TM-1183

Proceedings of FGCS '92 WORKSHOP on Constraint Logic Programming

by

F. Mizoguchi & A. Aiba

July, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Workshop W2 ICOT TM-1183

FGCS'92

INTERNATIONAL CONFERENCE ON
FIFTH GENERATION COMPUTER SYSTEMS 1992

WORKSHOP ON
CONSTRAINT LOGIC PROGRAMMING

June 6-7, 1992 Tokyo, Japan

PROCEEDINGS

Institute for New Generation Computer Technology

FGCS'92 WORKSHOP
– W2: Constraint Logic Programming –

June 6 Saturday, 1992 10:00 – 17:15
June 7 Sunday, 1992 10:00 – 17:35
at
Institute for New Generation Computer Technology
Mita-kokusai Bldg. 21Fl.
4-28, Mita 1-chome, Minato-ku, Tokyo 108

This workshop is aiming at bringing together researchers working on the constraint programming, and exchanging information on anything about constraints, such as theory, language design, implementation issues, applications, and others. In these years, constraints have been becoming very important issues on problem solving not only as a promising way to extend logic programming, but also as a new powerful paradigm. This workshop will emphasize applications of this paradigm as well as theory, and implementations.

WORKSHOP ORGANIZERS:

Fumio Mizoguchi (Tokyo Science University),
Ken McAloon (CUNY Graduate Center)
Martin Nilsson (Swedish Institute of Conputer Science),
Pascal van Hentenryck (Brown University),
Akira Aiba (Institute for New Generation Computer Technology).

PROGRAM

Saturday, June 6

10:00 – 10:15 Welcome speech

10:15 – 11:00 On Solution Compactness and Models of Clark's Axioms

Michael J. Maher (IBM – T. J. Watson Research Center)

11:00 – 11:45 An Event Structure Semantics for Concurrent Constraint Programming

Ugo Montanari, and Francesca Rossi (University of Pisa)

11:45 – 13:15 Lunch

13:15 – 14:00 The CORE approach to Constraint Logic Programming

Thom Frühwirth, Alexander Herold, Volker Kuechenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace (ECRC)

14:00 – 14:45 Parallel Constraint Logic Programming Language GDCC

Satoshi Terasaki (ICOT)

14:45 – 15:15 Coffee Break

15:15 – 16:00 A Practical Approach to the Global Analysis of Constraint Logic Programs

M. J. Garcia de la Banda, and M. Hermenegildo (Technical University of Madrid)

16:00 – 16:45 Impose Constraints in a Multi-User, Dynamically Changing Environment

Qing Ge (KBS Technology Inc.)

18:00 – Party

Sunday, June 7

10:00 – 10:45 Constraint Logic Programming with Priority

Takashi Hattori (RIMS, Kyoto University)

10:45 – 11:30 Anticipatory Pruning Networks in CLP Language

Geun-Sik Jo (In-Ha University) and Ken McAloon (CUNY Graduate Center)

11:30 – 12:15 Approximation in the Framework of Generalised Propagation

Thierry Le Provost (ECRC)

12:15 – 13:30 Lunch

13:30 – 14:15 Constraint Satisfaction and Optimization Using Sufficient Conditions for Constraint Violation

Fumihiro Maruyama, Yoriko Minoda, Shuho Sawada, and Yuka Takizawa (Fujitsu Laboratories Ltd.)

14:15 – 15:00 Representing Situations in Forward Planning with Boolean Arrays

Neng-Fa Zhou (Kyushu Institute of Technology)

15:00 – 15:30 Coffee Break

15:30 – 15:50 Short Presentation

15:50 – 16:10 Short Presentation

16:10 – 16:30 Short Presentation

16:30 – 16:50 Short Presentation

On Solution Compactness and Models of Clark's Axioms

Michael J. Maher
IBM - T.J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.
mjm@watson.ibm.com
Fax: +1-914-784-7455

March 26, 1992

1 Introduction

Solution compactness was introduced in [6, 7] as a condition on the parameter of the Constraint Logic Programming Scheme. Although several constraint domains now used in CLP languages have been shown to be solution compact, there has not yet been a serious study of the restrictions that solution compactness forces. This abstract presents preliminary results of a case study, where we consider only models of a familiar theory; the models of Clark's axioms for unification [3]. We also discuss some related model-theoretic properties of this theory.

2 Preliminary Definitions

Throughout this paper Σ will denote a set of function symbols, and, unless stated otherwise, Σ will be the signature of the algebras we discuss. $\tilde{x}$ denotes a sequence of distinct variables $x_1, x_2, \ldots, x_n$. For lack of space, we assume that many concepts are understood. In particular, the definitions of finite, rational (aka regular) and infinite trees and their associated algebras are assumed. The algebras are denoted respectively by $\mathcal{FT}, \mathcal{RT}$ and $\mathcal{IT}$. Similarly, we will not state Clark's axioms (denoted by E_{FT}) and the Domain Closure Axiom (DCA). See [9] or [2] for definitions not in this abstract. An element of a structure which is not in the range of any $f \in \Sigma$ is called an *isolated* element.

A *strictly rational term* is a rational tree over $\Sigma \cup X$ which is not a finite tree, where X is a set of variables. An infinite tree is said to have a *rational skeleton* if it can be obtained by substituting infinite trees for variables in some strictly rational term. Let $\mathcal{NRT}$ denote the subalgebra of $\mathcal{IT}$ formed by the set of infinite trees which do not have a rational skeleton.

Proposition 1 *$\mathcal{NRT}$ is a model of E_{FT}, DCA.*

A *basic formula* has the form $\exists \tilde{y} \ \bigwedge x_i = t_i(\tilde{y})$. A basic formula is *linear* if each variable y_i appears at most once in the formula $\bigwedge x_i = t_i(\tilde{y})$. The classes of linear basic formulas, basic formulas, formulas built from linear basic formulas, and linear basic formulas and their negations, are denoted respectively by $\mathcal{LBF}, \mathcal{BF}, \mathcal{LF}$ and $\mathcal{NLBF}$. For classes of formulas $\mathcal{L}_1$ and $\mathcal{L}_2$, we define $\mathcal{L}_1 \subseteq \mathcal{L}_2$ if, for every $\phi(\tilde{x}) \in \mathcal{L}_1$, there is $\psi(\tilde{x}) \in \mathcal{L}_2$ such that $E_{FT} \models \phi(\tilde{x}) \leftrightarrow \psi(\tilde{x})$.

3 Solution Compactness

A *constraint domain* $(\mathcal{A}, \mathcal{L})$ consists of a structure $\mathcal{A}$ and a subset $\mathcal{L}$ of the first-order formulas expressible using the signature of $\mathcal{A}$ (the language of constraints). We assume $\mathcal{L}$ is closed under variable renaming, conjunction and existential quantification. A constraint domain $(\mathcal{A}, \mathcal{L})$ is *solution compact* [6, 7] if it satisfies the following conditions:

(SC_1) For every element $a \in A$, there is a collection of constraints $\{c_i(x)\}_{i \in I}$ with a single free variable x such that given $d \in A$, $c_i(d)$ is true in $\mathcal{A}$ for every $i \in I$ iff $d = a$. (That is, roughly speaking, for every a there is a collection of constraints which can only be satisfied by a.) In symbols

$$\mathcal{A} \models x = a \leftrightarrow \bigwedge_{i \in I} c_i(x)$$

(SC_2) For every unary constraint $c(x) \in \mathcal{L}$, there is a collection of constraints $\{c_i(x)\}_{i \in I}$ with free variable x such that, given $d \in A$, $c(d)$ is true in $\mathcal{A}$ iff for every $i \in I$, $c_i(d)$ is false in $\mathcal{A}$. (Roughly speaking, for every unary constraint c there is a collection of constraints whose disjunction describes the complement of c.) In symbols

$$\mathcal{A} \models \neg c(x) \leftrightarrow \bigvee_{i \in I} c_i(x)$$

The first condition is comparatively regular, satisfying monotonicity properties on both $\mathcal{A}$ and $\mathcal{L}$. However SC_2 is considerably less regular, which makes it difficult to give clean characterizations of solution compactness. For example, while $(\mathcal{FT}, \mathcal{LBF})$ is solution compact, $(\mathcal{FT}, \mathcal{L})$ appears not to be when $\mathcal{L}$ is the constraint language generated by $\mathcal{LBF}$ and the constraint $\exists y\, x = f(y, y)$. This extreme sensitivity to the constraint language is due to SC_2.

In the case of linear constraints we can characterize those models of E_{FT} which form solution compact constraint domains:

Proposition 2 *If $\mathcal{A} \models E_{FT}, DCA$ and $\mathcal{LBF} \subseteq \mathcal{L} \subseteq \mathcal{LF}$ then $(\mathcal{A}, \mathcal{L})$ is solution compact iff $\mathcal{A}$ is isomorphically embedded in $\mathcal{NRT}$.*

If $\mathcal{A} \models E_{FT}$ and $\mathcal{NLBF} \subseteq \mathcal{L} \subseteq \mathcal{LF}$ then $(\mathcal{A}, \mathcal{L})$ is solution compact iff $\mathcal{A}$ is isomorphically embedded in $\mathcal{NRT}_{\odot}$.

Here $\mathcal{NRT}_{\odot}$ is the algebra of non-rational trees generated by Σ from one isolated point. These results imply that, at least for linear constraints, solution compact constraint domains involving models of E_{FT} can be viewed as domains over trees. At this stage of the work it is not clear whether this property extends to all solution compact constraint domains with non-linear constraints.

4 Model Completeness

We have a construction of the free product of models of E_{FT}. This gives the following result, which is not surprising when we consider that the models of E_{FT} are the locally free algebras.

Proposition 3 *The models of E_{FT} are closed under the taking of free products.*

This construction provides a method to produce an n-generic model for a logic program [10] and (essentially the same thing) produce an algebra $\mathcal{A}$ such that all logic programs are canonical in $\mathcal{A}$ [1, 8]. However several other methods are known.

A theory T is *model complete* [2, 11] if, for all models $\mathcal{A}_1, \mathcal{A}_2$ such that $\mathcal{A}_1$ is a submodel of $\mathcal{A}_2$ and for every formula $\phi(\bar{x})$ and sequence $\bar{a}$ of elements of A_1, $\mathcal{A}_1 \models \phi(\bar{a})$ iff $\mathcal{A}_2 \models \phi(\bar{a})$. Using the above construction, careful examination of quantifier elimination procedures [4, 9] and characterizations of model completeness [2, 11] we can show

Proposition 4 *If Σ is finite, E_{FT} is not model complete.*
If Σ is finite, E_{FT}, DCA is model complete.
If Σ is infinite, E_{FT} is model complete iff Σ contains only constants.

References

[1] H. Blair & A. Brown, Definite Clause Programs *are* Canonical (Over a Suitable Domain), *Annals of Mathematics and Artificial Intelligence*, to appear.

[2] C.C. Chang & H.J. Keisler, *Model Theory*, North-Holland, 1977.

[3] K.L. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (Eds.), Plenum Press, New York, 293–322, 1978.

