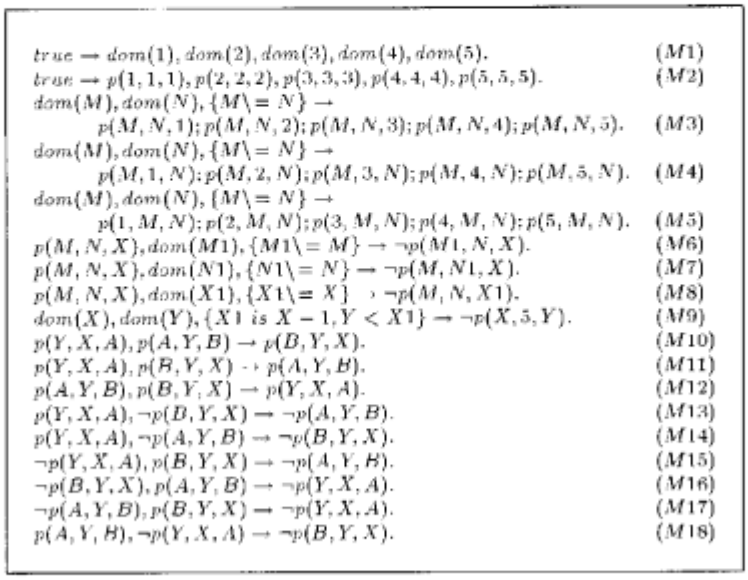
If a disjunction becomes empty as a result of unit simplification, then *false* is derived.

In Fig.5, $Ref(U_1, U_2)$ and $Simpl(U, D)$ are the functions described above, where U_1, U_2, U, are sets of atoms and D is a set of disjunctions. $Ref(U_1, U_2)$ returns *false* if there exist $A \in U_1, B \in U_2$, s.t., A and B are complementary. In this case, the branch is terminated with *unsat*. $Simpl(U, D)$ returns the simplified set of disjunctions by a set of atoms U. If *false* is derived as a result of simplification, then $Simpl$ returns *false*, and the branch is terminated with *unsat*.

There are 2 refutation processes and 2 simplification processes added to the original MGTP:

- $Ref(\{\Delta\}, M)$
- $Ref(NewUnit, M \cup \{\Delta\})$
- $Simpl(\{\Delta\}, D)$
- $Simpl(M \cup \{\Delta\}, NewDisj)$

where Δ is an atom. As a result, these functions guarantee that for any atom $A \in M$, A and $\neg A$ are not both in the current M, and disjunctions in the current D have already been simplified by all atoms in M.



$$true \rightarrow dom(1), dom(2), dom(3), dom(4), dom(5). \quad (M1)$$
$$true \rightarrow p(1,1,1), p(2,2,2), p(3,3,3), p(4,4,4), p(5,5,5). \quad (M2)$$
$$dom(M), dom(N), \{M \backslash = N\} \rightarrow p(M,N,1); p(M,N,2); p(M,N,3); p(M,N,4); p(M,N,5). \quad (M3)$$
$$dom(M), dom(N), \{M \backslash = N\} \rightarrow p(M,1,N); p(M,2,N); p(M,3,N); p(M,4,N); p(M,5,N). \quad (M4)$$
$$dom(M), dom(N), \{M \backslash = N\} \rightarrow p(1,M,N); p(2,M,N); p(3,M,N); p(4,M,N); p(5,M,N). \quad (M5)$$
$$p(M,N,X), dom(M1), \{M1 \backslash = M\} \rightarrow \neg p(M1,N,X). \quad (M6)$$
$$p(M,N,X), dom(N1), \{N1 \backslash = N\} \rightarrow \neg p(M,N1,X). \quad (M7)$$
$$p(M,N,X), dom(X1), \{X1 \backslash = X\} \rightarrow \neg p(M,N,X1). \quad (M8)$$
$$dom(X), dom(Y), \{X1 \ is \ X-1, Y < X1\} \rightarrow \neg p(X,5,Y). \quad (M9)$$
$$p(Y,X,A), p(A,Y,B) \rightarrow p(B,Y,X). \quad (M10)$$
$$p(Y,X,A), p(B,Y,X) \rightarrow p(A,Y,B). \quad (M11)$$
$$p(A,Y,B), p(B,Y,X) \rightarrow p(Y,X,A). \quad (M12)$$
$$p(Y,X,A), \neg p(B,Y,X) \rightarrow \neg p(A,Y,B). \quad (M13)$$
$$p(Y,X,A), \neg p(A,Y,B) \rightarrow \neg p(B,Y,X). \quad (M14)$$
$$\neg p(Y,X,A), p(B,Y,X) \rightarrow \neg p(A,Y,B). \quad (M15)$$
$$\neg p(B,Y,X), p(A,Y,B) \rightarrow \neg p(Y,X,A). \quad (M16)$$
$$\neg p(A,Y,B), p(B,Y,X) \rightarrow \neg p(Y,X,A). \quad (M17)$$
$$p(A,Y,B), \neg p(Y,X,A) \rightarrow \neg p(B,Y,X). \quad (M18)$$