[4] H. Comon, *Unification et Disunification: Théorie et Applications*, Doctoral thesis, I.N.P. de Grenoble, France, 1988.

[5] H. Comon, Disunification: A Survey, in: *Computational Logic*, J-L. Lassez & G. Plotkin (Eds.), MIT Press, 1991.

[6] J. Jaffar & J-L. Lassez, Constraint Logic Programming, Technical Report, Department of Computer Science, Monash University, 1986.

[7] J. Jaffar & J-L. Lassez, Constraint Logic Programming, *Proc. Conf. on Principles of Programming Languages*, 1987, 111–119.

[8] J. Jaffar & P. Stuckey, Canonical Logic Programs, *Journal of Logic Programming*, 3, 143–155, 1986.

[9] M.J. Maher, Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees, *Proc. 3rd. Symp. Logic in Computer Science*, Edinburgh, 348–357, 1988. Full version: IBM Research Report, T.J. Watson Research Center.

[10] J.C. Shepherdson, Negation in Logic Programming, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 19–88, 1988.

[11] A. Robinson, *Complete Theories*, North-Holland, 1956.

An Event Structure Semantics for Concurrent Constraint Programming*

Ugo Montanari   Francesca Rossi
University of Pisa
Computer Science Department
Corso Italia 40
56100 Pisa, Italy
{ugo,rossi}@dipisa.di.unipi.it

March 25, 1992

We propose a new semantics for concurrent constraint (cc) programs ([1, 5, 6]). The idea comes from a refinement and an extension of a partial order semantics proposed in [1, 2], where states of computations are represented by graphs, computation steps by graph production applications, and where each computation has an associated partial order, which is able to express the causal dependencies among the steps of such computation. That semantics, through rather interesting and original for the uniform treatment of tokens and agents, as well as the analysis of a cc program from the true-concurrency approach, is not entirely satisfactory, for two main reasons. First, the use of graph grammars and their categorical description to model cc computations allows an elegant formalization of all the basic cc features, but it is not familiar at all to the logic programming community. Second, a partial order is associated to each deterministic computation, but no unique structure is associated to a (possibly nondeterministic) cc program. In this paper we try to eliminate such unsatisfactory points of the approach in [1] by giving an adequate solution to both problems.

Graph grammars are here replaced by general rewrite rules, which are significantly simpler and more familiar to the logic programming environment while still maintaining all the properties of graph grammars which are fundamental for the true-concurrency approach. In particular, one of the most important features is the possibility of expressing formally what we call "context objects", i.e., objects which are needed for a computation step to take place, but which are not affected by such step. In fact, such objects allow to model faithfully the concept of asked constraints, which is necessary if we want to be able to express the possibility of having simultaneous ask operations. Therefore, our rewrite rules are context-dependent, i.e., they have a left hand side, a right hand side, and a context. A rule is applicable if both its left hand side and its context are present in the current state of the computation, and its application removes the left hand side (but not the context) and adds the right hand side (for a deeper analysis of context-dependent formalisms, and a proposal to extend Petri nets to treat context objects, see [3]). The evolution of each of the agents in a cc program, as well as the declarations of the program and its underlying constraint system, are all expressible by sets of such rules. In this way each computation step, i.e., the application of one of such rules, represents either

*Research partially supported by the GRAGRA Basic Research Esprit Working Group n.3299 and by Alenia S.p.A.

the evolution of an agent, or the expansion of a declaration, or the entailment of some new token. In such view of a computation, we adopt the eventual interpretation of the tell operation, where a constraint is added to the current store without any consistency check.

In this way, a set of rules is obtained from a cc program. Such a set could already provide an operational semantics of the given cc program. However, for us it is the starting point for our (truly concurrent) semantic treatment. The idea is to construct a set of terms from the rules. Each of such terms is a pair, where one element is the "type" of the term (which can be either a token, an agent, or a rule), while the other element is the coding of the computation path (or proof) which leads to the element of such type. Since elements occurring in different points of a computation are obtained through different computation paths, it is clear that they will give rise to different terms. In this way, no new term can coincide with an already generated term.

By the way terms are constructed, causal dependency of term t_1 on term t_2 amounts simply to the appearance of t_2 as a subterm of t_1. Furthermore, mutual exclusion among terms, which means that the elements represented by such terms cannot appear in the same computation, can also be expressed. In fact, two terms are mutually exclusive if some of the maximal subterms they share represent agents. In fact, each agent can evolve in either one way or another, but these two ways lead to alternative computations. In other words, agents can be seen as consumable resources, which can be consumed by at most one computation step in each computation. On the other hand, tokens are non-consumable resources, which thus can be used by any number of computation steps in the same computation. This is reflected by the fact that terms which either do not share any subterm, or whose maximal shared subterms all represent tokens, are considered to be concurrent.

Due to the presence of all these three relations (causal dependency, mutual exclusion, and concurrency) among the terms of the constructed set, such a set is able, alone, to represent not only one, but all possible deterministic computations in a given cc program. It is, therefore, both sound and complete. Furthermore, for each computation it is able to provide a suitable partial order which describes the causal dependency pattern among the computation steps. Moreover, this partial order coincides with the one obtained by the semantics in [1] for the same computation, thus making our new proposal a conservative extension of the old one.

The semantics is called an "event structure semantics" because the set of terms associated to a cc program, restricted to those terms representing rule applications, coincides with what is called an event structure in [7]. Moreover, the construction which associates a unique set of terms to a given set of rewrite rules resembles that in [7] to obtain an occurrence net from a Petri net, although there are some significant differences.

References

[1] Montanari U., Rossi F., "True Concurrency in Concurrent Constraint Programming", on *Proc. ILPS91*, MIT Press, 1991.

[2] Montanari U., Rossi F., "Graph rewriting for a partial ordering semantics of concurrent constraint programming", to appear on TCS, special issue on graph grammars, Courcelle B. editor.

[3] Montanari U., Rossi F., "Contextual nets", submitted for publication, 1992.

[4] Saraswat V. A., "Concurrent Constraint Programming Languages", Ph.D. Thesis, Carnegie-Mellon University, 1989. Also 1989 ACM Dissertation Award, MIT Press.

[5] Saraswat V. A., Rinard M., "Concurrent Constraint Programming", on *Proc. POPL*, ACM, 1990.

[6] Saraswat V. A., Rinard M., Panangaden P., "Semantic Foundations of Concurrent Constraint Programming", on *Proc. POPL*, ACM, 1991.

[7] Winskel G., "Event Structures", on *Petri nets: applications and relationships to other models of concurrency*, Springer-Verlag, LNCS 255.

The CORE approach to Constraint Logic Programming*

An extended abstract

Thom Frühwirth, Alexander Herold, Volker Kuechenhoff, Thierry Le Provost
Pierre Lim, Eric Monfroy, Mark Wallace

European Computer-Industry Research Centre
Arabellastr. 17, D-8000 Munich 81, Germany
email: alex@ecrc.de

Constraint Logic Programming has been pioneered at ECRC with the development of CHIP [3, 6]. Following on ¿from the former CHIP project the new CORE (COnstraint REasoning) project at ECRC aims at developing a cleaner more orthogonal constraint programming environment where user-defined constraints provide the required extensibility. Constraints can be defined directly, and their behaviour can be tuned by experiment without undermining program correctness.

The work currently pursued covers the whole spectrum of constraint logic programming: investigating spezialized constraint solvers and the interaction of such constraint solvers in tradition of the CLP framework; providing the possibilities for the user to define and implement constraint solvers or to strengthen and spezialize already existing constraint solvers; generalizing the concept of propagation in order to make it applicable to other constraints than finite domains; improving the search capabilities by incorporating techniques such as simulated annealing into the constraints framework. In following we describe the different activities in more detail.

- Constraint solving in the tradition of CLP is supported in CHIP on two new computation domains: boolean constraints with a boolean unification algorithm and linear rational arithmetic with a "symbolic" simplex algorithm. However, some interesting problems in areas such as financial planning, computational geometry and computer-aided design require the solution of non-linear constraints. Gröbner Bases [2, 1] provide an interesting approach to their solution. We are currently investigating different strategies and heuristics to speed up their computation and extending the range of applications through the introduction of polynomial inequalities.

- The question of how various decision procedures in a CLP system interact and cooperate is a very important one for practical reasons. For example in a CLP system with a built-in non-linear solver, linear equations should be handled by a spezialized solver, e.g. by Gaussian elimination. We are investigating how to network several decision procedures

*This work is partially funded by the ESPRIT project CHIC, Nr. 5291

with overlapping domains to form a relatively efficient general decision procedure. The issues addressed are the specification of the possible network topologies, of soundness and correctness.

- In current constraint logic programming systems constraint solving is hard-wired in a "black box". We are studying the use of logic programs to define and implement constraint solvers. The representation of constraint evaluation in the same formalism as the rest of the program greatly facilitates the prototyping, extension, specialization and combination of constraint solvers. In our approach constraint evaluation is specified using multi-headed guarded clauses called *Constraint Simplification Rules* (SiRs). SiRs define a determinate conditional rewrite system that controls how conjunctions of constraints simplify. SiRs operate on top of a so called *host language* which allows constraints to be specified by the definite Horn Clauses provided by this host language. In this way the approach merges the advantages of constraints (eager simplification) and predicates (lazy choices) by definite clauses. A prototype system within the SEPIA [5] environment has been built.
- In CHIP propagation is only provided on finite domains [6]. This concept has been generalized and thus made available for other computation domains, e.g. linear rational arithmetic. More importantly the degree of propagation can now be controlled through the definition of approximations of generalized propagation. The general concept of propagation will be presented at the FGCS conference and more recent results will be reported in another contribution [4] to this workshop.
- Search is supported in CHIP by the usual Prolog unfolding and by some specialized predicates for selecting values from domains such as *"deleteff"* implementing the first fail principle [6]. However, CHIP offers no alternative to naive backtracking after a failure. Methods which efficiently solve hard problems, such as the travelling salesman problem, use ways of intelligently improving on current guesses, rather than blind backtracking. We are investigating ways of incorporating such search techniques into the constraints framework in order to solve such hard search problems while still satisfying customer-specific constraints.

This research is embedded into the European ESPRIT project 5291, "Constraints Handling in Industry and Commerce" (CHIC). The CHIC project aims at accelerating the exploitation of Constraint Logic Programming in industrial environments. The Core project has a critical role in CHIC as the research and integration backbone. Partners in CHIC include all ECRC's shareholders - Bull, ICL, Siemens; further academic partners Imperial College from London, CERT from Toulouse and CMSU from Athens; and finally a "user group" of five industrial associates who seek to use constraints programming to help tackle their specific business problems: AIS Italy, Braghenti Italy, OCT Spain, Dassault France, Renault France.

The industrial applications cover the following areas:

- Production management and Scheduling:
  Three different applications are developed in this area. The first one is an integrated production planning and scheduling system in the weaving industry (Braghenti, Italy) where the specific characteristics are the large quantity and diversity of orders. Long set-up times of the weaving machines make a careful planning of certain bottle-necks of the production process necessary.

The second one tackles a scheduling problem in a workshop of Dassault manufacturing elements for the production of aircrafts. Contrary to the first application this workshop produces small quantities of a highly diversified range of primary parts. The main result expected ¿from the system is reduction and stabilization of the cycle duration (time needed to produce one part).

Finally a decision support tool for the short term production planning for the car manufacturing of Renault is being developed.

- VLSI verification environment:
This work aims at developing the algorithms and interfaces for a VLSI verification environment of Siemens. Extending the range of application ¿from combinatorial to sequential circuits requires a completely new class of algorithms based on existing constraint solvers. The development of suitable interfaces to standard CAD systems will be essential for a wide-spread use of such systems.

- Logistics and network management:
In this area two problems are investigated. A treasury management system is being developed by AIS, Italy to improve short term liquidity management of an italian bank. CLP seems to be particularly suited to cope with additional constraints imposed by the central bank of Italy.

  The second application in this area deals with vehicle-fleet scheduling investigated by CMSU. The basic problem can be described as follows: For a given fleet of vehicles construct a schedule of vehicle routes from one depot to a number of delivery points in such a way that the requirements of all customers must be met, the capacity of the vehicles may not be violated and the total cost of delivery must be minimized. The fleet capacity is fixed and the demand for some commodity is known. Various extensions of this basic problem occur in many real-life situations.

- Traffic Control:
Controlling the assignment of times for the red and green phases of a traffic light depending on the traffic load is a challenging problem which will become more and more important. CLP offers the possibility to develop a simulator incorporating a large variety of constraints.

All these applications show the potential of the CLP technology for industry and are at the same time a very valuable test bed for the new developments that have been listed above.

References

[1] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. The Constraint Logic Programming Language CAL. In *Proceedings on the International Conference on Fifth Generation Computer Systems (FGCS-88)*, ICOT, Tokyo, Japan, December 1988.

[2] B. Buchberger. Applications of Gröbner Beses in Non-Linear Computational Geometry. In D. Kapur and J.L. Mundy, editors, *Geometric Reasoning*, pages 413–446. MIT Press, 1989.

[3] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.

[4] T. Le Provost. Approximation in the Framework of Generalized Propagation. In *Workshop on Constraint Logic Programming at FCGS-92*, Tokyo, Japan, June 1992.

[5] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - An Extendible Prolog System. In *Proceedings of the 11th World Computer Congress IFIP'89*, San Francisco, August 1989.

[6] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, Ma, 1989.

Parallel Constraint Logic Programming Language GDCC (Extended Abstract)

Satoshi Terasaki

Institute for New Generation Computer Technology
4-28. Mita 1-chome. Minato-ku. Tokyo 108. Japan
terasaki@icot.or.jp

1 Introduction

The constraint logic programming (CLP) paradigm was proposed by Colmeraure[7]. and Jaffar and Lassez[13] as an extension of logic programming. A similar paradigm (or language) was proposed by the ECRC group [8]. A sequential CLP language CAL (*Contrainte avec Logique*) was also developed at ICOT[1].

The CLP paradigm is a powerful programming methodology that allows users to specify what (declarative knowledge) without specifying how (procedural knowledge). This abstraction allows programs to be more concise and more expressive. Unfortunately. the generality of constraint programs brings with it a higher computational cost. Parallelization is an effective way of making CLP systems efficient. There are two major levels to CLP system parallelization. One is the execution of an inference engine and constraint solvers in parallel. The other is the execution of a constraint solver in parallel.

Several works have been published on extending this work from the sequential frame to the concurrent frame. Among them are a proposal of ALPS[14] which introduces constraints into committed-choice language. a report on some preliminary experiments in integrating constraints into the PEPSys parallel logic system[12]. and a framework for a concurrent constraint (cc) language to integrate constraint programming with concurrent logic programming languages[17].

The cc programming language paradigm models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a (consistent) global database of constraints called the *store*. Constraints occurring in program text are classified by whether they are querying or asserting information. into the *Ask* and *Tell* constraints. This paradigm is embedded in a guarded (conditional) reduction system. where guards contain the *Ask* and *Tell*. Control is achieved by requiring that the *Ask* constraints in a guard are true (entailed). and that the *Tell* constraints are consistent (satisfiable). with respect to the current state of the *store*. Thus. this paradigm has a high affinity with KL1.

GDCC[10. 22]. Guarded Definite Clauses with Constraints. that satisfies two level parallelism. is a parallel CLP language that introduces the framework of cc into a committed-choice language KL1[23]. and is currently running on the Multi-PSI. a loosely coupled distributed memory parallel logic machine. GDCC has multiple solvers to enable a user to easily specify a proper solver for a domain. and a *block* mechanism that enables meta-operation to a constraint set. We introduce the language and a parallel constraint solver for rational polynomials based on a parallel implementation of the Buchberger algorithm[4]. This algorithm is widely used in computer algebra. and also fits reasonably well into the CLP scheme since it is incremental and (almost) satisfaction-complete as shown in [1. 16]. Recently. there have been several attempts made to parallelize the Buchberger algorithm. with generally disappointing results[15. 20. 21]. except for shared-memory machines[24. 6]. We parallelize the Buchberger algorithm on the distributed memory parallel machine.

2 GDCC

GDCC is an experimental committed-choice cc language that includes most of KL1 as a subset. since KL1 built-in predicates and unification can be regarded as the constraints of distinguished domain HERBRAND[17].

GDCC contains *Store*. a central database to save the canonical forms of constraints. Whenever the system meets an *Ask* or a *Tell* constraint. the system sends it to the proper solver. *Ask* constraints are only allowed passive constraints which can be solved without changing the content of the *Store*. While in the *Tell* part. constraints which may change the *Store* can be written. In the GDCC program. only *Ask* constraints can be written in guards. This is similar to the KL1 guard in which active unification is inhibited.

2.1 GDCC Language

Now we define the logical semantics of GDCC as follows. S is a finite set of *sorts*, including the distinguished sort HERBRAND. F is a set of *function symbols*. C is a set of *constraint symbols*, P is a set of *predicate symbols*, and V is a set of *variables*. A sort is assigned to each variable and function symbol. A finite sequence of sorts, called a *signature*, is assigned to each function, predicate, and constraint symbol. We define the following notations.

- We write $v : s$ if variable v has sort s.
- $f : s_1 s_2 \ldots s_n \rightarrow s$ if functor f has signature $s_1 s_2 \ldots s_n$ and sort s, and
- $p : s_1 s_2 \ldots s_n$ if predicate or constraint symbol p has signature $s_1 s_2 \ldots s_n$.

We require that terms be well-sorted, according to the standard inductive definitions. An *atomic constraint* is a well-sorted term of the form $c(t_1, t_2, \ldots, t_n)$ where c is a constraint symbol, and a *constraint* is a set of atomic constraints. Let Σ be the many-sorted vocabulary $F \cup C \cup P$. A *constraint system* is a tuple (Σ, Δ, V, C), where Δ is a class of Σ structures. We define the following meta-variables: c ranges over constraints and g,h range over atoms. We can now define the four relations *entails*, *accepts*, *rejects*, and *suspends*. Let x_g be the variables in constraints c and c_l.

Definition 2.1.1 c *entails* $c_l \overset{\text{def}}{=} \Delta \models (\forall x_g)(c \Rightarrow c_l)$

Definition 2.1.2 c *accepts* $c_l \overset{\text{def}}{=} \Delta \models (\exists)(c \wedge c_l)$

Definition 2.1.3 c *rejects* $c_l \overset{\text{def}}{=} \Delta \models (\forall x_g)(c \Rightarrow \neg c_l)$

Note that the property *entails* is strictly stronger than *accepts*, and that *accepts* and *rejects* are complementary.

Definition 2.1.4 c *suspends* c_l
$\overset{\text{def}}{=} c$ *accepts* $c_l \wedge \neg$ (c *entails* c_l).

A GDCC program is comprised of clauses that are defined as tuples (head, ask, tell, body), where "head" is a term having unique variables as arguments. "body" is a set of terms. "ask" is said to be *Ask constraint*, and "tell" is said to be *Tell constraint*. The "head" is the head part of the KL1 clause, "ask" corresponds to the guard part[1], and "tell" and "body" are the body part.

A clause (h, a, c, b) is a candidate for goal g in the presence of *store* s if $s \wedge g = h$ *entails* a. A goal g *commits* to candidate clause (h, a, c, b), by adding $t \cup c$ to the *store* s, and replacing g with b. A goal fails if all the candidate clauses are *rejected*. The determination of *entailment* for multiple clauses and *commitment* for multiple goals can be done in parallel.

[1] "ask" contains constraints in the HERBRAND domain, that is, it includes the normal guards in KL1.

2.2 GDCC System

The GDCC system consists of the compiler, the shell, the interface and the constraint solvers. The compiler translates a GDCC source program into KL1 code. The shell translates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace and spy functions, together with solver-level event logging. The shell also provides limited support for incremental querying. The interface interacts with a GDCC program (object code), sends body constraints to a solver and checks guard constraints using results from a solver. The GDCC system is shown in Figure 1. The components are concurrent processes. Specifically, a GDCC program and the constraint solvers may execute in parallel, synchronizing only when, and to the extent necessary, at the program's guard constraints.

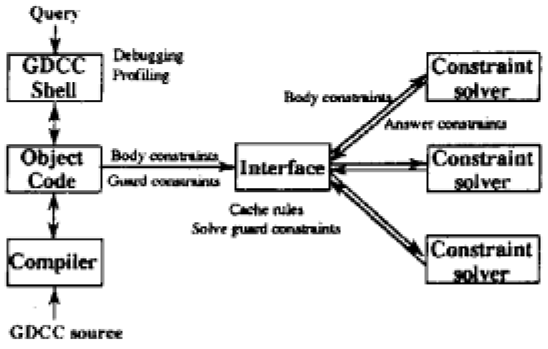

Figure 1: System Construction

2.3 Block

In applying GDCC to application problems, two problems arose. These were the handling of multiple contexts and the synchronization between an inference engine and solvers.

For instance, when the solution $X^2 = 2$ is obtained from the algebraic solver, it must be solved in more detail using a function to compute approximated real roots in univariate equations. In this example, there are two constraint sets, one that includes $X = \sqrt{2}$, and another that includes $X = -\sqrt{2}$. In a sequential CLP system, these can be handled using a backtrack mechanism. In committed-choice language GDCC, however, we cannot use backtracking to handle multiple contexts. A similar situation occurs when a meta operation to constraint sets is required, such as when computing a maximum value with respect to a given objective function. Thus other mechanism is necessary to describe the timing and the target constraints for executing a meta operation, as a clause sequence in a program does not relate to the execution sequence in GDCC.

Introducing local constraint sets, however, independent of the global constraints can eliminate these problems. Multiple contexts are realized by considering each local constraint as one context. An inference engine and solvers can be synchronized at the end point of the evaluation of a local constraint set.