Figure 6: CMGTP rules for QG5.5

4.3 CMGTP Rules for QG5.5

Fig.6 shows the CMGTP input clauses for QG5.5. In this figure, (M1) defines the domain of variables, (M2) corresponds to the idempotence condition, (M3)-(M5) are for generating disjunctions, (M6)-(M8) represent the distinctness property, (M9) is a heuristic rule in order to remove isomorphic proofs, and (M10)-(M18) are the constraint propagation rules for QG5. As shown here, constraint propagation rules in Fig.3 can be written directly as CMGTP input clauses.

In this sense, CMGTP can be considered a meta-language for representing constraint propagation. For example, the original MGTP rule for QG5,

$$p(Y,X,A), p(A,Y,B), p(B,Y,C), X \neq C \rightarrow false$$

can be rewritten in CMGTP rules as :

$$p(Y,X,A), p(A,Y,B) \rightarrow p(B,Y,X).$$
$$p(Y,X,A), \neg p(B,Y,X) \rightarrow \neg p(A,Y,B).$$
$$\neg p(B,Y,X), p(A,Y,B) \rightarrow \neg p(Y,X,A).$$

where negative information is propagated by using the last 2 rules.

In the representation of Fig.6, we can write the integrity constraint $P, \neg P \rightarrow false$ as an input clause instead of using the built-in Ref function. In this case, we do not need the function Ref.

QG5	Models	CMGTP (ss10)		CP (ss10)	
		Failed Branches	Time (sec)	Failed Branches	Time (sec)
7	3	2	2.60	2	0.23
8	1	9	3.03	9	0.30
9	0	15	5.87	15	0.45
10	0	38	14.06	38	1.71
11	5	117	60.45	117	6.75
12	0	372	197.63	372	19.79
13	0	13914	9975.68	13914	1033.63
14	0	64541	42167.29	64541	5081.03

Table 2: Comparison of CMGTP with CP

Modules	CMGTP	CP
Control	32061	1357
Term memory	354916	1967
Finite domain check	36050	32811
Unit simplification	86809	0
Unit refutation	30542	0
Constraint propagation	58899	5157
Candidate selection	20883	23001
Total	620160	64293

Table 3: Execution time profile

4.4 Experimental Results on CMGTP

Table 2 compares the experimental results for QG5 problems on CMGTP and CP with the same selection function, which is different from the selection function used in Table 1 [3].

The number of failed branches is the same. We confirmed that CMGTP and CP generate identical proof trees when they use the same selection function. When it comes to CPU time, however, CMGTP is about 10 times slower than CP for problems of order from 7 to 14.

Table 3 compares the execution time profiles of CP and CMGTP for QG5.10. For finite-domain checking and candidate selection, both systems take almost the same time. The main speed difference occurs during term memory manipulation.

Another major speed difference occurs when performing unit simplification and unit refutation. CMGTP simplifies and modifies disjunctions using a disjunction DB that incorporates discrimination trees. CP does not manipulate disjunctions explicitly. Instead, it maintains domain element variables of the 3 squares to represent candidate values of a domain variable. Updating of candidate values is done by unification implicitly. CMGTP handles unit refutation by conjunctive matching, while CP uses unification failure.

[3] So the results on CP shown in Table 2 are slightly different from the results in Table 1.