Therefore, we introduced a mechanism, called *block*, to describe the scope of a constraint set. We can solve a certain goal sequence with respect to a local constraint set. The block is represented in a program by a built-in predicate **call**, as follows.

call(*Goals*) **using** *Solver-Package* **for** *Domain*
initial *Input-Con* **giving** *Output-Con*

Constraints in goal sequence *Goals* are computed in a local constraint set. "**using** *Solver-Package* **for** *Domain*" denotes the use of *Solver-Package* for *Domain* in this block. "**initial** *Input-Con*" specifies the initial constraint set. "**giving** *Output-Con*" indicates that the result of computing in the block is *Output-Con*. To encapsulate failure in a block, the *shoen* mechanism of PIMOS[19] is used.

3 Parallel Algebraic Solver

The constraint domain of the algebraic solver is multivariate (non-linear) algebraic equations. The Buchberger algorithm [4] is a method to solve non-linear algebraic equations which have been widely used in computer algebra over the past few years. Since this algorithm satisfies almost criteria as shown in [1, 16], this algorithm is utilized as the constraint solver in GDCC.

In this section, we outline both the sequential and the parallel versions of the Buchberger algorithm.

3.1 Gröbner Basis and Buchberger Algorithm

With no loss of generality, we can assume that all polynomial equations are in the form $f = 0$. Let $E = 0$ be a system of polynomial equations $\{f_1 = 0, \ldots, f_n = 0\}$. The following close relation between the solutions of $E = 0$ and the elements of $\mathcal{I}(E)$ of the ideal generated by E is well known.

Theorem 3.1.1 (Hilbert zero point theorem)
Let f be a polynomial. Every solution of $E = 0$ is also a solution of $f = 0$, iff there exists a natural number s such that $f^s \in \mathcal{I}(E)$.

Corollary 3.1.1 *E has no solution iff $1 \in \mathcal{I}(E)$.*

Thus, the problem of solving given polynomial equations is reduced to that of deciding whether a polynomial belongs to the ideal. Buchberger introduced the notion of Gröbner bases, and gave an algorithm to determine the membership relations of a polynomial and to the ideal. A rough sketch of the algorithm is as follows (see [4] for a precise definition).

Let there be a total ordering among monomials and let a system of polynomial equations $E = 0$ be given. An equation can be considered a rewrite rule which rewrites the greatest monomial in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is $X > W > V > U$, a polynomial equation, $X - W + V = U$, can be considered to be the rewrite rule, $X \rightarrow W - V + U$. A pair of rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, is called a *critical pair*, since the least common multiple of their left-hand sides can be rewritten in two different ways. The S-polynomial of such a pair is defined as:

$$\text{S-poly}(L_1, L_2) = R_1\frac{\text{lcm}(L_1, L_2)}{L_2} - R_2\frac{\text{lcm}(L_1, L_2)}{L_1}$$

If further rewriting does not succeed in rewriting the S-polynomial of a critical pair to zero, the pair is said to be *divergent* and the S-polynomial is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called the *Gröbner basis* of the original system of equations.

Definition 3.1.1 (Gröbner basis [4])
The Gröbner basis $G(E)$ is a finite set that satisfies the following properties.

(i) $\mathcal{I}(E) = \mathcal{I}(G(E))$

(ii) For all f and g, $f - g \in \mathcal{I}(E)$ iff the reduced forms of both f and g by $G(E)$ are the same. Especially, $f \in \mathcal{I}(E)$ iff f is reduced to "0" by $G(E)$.

(iii) G is reduced *if it has no rules, where one rewrites the other.*

From Theorem 3.1.1, the reduced $G(E)$ can be regarded as being the canonical form of the solution of $E = 0$, because the reduced Gröbner basis with respect to a given admissible ordering is unique. Moreover, when $E = 0$ does not have a solution, $\{1\} \in G(E)$ is deduced from Corollary 3.1.1.

3.2 Satisfiability and Entailment

Based on the above results, we could determine satisfiability by using the Buchberger algorithm to incorporate the polynomial into the Gröbner bases as per Corollary 3.1.1. But the method of Definition 3.1.1(ii) is incomplete in terms of deciding entailment, since the relation between the solutions and the ideal described in Theorem 3.1.1 is incomplete. There are several approaches that solve the entailment problem: (a) Use the Gröbner basis of the radical of the generated ideal $\mathcal{I}$. (b) Add $p\alpha$ to the Gröbner basis as a negation of $p = 0$ and use the

Buchberger algorithm(where α is a new variable), and (c) Find n such that p^n is rewritten to 0 by the Gröbner basis of the generated ideal[5]. Unfortunately, these methods are computationally expensive, while the total efficiency of the system is greatly affected by the computation time in deciding entailment. Therefore, we determine the entailment by rewriting using a Gröbner basis from the viewpoint of efficiency, even though this method is incomplete. This decision procedure runs on the interface module parallel with the solver execution, as shown in Figure 1. Whenever a new rule is generated, the solver sends the new rule to the interface module via a communication stream. The interface determines entailment while storing (intermediate) rules to a self database.

3.3 Parallel Algebraic Solver

There are two main sources of parallelism in the Buchberger algorithm, the parallel rewriting of a set of polynomials, and the parallel testing for subsumption of a new rule against the other rules. Since the latter is inexpensive, we should concentrate on parallelizing the coarse-grained reduction component for the distributed memory machine. However, since the convergence rate of the Buchberger algorithm is very sensitive to the order in which polynomials are converted into rules, an implementation must be careful to select "small" polynomials early.

Three different architectures have been implemented; namely, a pipeline, a distributed, and a master-slave architecture [11, 10, 22]. Among them, the master-slave architecture is described, since it offers good performance.

The set of polynomials E is physically partitioned and each slave has a different part of them. The initial rule set of $G(E)$ is duplicated and assigned to all slaves. New input polynomials are distributed to the slaves by the master. Figure 2 shows the architecture.

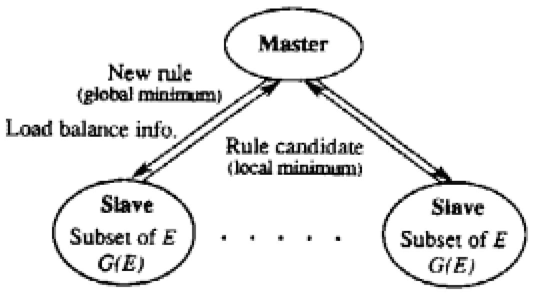

Figure 2: Architecture of the parallel solver

The reduction cycle proceeds as follows. Each slave rewrites its own polynomials by the $G(E)$, selects the local minimum polynomial from them, and sends its leading power product to the master. The master processor awaits reports from all the slaves, and selects the global minimum power product. The slave that receives the *minimum* message converts the polynomial into a new rule and sends it to the master, the master sends the new rule to all the slaves except the owner. If several candidates are equal power products, all candidates are converted to rules by owner slaves and go to final selection by the master.

To achieve load balance during rewriting, each slave reports the number of polynomials it owns, piggybacked onto leading power product information. The master sorts these numbers into increasing order and decides the order in which to distribute S-polynomials. After applying the unnecessary S-polynomial criterion, each slave generates the S-polynomials it should own corresponding to the order decided by the master. Subsumption test and rule updating are done independently by each slave.

Table 1 shows the results of benchmark problems to show the performance of this parallel algorithm. The monomial ordering is degree reverse lexicographic, and low level bignum (multiple precision integer) support on PIMOS is used for coefficient calculation. The method of detecting unnecessary S-polynomials proposed by [9] is implemented. The problems and their variable ordering are:

Katsura-4 : Katsura's system of 5 equations in 5 variables, using the ordering $U_0 < U_1 < U_2 < U_3 < U_4$ (see [3]).

Katsura-5 : Katsura's system of 6 equations in 6 variables, using the ordering $U_0 < U_1 < U_2 < U_3 < U_4 < U_5$ (see [3]).

Cyclic 5-roots : Cyclic 5 equations in 5 variables using the ordering $X_1 < X_2 < X_3 < X_4 < X_5$ (see [2]).

Cyclic 5-roots : Cyclic 6 equations in 6 variables using the ordering $X_1 < X_2 < X_3 < X_4 < X_5 < X_6$ (see [2]).

Speedup appears to become saturated at 4 or 8 processors except for "cyclic 6-roots". However, these problems are too small to obtain a good speedup because it takes about half a minute for all the processors to become fully operational as the unnecessary S-polynomial criterion works well.

Table 1: Timing and speedup of the master-slave architecture

	Number of processors				
Problems	1	2	4	8	16
Katsura-4 (sec)	8.90	7.00	5.83	6.53	9.26
	1	1.27	1.53	1.36	0.96
Katsura-5 (sec)	86.74	57.81	39.88	31.89	36.00
	1	1.50	2.18	2.72	2.41
Cyc.5-roots (sec)	27.58	21.08	19.27	19.16	25.20
	1	1.31	1.43	1.44	1.10
Cyc.6-roots (sec)	1430.18	863.62	433.73	333.25	323.38
	1	1.66	3.30	4.29	4.42

References

[1] A. Aiba, K. Sakai, Y. Sato, D. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, 1988.

[2] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor, *Proceedings of ISSAC'91*, pages 103–111, July 1991.

[3] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating groebner bases. *Symbolic Computation*, 2(1):83–98, 1986.

[4] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.

[5] L. Caniglia, A. Galligo, and J. Heintz. Some new effectivity bounds in computational geometry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes - 6th International Conference*, pages 131–151. Springer-Verlag, 1988. Lecture Notes in Computer Science 357.

[6] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, October 1990.

[7] A. Colmerauer. Opening the Prolog III Universe: A new generation of Prolog promises some powerful capabilities. *BYTE*, pages 177–182, August 1987.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, 1988.

[9] R. Gebauer and H. M. Möller. On an installation of Buchberger's algorithm. *Symbolic Computation*, 6:275–286, 1988.

[10] D. J. Hawley. The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver. Technical Report TM–713, Institute for New Generation Computer Technology, 1991.

[11] D. J. Hawley. A Buchberger Algorithm for Distributed Memory Multi-Processors. In *The first International Conference of the Austrian Center for Parallel Computation*, Salzburg, September 1991. Also in Technical Report TR-677, Institute for New Generation Computer Technology, 1991.

[12] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Prelimiary Results of CHIP with PEPSys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[13] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *4th IEEE Symposium on Logic Programming*, 1987.

[14] M. J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[15] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 51–74. Academic Press, 1990.

[16] K. Sakai and A. Aiba. CAL: A Theoritical Background of Constraint Logic Programming and its Applications. *Symbolic Computation*, 8(6):589–603, 1989.

[17] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[18] S. Sato and A. Aiba. An Application of CAL to Robotics. Technical Report TM–1032, Institute for New Generation Computer Technology.

[19] H. Sato T. Chikayama and T. Miyazaki. Overview of Parallel Inference Machine Operationg System (PIMOS). In *International Conference on Fifth Generation Computer Systems 1988*, pages 230–251, 1988.

[20] P. Senechaud. Implementation of a parallel algorithm to compute a Gröbner basis on Boolean polynomials. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[21] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[22] S. Terasaki, D. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama and A. Aiba. Prallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers. In *International Conference on Fifth Generation Computer Systems 1992*, 1992.

[23] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33(6):494–500, December 1990.

[24] J. P. Vidal. The Computation of Gröbner bases on a shared memory multi-processor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

A Practical Approach to the Global Analysis of Constraint Logic Programs

M.J. Garcia de la Banda
M. Hermenegildo

maria,herme@fi.upm.es
Computer Science Department
Technical University of Madrid

This paper presents and illustrates with an example a practical approach to the dataflow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. It is first argued that, from the framework point of view, it suffices to propose quite simple extensions of traditional analysis methods which have already been proved useful and practical and for which efficient fixpoint algorithms have been developed. This is shown by proposing a simple but quite general extension to the analysis of CLP programs of Bruynooghe's traditional framework. In this extension constraints are viewed not as "suspended goals" but rather as new information in the store, following the traditional view of CLP. Using this approach, a complete, constraint system independent, abstract analysis is presented for approximating definiteness information. The analysis is of quite general applicability since it uses in its implementation only constraints over the Herbrand domain. Some results from the implementation of this analysis are also presented.

(The research presented in this paper has been supported in part by ESPRIT project 5246 "PRINCE" and by CICYT project TIC91-0106-CE.)

Impose Constraints in a Multi-User, Dynamically Changing Environment (Extended Abstract)

Qing Ge

KBS Technology Inc.
350 Highway 7, suite 402
Richmond Hill, Ontario
L4B 3N2 Canada
E-mail: ge@ai.toronto.edu

Constraint logic programming has become a powerful tool for problem solving and information processing. A large scale, real world application implemented by using constraint logic programming methodology is described in this paper. Several interesting issues caused by the dynamic nature of the application are discussed.

The Development Workstation (DWS) provides an environment for IBM worldwide to collect and validate all business type product information throughout the product development cycle, and to easily extract the subsets of the information acquired for different business practices such as Product Assumptions & Commitments, Pricing, Product Announcement and so on. DWS, which differs substantially from the traditional information processing systems, is part of IBM "re-engineered" process. DWS will serve as the single resource for all IBM business-related product data and there will be thousands of the users worldwide to input information to and to obtain information from DWS. Various kinds of product data will start to be entered into DWS at the beginning of the development cycle. Additional information will continue to be added throughout the life of the product. The data for one product can be entered by different users from IBM Worldwide Development, IBM Worldwide Marketing, development labs, etc., either concurrently or over a rather long time period. The data are validated at data entry. Once accepted, the data must be guaranteed to be correct, consistent and legal (e.g., for product announcements). The kernel of DWS is a forward chaining inference engine which does reasoning with constraints. DWS is implemented mainly by Prolog. Only the I/O intensive operations are done through C programs.

The product data are not isolated. They related to each other and there are constraints that apply on them to restrict their value ranges and formats. It is the relationships and the constraints that make a set of data meaningful for a particular application domain and that, on the other hand, complicate the issue of keeping the data correct and consistent. Thus, DWS needs two kinds of knowledge to accomplish its task: knowledge about relationships and knowledge about constraints. The former is provided by an entity-relationship database and the latter represented in the format of constraint rules.

A rich constraint rule language has been developed to express conditions (with quantifiers) and actions. A large amount of expertise is obtained from the human experts and provided to DWS as the constraint rules by a knowledge engineering group. The knowledge about constraints is applied every time a piece of information is entered into DWS. The following issues are encountered when DWS attempts to impose the constraints through its inference engine:

First, DWS stores the information acquired on a global repository. Even with privilege checking, it can happen quite frequently that multiple users who have proper access rights need to change the data regarding to the same product at the same time. Locking up the repository to enforce sequential access is not practical because DWS is the only resource for many applications and for thousands of the users. Giving each user an equal time slice is not a satisfactory solution either because one piece of data can be changed only by one user at any time. Even though the users may only need to work on the different pieces of data, these data are related to each other and constraints need to be applied against the information provided by the user. Synchronization in this case is rather complicated and mutual restriction (similar to deadlock) may occur. Therefore, a semi-distributed paradigm is employed which allows local data input and constraint imposition first and global data merge including automatic error and conflict checking later.

Secondly, any data collected by DWS can be modified by the user. This implies that the restrictions imposed by certain constraint rules may need to be lifted. For example, the value for data A causes a constraint rule C to be fired and puts a restriction R on the data B. The value of data A is then changed so that the condition of the rule C no longer holds. Consequently, restriction R is not applicable any more. However, how to remove restriction R is a problem. Due to the large number of rules in the system, it is not a good approach to examine every rule that cannot be fired to check if there are restrictions that have been imposed by it and that should be removed. In general, this is a problem of undoing the effects of constraint rules when changes make the rule conditions no longer satisfied. Instead of doing and undoing action effects, the action part of a constraint rule is indexed in a special way based on the data to which the constraint rule puts restrictions when being fired. When the data that contribute to the condition part of such a rule are changed, the rule, although the condition can be matched, does not actually fire. Only the data that are in the action part can trigger such a rule and appropriate restrictions will be applied dynamically. In this way, when the condition of a rule cannot be satisfied, there is no need to clear the restrictions that may be imposed by it.

Thirdly, data can be entered into DWS in arbitrary order although the system can prompt for all the information in a systematic and smooth manner. It is not always possible to connect together the information entered at random. As a result, fragments of data may float in the system. However, in order to find problems as early as possible and as many as possible, efficient partial constraint imposition needs to be supported. When DWS receives a piece of information, it attempts to connect it with other existing information in the system to the maximum extent and imposes the constraints within this maximum range. When more information or fragments can be connected, more restrictions are applied incrementally. Such a process is repeated until all information is connected and no information is missing.

Fourthly, as in any large systems, there are a large number of constraint rules in DWS. To quickly find applicable constraints in any situations is one of the most important factors that have to be considered for such a real world project. A special indexing mechanism is used for this purpose, which also takes into account the arbitrary order data entries. To further improve system performance (since the user uses the system interactively), a set of heuristic rules are developed based on the availability of the information required to apply the constraints. Since multiple constraints can be applied for a given situation, conflict resolution is also necessary. Conflicts are resolved at two levels. The constraints chosen are first divided into pre-prioritized categories according to their actions and then within each category specificity strategy is used.

The information is DWS can be changed constantly by different users. Proper constraints have to be applied effectively and efficiently to ensure, all the time, the correctness and consistency. Changing one data usually results in changing a group of related data. Ripple effect is another interesting issue that DWS handles well, especially in determining the scope of affection.

The success of DWS encourages us to use the methodology and technology of constraint logic programming in other application areas in which constraints are applied to filter large amounts of information, such as large organizations, law firms and government offices.

Constraint Logic Programming with Priority
– extended abstract –

Takashi Hattori
RIMS, Kyoto University
hattori@kurims.kyoto-u.ac.jp

March 19, 1992

Constraints are widely investigated recently, but over-constrained systems can be handled as mere contradictions in most cases. However, when we would like to describe the notions of default values or preferences by means of constraints, it must be necessary to somehow produce useful information from over-constrained systems because default constraints or preferred constraints usually lead to a over-constrained system. One of such attempts is the constraint hierarchy introduced by Borning. It is not, however, fully satisfiable especially when it is combined with the constraint logic programming (CLP) scheme since the constraint hierarchy is of a different nature from the logic programming paradigm. The main aim of this paper is to define the *constraint logic programming with priority (CLP/P)* as a natural extension of CLP in order to deal with such notions as default values and preferences in the framework of logic.

Assume that there are three disjoint sets of predicate symbols $Pred_C$, $Pred_P$, $Pred_Q$. We call them *constraint predicates*, *control predicates*, and *priority predicates* respectively. The *program* of CLP/P is a pair $\langle P, Q\rangle$ where Q is a set of logical formulas containing only priority predicates and P is a set of formulas of the following form:

$$A :- B_1, \cdots, B_m, D_1 : C_1, \cdots, D_n : C_n \quad (m \geq 0, n \geq 0) \tag{1}$$

in which $A, B_1, \cdots, B_m$ are positive literals containing control predicates, $C_1, \cdots, C_n$ are logical formulas containing constraint predicates and $D_1, \cdots, D_n$ are positive literals containing priority predicates. We call P a *control part*, and Q a *priority part*.

The intuitive meaning of the formula (1) is almost same as that of CLP except that, given a model M of the priority part, a constraint C_i is taken into account only when the associated priority literal D_i is true in M. This is logically represented by the formula:

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge (\neg D_1 \vee C_1) \wedge \cdots \wedge (\neg D_n \vee C_n). \qquad (2)$$

By varying the model M of the priority part, we can get a set of different solutions each of which is associated to a certain model of the priority part.

For example, suppose we have three constraints:

$$\begin{aligned} C_1 &: a \leq b \\ C_2 &: b \leq c \\ C_3 &: c \leq a \end{aligned}$$

and CLP program:

$$p(a,b,c) \text{ :- } C_1, C_2, C_3.$$

If we give a query :- $p(0,b,c)$, the answer is $b = 0$ and $c = 0$. However, if we give a query :- $p(0,1,c)$, then constraints C_1, C_2, C_3 and constraints $a = 0$, $b = 1$ imposed by the query lead to a contradiction from which no answer can be obtained. In contrast to that, suppose we have the same constraints and CLP/P program $\langle P, Q \rangle$ where

$$\begin{aligned} P &= \{p(a,b,c) \text{ :- } q_1 : C_1, q_2 : C_2, q_3 : C_3\} \\ Q &= \{q_1 \wedge (q_3 \rightarrow q_2)\}. \end{aligned}$$

In this time, for the given query :- $p(0, b, c)$, we will get three solutions each of which is associated to the model of Q as follows:

model of Q	solution
$\{q_1\}$	$b \geq 0$
$\{q_1, q_2\}$	$b \geq 0 \wedge b \leq c$
$\{q_1, q_2, q_3\}$	$b = 0 \wedge c = 0$

In the same way, for the query :- $p(0, 1, c)$:

model of Q	solution
$\{q_1\}$	no limitation
$\{q_1, q_2\}$	$c \geq 1$
$\{q_1, q_2, q_3\}$	contradiction

Shown as above, the greater model we select, the more constraints are taken into consideration. It can also be seen that $q_3 \rightarrow q_2$ implies that C_3 is taken into account whenever C_2 does. In other words, a priority predicate which is true in a smaller model represents higher priority. Note that the empty priority part involves all combination of constraints while it is expected we are usually interested in some of them, therefore we should give a priority part which can effectively reduce the search space. In addition, we can use variables as arguments of priority predicates in order to give priority dynamically. For example,

$$\begin{aligned} P &= \{p(a, b, c, x) :- q_1(x) : C_1, q_2(x) : C_2, q_3(x) : C_3\} \\ Q &= \{q_1(x) \wedge (q_3(0) \rightarrow q_2(0)) \wedge (q_2(1) \rightarrow q_3(1))\} \end{aligned}$$

implies that C_2 takes priority over C_3 for the query :- $p(0, 1, c, 0)$ while C_3 takes priority over C_2 for the query :- $p(0, 1, c, 1)$.