In CP, constraint propagation is controlled by a freeze facility; in CMGTP, it is controlled by conjunctive matching. Although both CP and CMGTP perform the constraint propagation rules shown in Fig.3, the former is faster than the latter. This is possibly because in CP these rules are compiled into Prolog clauses.

The current implementation of CMGTP is only a primary version. We expect that its performance can be further improved by a factor of 3 or 4.

5 Discussion

As we described, CP implements a finite-domain constraint propagation facility, which can be considered as a *generate-and-test* scheme with a forward-checking mechanism. On the other hand, MGTP's proving processes are essentially based on the *generate-and-test* scheme. Thus it suffices to introduce negative atoms and a unit simplification mechanism in order to add the forward-checking facility to MGTP.

The forward-checking rules for QG5 shown in Fig.3 can be implemented in both CP and CMGTP. However, it is possible to further reduce the number of redundant branches. For example, consider the following constraint:

- Variables a, b, c are distinct from each other.
- The domain of each variable : $a = \{1,2\},\ b = \{1,2\},\ c = \{1,2,3,4,5\}$

In this case, since values 1 and 2 must be assigned to variables a and b, c must not take 1 or 2. So the domain of c should be rewritten to $\{3,4,5\}$. But neither CP nor CMGTP supports this kind of pruning facility.

If we want our constraint propagation program to have more pruning facilities such as in the above example, we need a *lookahead procedure*, which however is inefficient in general.

We have introduced a naive lookahead mechanism into CMGTP, and have experimented with several typical examples. Some kinds of problems such as cryptarithmetic are solved very efficiently by using lookahead. However, many problems including QG problems are solved less efficiently than by the original one, although the number of branches becomes small. For example, in the case of QG5.9 the original CMGTP generates 15 branches. On the other hand, CMGTP with lookahead procedure generates only 1 branch. But the execution time is 60 times longer than for the original one.

The same may be said of CLP systems. The lookahead mechanism used in CLP systems cannot be helpful for QG problems.

6 Summary and Future Work

CP is a CLP based finite-domain constraint propagation program written in SICStus Prolog. With CP, we can directly handle domain element variables as well as domain variables. This mechanism allows us to unify domain

element variables in different conjugate squares in solving QG problems. As a result, it is possible to have efficient constraint propagation through inverse operations, giving CP better performance than other systems for QG problems.

CMGTP is a modified MGTP to which we can add finite-domain constraint propagation rules such as forward checking rules, by incorporating unit refutation and unit simplification mechanisms as well as negative atoms. We have confirmed that both CP and CMGTP have the same constraint propagation ability as DDPP for pruning the search spaces in QG problems.

While DDPP needs a huge number of input clauses to solve QG problems, in CMGTP we can write intuitive constraint propagation rules directly and compactly using full first order formulas. In this sense, CMGTP seems to be a natural framework for representing constraint propagation.

We now need to improve CMGTP performance by parallelization and refining implementation techniques. We have already developed a parallel version of CMGTP written in KL1 (a parallel logic programming language). Experiments on a parallel inference machine PIM/m and on a parallel UNIX machine are in progress.

References

[1] H. Fujita and R. Hasegawa. A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm. *Proc. of the Eighth International Conference on Logic Programming*, The MIT Press, 1991.

[2] F. Bennett. Quasigroup Identities and Mendelsohn Designs. *Canadian Journal of Mathematics 41*, 341-368, 1989.

[3] J. Slaney. FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guides. *TR-ARP-1/92*, Australian National University, 1992.

[4] P. V. Hentenryck. Constraint Satisfaction in Logic Programming. The MIT Press, 1989.

[5] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. *Proc. of CADE 88*, Argonne, Illinois, 1988.

[6] J. Slaney, M. Fujita, and M. Stickel. Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems. *Computers and Mathematics with Applications*, 1993.

[7] M. Fujita, J. Slaney, and F. Bennett. Automatic Generation of Some Results in Finite Algebra. *Proc. of International Joint Conference on Artificial Intelligence*, 1993.

[8] R. Hasegawa, Y.Shirai. Constraint Propagation of CP and CMGTP: Experiments on Quasigroup Problems. *Workshop 1C (Automated Reasoning in Algebra), CADE-12*, 1994.