Let B_P be the Herbrand base of control predicates, $\mathcal{M}_Q$ be a set of all models of a priority part. A mapping $\alpha : \mathcal{M}_Q \longrightarrow (B_P \longrightarrow \{\mathrm{t}, \mathrm{f}\})$ is a model of CLP/P program $\langle P, Q\rangle$ if and only if, for any model $M \in \mathcal{M}_Q$ and any ground instance

$$A :- B_1, \cdots, B_m, D_1 : C_1, \cdots, D_n : C_n$$

of a formula in P, $\alpha(M)(B_i) = \mathbf{t}$ $(1 \leq i \leq m)$ and $M(D_i) = \mathbf{t} \Rightarrow C_i$ $(1 \leq i \leq n)$ implies $\alpha(M)(A) = \mathbf{t}$. We can show that there exists the least model which coincides with the least fixpoint of a transformation function as shown in logic programming.

It is possible to define a refutation procedure for CLP/P by adding an equivalence class of $\mathcal{M}_Q$ to each goal where the equivalence class is classified by truth values of priority predicates occurring in the goal. The reason why we use an equivalence class of models instead of a model itself is that $\mathcal{M}_Q$ may be infinite in general. Since choices of the equivalence classes are not uniquely determined at each step of refutation, backtracking is needed in case the chosen equivalent class lead to a contradiction. We can show completeness of the refutation procedure for CLP/P.

Evaluation of solutions are the most popular method to handle default values or preferences. CLP/P program can be seen as a special kind of evaluation function whose range is $\mathcal{M}_Q$ instead of $\mathbf{R}$. From this point of view, the expected solutions are those associated to maximal elements of $\mathcal{M}_Q$ to the extent that do not lead to contradictions. Since $\mathcal{M}_Q$ is not totally ordered, there can be more than one local maximum, which bring about a well-known problem to search the global maximum. However, finding one of local maxima is satisfactory in some cases. For example, suppose that we would like to design a plan interactively with assistance of a set of design constraints. It is not convenient that slight changes of input make output to change busily, rather we prefer conservative behavior even if it is not globally optimal. Assume that a series of query is given where each query is slightly different from the predecessor, then we can quickly find a local maximum by utilizing a equivalence class of models obtained by refutation of the preceding query. This can be seen as simulation of hysteresis.

Anticipatory Pruning Networks in CLP language

Geun-Sik Jo *and Ken McAloon†

*In-Ha University and Center for Artificial Intelligence Research,Korea
†Brooklyn College and CUNY Graduate Center,USA

1 Introduction

In this research, forward checking is implemented through a kind of forward chaining in Production Systems. The APN propagate constraints inconsistent with the current environments. The inconsistent constraints can be found by checking the consistency with the current environments in parallel. We used the Rete style of rule compilation method to implement the forward checking mechanism. To develop and test the idea, we worked with 2LP, a CLP system with propositional logic and linear constraints. The APN is somewhat analogous to the consistency technique over discrete domain in CHIP [3]. However, the APN provides forward checking mechanism for constraints over continuous as well as discrete domain for the constraint solver in logic programming environment. In addition, the APN here preserves the operational semantics of the language.

2 APN algorithm

The APN algorithm computes the consequences of the constraints inconsistent with the current environment. The APN also explicitly saves the consequences of the inconsistency in the network. Furthermore, the APN algorithm can undo the processes of computing and saving the consequences of the inconsistent constraint to meet the backtracking behavior of logic programming.

2.1 Rule compilation in the 2LP system

The compilation of rules in the 2LP system is analogous to the compilation done in the OPS5 system. The condition elements of the rule here have only the name of the class and do not have any attributes in it if we consider it in the OPS5 context.

The linearization of the network in an efficient machine executable form is important for real applications. For the basic human readable forms of linearized network in APN, the following five different types of nodes are necessary to linearize the APN.

*Research partially supported by NSF and CUNY Grant
†Research partially supported by NSF Grant IRI-8902511

1. (Fork label) ; The label represents the position of a node for another successor.
2. (EQ atom) ; One-input node for testing for equality.
3. (AND Left-memory Right-memory) ; Two input node which has the left memory and right memory. The left-memory takes input from the previous node and the right memory takes input from the Merge node.
4. (Join label) ; This node is the same as the *goto* instruction in the von Neumann machine architecture, but affecting only the right-memory of the AND-node.
5. (Update rule-id head) ; Update the counter of head of the rule and the list of available rules for selections.

2.2 Anticipatory Pruning Networks

There are two functionally different components of computation in APN. One is finding the constraints inconsistent with the current environment. The other is forward reasoning with these inconsistent constraints. In testing consistency, an incremental version of the Simplex algorithm is used. By propagating inconsistency, all deterministic goals are selected and resolved at once. Moreover, failure can be detected easily if an unsolvable atom occurs in the goal list.

2.2.1 Consistency checking with the current environments

Let us consider the consistency checking in the 2LP system. Let the active constraints (*AC*) be the constraints which are enforced by the logic interpreter. Let us call the set of all constraint for the given program the *quick* constraints (*QC*). The quick constraints are also called the *quick list*. Finally, let us call the constraints inconsistent with the current environment the dead constraints (*DC*). Whenever the constraint is enforced by the logic interpreter, the system checks consistency with the current environment. The environment here is the Simplex table which is formulated by the AC. If the enforced constraint is consistent, the current environment is updated. Furthermore, the constraint is removed from the *QC* and added to the *AC*. To find out the inconsistent constraints with the current environment, the system performs consistency checking with the rest of the *QC*. If the enforced constraint is not consistent, it is also removed from the *QC* and added to the *DC*. Then the consistency checking in 2LP is to check the consistency of all the constraints in *QC* with the newly updated environment. Therefore, everytime the current environment is updated, consistency checking is performed. The consistency checking here can be done in parallel.

Instead of checking consistency on all the constraints in the *QC* with the current environment, we can have the system choose the several different sets of constraints. One of them is to compute the relevant constraints with the goal list which are the constraints set that can be derivable from the left most branch of the search space. This is to detect the early failure in the current direction of search space. This is called Partial Anticipatory Pruning. If we check the consistency with all constraints in *QC*, it is called Full Anticipatory Pruning. The benchmark results for each technique are demonstrated in the next section.

2.2.2 Inconsistency propagations in APN

Since we are interested in the inconsistent constraint set to prune unnecessary searching, all the two-input memories are initially set to be available, which means that every rule is available for selection at the beginning. As the logic interpreter finds the inconsistent constraints with the current constraint environment, the system generates the negative token and the interpreter of APN propagates the negative token down to the APN. As a result of propagating the negative token, some rules may be ruled out from the knowledge base to remove the unnecessary search. The interpreter also can undo what has been done by propagating the positive tokens to take care of the backtracking behavior of logic programming style of control mechanism.

The APN interpreter basically performs the following *inconsistency propagation-update cycles*.

1. Match The condition elements in the body of rules are matched against the atoms which refer to the inconsistent constraints with the current environment.
2. Propagating and updating the procedure table The successfully matched atoms are propagated depending upon the inconsistency found in the condition elements in the same rule. If the inconsistency is propagated successfully, the counter associated with the head is decreased by 1 and the index of the rule associated with the head is removed from the available rule list for that head. If the backtracking occurs, the counter associated with the head is increased by 1, and the index of the rule is added to the available rule list.
3. Selection The head with the counter changed to 0 or the counter changed to 1 from 0 is selected for the further propagation.
4. go to 1.

The description of implementation in the APN interpreter is provided by the Lisp code in the Appendix A in [7].

3 Measurement on APN

This section presents the effectiveness of *APN* in *2LP* system. The table 1 provides benchmark results in solving some puzzles which are traditionally considered as integer programming problems. The table 2 provides the benchmark results for some linear programming problems which can have integer or rational variables in the problem domains.

In this benchmark, the number of node visited which is represented as *Node* in the table is counted cumulatively during the tree search process. The number of constraints checks which is represented as *Constraint* in the table is also counted cumulatively. We counted the number of constraints appearing on the right hand side of rule and added up cumulatively as the rule is rewritten. The consistency checking in APN is counted as one constraint check for each node visited. The number of constraint checks here is a cumulative addition of the number of constraits appearing on the right hand of rule and the consistency checking. However, we regarded that the deterministic rewriting occurs as part of *APN* activity at a given node.

	NoAPN		*PartialAPN*		*FullAPN*	
	Node	*Constraint*	*Node*	*Constraint*	*Node*	*Constraint*
8-Queens	23,359	19,904	8,959	13,097	4,916	8,054
SEND+MORE=MONEY	1,641	1,509	144	243	4	12
GERALD+DONALD=ROBERT	11,317	10,472	1,203	1,798	438	739

Table 1: Integer programming problems

	NoAPN		*PartialAPN*		*FullAPN*	
	Node	*Constraint*	*Node*	*Constraint*	*Node*	*Constraint*
Six months blending problem	11,863	17,172	3,991	8,770	2,632	6,693
Mining problem	33,598	48,563	10,736	26,923	9,753	24,722

Table 2: Linear programming problems

The *NoAPN* in this benchmark is the case which the *APN* is not active. Therefore, the basic search strategy is the same as the Prolog search strategy. In the the *FullAPN*, we check the consistency with all the bottom-most leaves which have not been explored yet. The *PartialAPN* is to check the consistency with the constraints set that can be derivable from the left most branch of search space.

All the problems described table 1 can be viewed as the integer programming problems. However, many problems in linear programming are not pure integer programming problems. Some examples of the mixed linear integer programming problems are drawn from [9] and evaluated in the table 2 to test the effectiveness of APN in 2*LP* system. These two benchmarks in the table 2, Six month blending problem and Mining problem, are optimization problems. These mixed integer linear programming problems are not amenable to the consistency technique in CHIP because of the presense of continuous variables. In a blending problem, it is very reasonable to mix 1 1/2 gallons of vegetable oil with 2 1/3 gallons of non-vegetable oil to maximize the profit. Therefore, the concept of CSP cannot be applied to this kind of linear programming problem. However, with APN, it does not matter whether the domain is discrete or continuous, since we can use Simplex based consistency checking. We are required to use either none or more than 20 tons of an oil. This kind of "or" constraint can be expressed naturally by means of logic in the 2*LP* system.

4 Conclusion

The study of APN in 2LP system lead us to make the following conclusions.

1. The APN introduces Forward Checking mechanism for constraints over continuous and discrete domain. It also provides very deep look ahead for the Simplex-based constraint solver. This property, Forward Checking mechanisms independent of domains, is highly desirable in linear programming environments if we consider that many linear programming problems are mixed integer linear programming.

2. By using the APN, we can exploit the OR-parallelism, i.e., parallelism in consistency checking, in constraint optimization problems effectively in the 2LP system [8]. As we have pointed out in the above section, the APN is very useful to solve the optimization problems. Especially if we want to find an optimization solution which is a near-failure, the APN can reduce the tree search space drastically as shown in [7].

3. The APN, which is a domain independent forward checking mechanism, opens a possibility of integrating the Simplex-based constraint solver with CSP.

4. The APN can address the partial solution naturally since the APN can be viewed as the propositional expert system and the current working memories in APN are the inconsistent constraints found so far with the current environment.

References

[1] Krzysztof R. Apt, Howard A. Blair, Adrian Walker, Towards a theory of declarative knowledge, *Foundations of deductive databases and logic programming*, Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp 89-148.

[2] Jim Cox, Ken McAloon and Carol Tretkoff, Computational Complexity and Constraint Logic Programming Languages, Brooklyn College Computer Science Technical Report, No.90-4.

[3] M. Dincbas, Pascal Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of International Conference on Fifth Generation Computing Systems*, 1988.

[4] On the efficient implementation of Production systems, *Ph. D. Thesis*, Carnegie-Mellon University, 1979.

[5] Rete A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence* 19, Sept. 1982,pp 17-37.

[6] H E Robert M. Haralick and Gordon L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence* 14, 1980,pp 263-313.

[7] *Anticipatory Pruning Networks and Minimal 2LP*, Ph.D. Thesis, CUNY, Feb. 1991.

[8] K. McAloon and C. Tretkoff, 2*LP* A Logic Programming and Linear Programming System, Brooklyn College Computer Science Technical Report No. 1989-21.

[9] *Model building in mathematical programming*, John Wiley & Sons,New York, 1985.

Approximation in the Framework of Generalised Propagation *

(Extended Abstract)

Thierry Le Provost

European Computer-Industry Research Centre
Arabellastr. 17, D-8000 Munich 81, Germany

e-mail: thierry@ecrc.de

1 Introduction

Generalised Propagation [5] aims at formalising and generalising constraint propagation techniques. In this scheme, propagation constraints are always defined *declaratively*, in contrast to so-called *reactive* programs.

Although Generalised Propagation has already been successfully implemented and applied to nontrivial search problems, its expressiveness and efficiency were found insufficient in many cases.

This presentation introduces and motivates an essential extension to the scheme, dubbed *Approximate Generalised Propagation*, where the introduction of *approximation languages* dramatically increases the expressiveness and efficiency of propagation constraints, while fully preserving their declarativeness.

2 Generalised Propagation

2.1 Origin and Motivation

Propagation or *consistency* techniques were first introduced into the context of logic programming by the CHIP system [1, 9], which is currently the mainstay of the ESPRIT Project CHIC [3]. It provides a rich set of symbolic and arithmetic "built-in" propagation constraints for efficiently stating and solving complex search problems.

CHIP also allows for some means of defining new propagation constraints, but these additions have proven either hard to reason about for non-expert programmers

*This work is partially funded by the ESPRIT project CHIC, Nr. 5291 (e.g. *demon rules*), or conceptually simple but rather inefficient (*lookahead declarations*). Besides, the constraint propagation mechanism in CHIP appears to be tied to a particular computation domain (namely, finite-domain variables).

In contrast, the Generalised Propagation mechanism applies to any language in the CLP Scheme of Jaffar, Lassez *et al.* [4], and allows programmers to designate *any* subgoal in a program or query as a (generalised) propagation constraint.

2.2 Informal Description of Generalised Propagation

Semantically, such a designated subgoal, together with its declarative predicate definition in $CLP(X)$, *uniquely* identifies a closure operator on the partially ordered set of "basic constraints", as defined by the computation domain X. In other words, a subgoal earmarked as a generalised propagation constraint approximates its conjunction with the current basic constraint store whenever possible, and *as precisely as possible*, given the expressive power of the computation domain (i.e., the basic constraints). The inference mechanism is defined so that the set of logical answers to the subgoal is respected. This refinement of the basic constraint store usually allows for very large reductions in the size of the $CLP(X)$ search tree.

The concept of closest approximation of "complex" (as yet unsolved) subgoals by "simple" basic constraints happens to be universally applicable, and bears no special connection with finite domains, as was formerly believed [8]. For instance, one may apply Generalised Propagation to $CLP(HU)$ (or even to impure Prolog), thereby performing a novel kind of propagation *on structured terms* [1]. One may also specialise the concept back to its CHIP source; this produces a powerful notion of declaratively defined (i.e., *program-defined*) propagation constraints that deal with both finite-domain variables and structured terms.

Operationally, generalised propagation subgoals are repeatedly submitted to a metalogical procedure that enumerates subgoal answers, and generalises over them. The implementation of generalised propagation over $CLP(X)$ therefore requires an operation of *constraint generalisation*, which can informally be seen as dual to the customary one of *constraint solving*. Practical algorithms for implementing generalised propagation only perform an *implicit* enumeration of subgoal answers, thereby allowing an efficient production of the closest approximation to many subgoals even when their search-trees are large or even infinite [5] [2].

2.3 Motivating Approximate Constraints

However, a shortcoming of Generalised Propagation as described above is its inflexibility. Each designated subgoal denotes exactly *one* generalised propagation constraint. Such a constraint is sometimes too expensive to compute by the implicit enumeration procedure outlined above, when balanced against the pruning in search space that it affords.

The resulting overall efficiency might in such cases fail to meet expectations from the programmer. Indeed, overly strong propagation constraints can sometimes lead to a net loss of efficiency, compared with the original $CLP(X)$ program !

[1]The restricted size of this abstract confines the use of program examples to the presentation.
[2]However, Generalised Propagation is uncomputable for nontrivial computation domains.

3 Approximate Generalised Propagation

3.1 How to Approximate Declarative Constraints ?

Some broader notion of *approximation* is therefore needed to cope with excessively strong or expensive generalised propagation constraints. Briefly considering various alternatives, and motivating our current choice for a certain notion of *post-approximation*, are the subject matters of this presentation.

As generalised propagation constraints are *only defined declaratively* (by the relation over X denoted by some predicate definition in $CLP(X)$), we are willingly deprived of any opportunity for *explicitly* weakening the activity of a constraint.

So-called *reactive* approaches for expressing propagation constraints, such as the *Ask and Tell Scheme* [6], or extension of it such as $cc(FD)$ [10], feature explicit synchronisation operators that have been omitted from the Generalised Propagation Scheme. Our contention is that explicit synchronisation (*Ask* and commitment operators) may lead to programs that are hard to write and verify. In particular, such *reactive* languages allow for the unwitting definition of indeterministic propagation constraints. A lesser but severe problem with reactive definitions is the absence of guarantee that the explicitly defined constraint will "behave well" (i.e., that it will indeed be a closure operator).

In other words, Generalised Propagation was designed with the objective of avoiding explicit synchronisation and other traditional reactive operators. Consequently, approximation cannot be achieved by e.g. modifying program guards.

3.2 Declarative Notions of Approximation

The solution we propose here is faithful to the declarative approach taken by Generalised Propagation, in that approximation is performed through *approximation operators*. They just happen to be sorts of "anti-propagation constraints". Conceptually, a "most specific" generalised propagation constraint, as defined by a $CLP(X)$ predicate, is composed with an built-in approximation operator that *discards* (filters out) some of the information produced by the "most specific" constraint.

We are now left with the crucial choice of what to apply these approximation operators to. Space limitations bar us from considering several obvious alternatives, such as applying approximation in a static way to the $CLP(X)$ program itself. Connections with abstract interpretation schemes will also be omitted here. Instead, we shall focus on two possibilities: *pre-* and *post-approximations.*

Pre-approximations are applied to the current store of basic constraints *before* the generalised "most specific" propagation constraint. Post-approximations, conversely, are applied to the *result* of applying the generalised propagation constraint to the constraint store.

3.3 Post-Approximations Seem to Be Better

Provided that certain conditions hold (mainly, that propagation constraints and approximation operators both be closure operators in a partially ordered structure of basic constraints), any composition of a "most specific" constraint with an *a* *posteriori* approximation operator "behaves well". That is, all properties that can be expected of a Generalised Propagation constraint continue to hold for such a post-approximated constraint. (This amounts to showing that the resulting post-approximated constraint is still a closure operator.)

Such nice things cannot be said about pre-approximations, and the latter appear quite insufficient in terms of defining useful and classical approximate propagation constraints.

A decent implementation should not compute first the "most specific" constraint, and *then* discard part of the produced information through the post-approximation operator. A simple lattice-theoretical result allows for the design of implicit enumeration algorithms where the computation of the constraint's result and its approximation can be soundly interleaved (the approximation is somehow *pushed down*). This leads to the sought-after increase in efficiency of approximate propagation constraints, compared to their "most specific" versions.

3.4 Applying Approximate Generalised Propagation

Two interesting cases of seemingly wasteful, but practically efficient, post-approximate propagation constraints are *pure tests* (where the approximation consists in discarding *all* information but failure) and *generalised forward checking* constraints (powerful generalisations of CHIP's *forward-checking* constraints, which sit idle in the resolvent until they can entirely express themselves as basic constraints from the computation domain).

As for putting these ideas about "declaratively approximated generalised propagation constraints" into practice, three computation domains spring to mind:

- Datalog (no function symbols);
- $CLP(HU)$ (in practice, full Prolog);
- $CLP(FD)$ (Prolog with finite-domain variables, which can be viewed as an extended and declarative "symbolic CHIP").

Approximate Generalised Propagation is currently implemented on all three, hosted by SEPIA and ElipSys [7, 2], two extensible Prolog systems developed at ECRC. Dramatic reductions in search space size have been obtained for several large applications, some of them combining Approximate Generalised Propagation with regular CHIP constraints [2].

4 Related and Future Work

We are investigating several open questions, mostly about the expressive power of Approximate Generalised Propagation (e.g., are our definable constraints the "desirable" ones for problem-solving in practice ?).

A comparison contrasting the expressive power of our declarative propagation scheme with "reactive" schemes such as $cc(FD)$ is being attempted, although the matter will require further discussions between "declarative" and "reactive" propagation constraints proponents !

References

[1] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems (FGCS'88)*, Tokyo, Japan, December 1988.

[2] M. Dorochevshy, L.-L. Li, M. Reeve, K. Shuermann, and A. Véron. ElipSys: Real Applications Need More Than Parallelism !. *Proceedings of the FGCS'92 Workshop on Future Directions of Parallel Programming and Architecture*, Tokyo, Japan, June 1992.

[3] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, E. Monfroy, and M. Wallace. The CORE Approach to Constraint Logic Programming. In *Proceedings of the FGCS'92 Workshop on Constraint Logic Programming*, Tokyo, Japan, June 1992.

[4] J. Jaffar, and J.-L. Lassez. Constraint Logic Programming. *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, Germany, January 1987.

[5] T. Le Provost, and M. Wallace. Domain-Independent Propagation. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92)*, Tokyo, Japan, June 1992.

[6] V.A. Saraswat. Concurrent Constraint Programming Languages. *PhD Thesis*, Carnegie-Mellon University, Pa, January 1989.

[7] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - An Extendible Prolog System. In *Proceedings of the 11th World Computer Congress (IFIP'89)*, San Francisco, Ca, August 1989.

[8] P. Van Hentenryck, and M. Dincbas. Domains in Logic Programming. *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, Philadelphia, Pa, August 1986.

[9] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. *Logic Programming Series, MIT Press*, Cambridge, Ma, 1989.

[10] P. Van Hentenryck, and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. In *Proceedings of the PLILP'91 Conference*, Passau, Germany, August 1991.

Constraint Satisfaction and Optimization Using Sufficient Conditions for Constraint Violation

Fumihiro Maruyama, Yoriko Minoda, Shuho Sawada, and Yuka Takizawa

Knowledge Processing Laboratory
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
Phone: +81-44-754-2661
Fax: +81-44-754-2664
E-mail: superb@flab.fujitsu.co.jp

1. Introduction

Many problems in engineering and OR can be formulated as discrete constraint satisfaction or optimization problems, e.g., LSI logic design, cutting-stock, and job-shop scheduling problems. Our approach to such problems represents constraints with inequalities and equalities.

In the formulation of LSI logic design that follows, variables correspond to functional blocks to be implemented with cells, or LSI parts. Each variable x_i takes an implementation alternative c_{ij} (j=1, 2, 3, ...) as its value. The following system of inequalities represents time constraints:

$$\sum_{i=1}^{\text{blocks}} a_{ki} d_k(x_i) \le b_k \quad (k=1, 2, \ldots, m) \tag{1}$$

Attribute d_k represents the delay time within the functional block along the kth path, on which a constraint on maximum total delay is imposed. b_k represents the limits of maximum delays. a_{ki} is 1 if the kth path passes the ith block; otherwise it is 0. The area-optimization problem with time constraints is to find the values for each variable, i.e., to choose the implementation alternatives for each functional block, that satisfy (1) and minimize the objective function (2), where attribute d_0 represents the functional block area in terms of the gate count:

$$\sum_{i=1}^{\text{blocks}} d_0(x_i). \tag{2}$$

Figure 1 shows an example of a job-shop scheduling problem with four jobs and three machines. Each job consists of three operations processed by each machine, whose sequence is given for each job. Every job can be started at time 0. In the figure, each operation is represented by a rectangle containing three digits representing, first, the job number, second, the operation sequence number within the job, and, third, the machine number. Numbers at the left and right below each rectangle represent operation start and end times. The problem is to determine start times for each operation that satisfy the constraints below and make the last end time as early as possible. We assume that times take only integers.

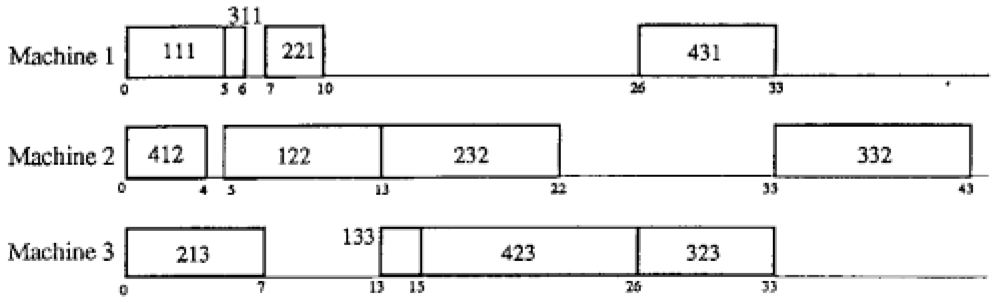

Figure 1: Example of job-shop scheduling problem

We use variables that take as their values start times for each operation. Constraints fall into the four categories:

(a) Constraints on earliest possible start times

Since each job can be started at time 0, a constraint in this category is $x_{111} \geq 0$.

(b) Constraints on due times

Since operation 133 is the last operation of the first job, a constraint in this category is $x_{133}+l_{133} \leq E$, where E is the due time of the job and l_{133} is the operating time of the operation.

(c) Constraints on precedence relationships

The constraints in this category come from the sequence of operations in a job. This example has eight, one of which is $x_{111}+l_{111} \leq x_{122}$.

(d) Constraints on mutual exclusiveness

The constraints in this category take the following form,

$$x_\alpha+l_\alpha \leq x_\beta \vee x_\beta+l_\beta \leq x_\alpha \tag{3}$$

where α and β are two operations to be processed by the same machine.

2. Constraint Satisfaction and Optimization

We use sufficient conditions for constraint violation, which we call nogood justifications (NJs) to prune the search space. An NJ is either an inequality or a conjunction of inequalities. We present an algorithm for discrete constraint satisfaction. NJs are generated and stored during execution and used to prune unexplored subtrees of the search tree. To optimize an objective function, the algorithm iteratively applies itself to the corresponding constraint satisfaction problem.

2.1 Nogood justification

We define nogood justifications inductively as follows:

Definition: *Nogood justification (NJ)*

(i) The logical negations of given constraints are NJs, or *default NJs*.

(ii) If at least one NJ, e.g., D_j, holds for each value c_{ij} of a particular variable x_i (j=1, 2, 3, ...), the logical product of D_j' (j=1, 2, 3, ...) is also an NJ, where D_j' is a substitute of D_j with each occurrence of x_i replaced with c_{ij}. If no value is assigned to a variable in an NJ, i.e., the NJ refers to an *indefinite* variable, we regard the NJ as false. End of Definition[1]

The NJs defined above are sufficient conditions for constraint violation in the sense that if any NJ holds, at least one default NJ will become true no matter what values are assigned to currently indefinite variables. When pruning an unexplored subtree during execution, this guarantees that no solution is among the leaves of the subtree.

2.2 Constraint satisfaction algorithm

The following algorithm satisfies all given constraints using NJs:

Algorithm

Step 0 Assign a value to each variable. Go to **Step 1**.

Step 1 Check for a satisfied NJ. If one is found, go to **Step 2**. Otherwise, go to **Step 3**.

Step 2 Select a variable among those to which values are assigned. Change its value from the current one to another. If there is a value that makes all NJs false, go to **Step 3**. Otherwise, generate a new NJ by **Definition** (ii) and go to **Step 5** with the variable changed to indefinite.

Step 3 If any indefinite variables are left, go to **Step 4**. Otherwise, all constraints have been satisfied and a solution found, so exit.

Step 4 Select an indefinite variable and assign a value to it. If a value makes all NJs false, return to **Step 3**. Otherwise, generate a new NJ by **Definition** (ii) and go to **Step 5** with the variable remaining indefinite.

Step 5 If no variables are in the generated NJ, constraint satisfaction has failed, so exit. Otherwise, return to **Step 2**. End of Algorithm

In **Step 4**, select the variable last made indefinite. The variable in **Step 2** is selected one of

[1] We can pursue a dual argument by considering necessary conditions for constraint satisfaction, which include given constraints, and their logical sums, instead of NJs and logical products.

two ways: the fixed order method and the criterion-based selection method.

2.3 Optimization

For optimization problems with an objective function to be minimized or maximized, a new constraint corresponding to the objective function is added to the original constraints. For example, the following constraint is added corresponding to (2):

$$\sum_{i=1}^{blocks} d_0(x_i) \leq b_0.$$

The job-shop scheduling problem has counterparts such as $x_{133}+l_{133} \leq E$.

The above algorithm is then applied to the augmented constraint satisfaction problem. It starts with an appropriate initial value of b_0 (or E). As long as a solution is obtained, b_0 is updated to

(the value of (2) for the solution just obtained) - ε

where ε is a sufficiently small number. If (2) takes only integers, ε can be set to 1. The algorithm is applied again from **Step 1**. For the job-shop scheduling problem, E is updated to

(the last end time for the schedule just obtained) -1.

When constraint satisfaction fails, the last solution obtained is the optimal solution. The algorithm checks all NJs generated thus far, including those generated before b_0 (or E) was updated to the current value.

3. Evaluation

Table 1 shows some of the results from a SPARCstation for the two-dimensional cutting-stock problem [Dincbas 88] with 92 *configurations* (values for each variable). Although the time grows exponentially with the number of lots (problem size), which seems inevitable with such a combinatorial problem, note that the ratio of the size of the potential search space (92^n) to the number of generated NJs increases monotonically from 10^7 to 10^{43}. Since, in effect, our approach searches through the space, this suggests that NJs prune the search space efficiently.

Table 1: Cutting-stock problem results

Lots	4	8	12	16	20	24
Total waste (%)	1.87	1.85	1.73	1.69	1.69	1.69
Time (s)	0.02	0.5	1.8	11.6	67.6	607.6
NJs	8	139	395	1,387	4,397	14,772

Figure 2 compares our approach with two others -- integer programming and CHIP, in execution time. These two approaches take roughly the same amount of time. Our approach can optimize more than ten times faster.

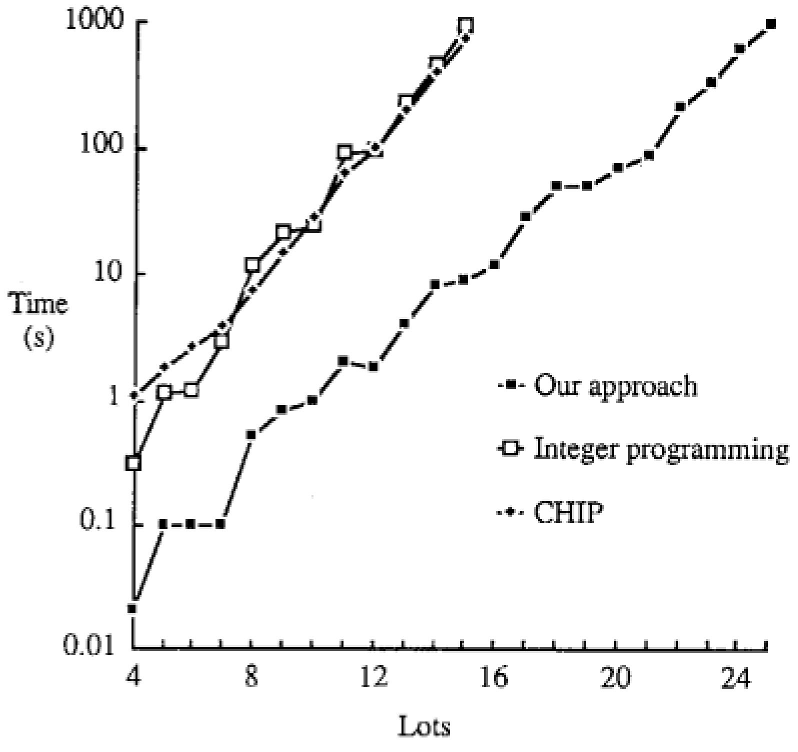

Figure 2: Execution time comparison with other approaches

4. Conclusion

We have proposed an approach to solving discrete constraint satisfaction or optimization problems, including LSI logic design, cutting-stock, and job-shop scheduling problems. Our approach uses sufficient conditions for constraint violation (NJs) to prune the search space. Our results show that (1) NJs enable us to prune the search space efficiently, (2) the approach can optimize over ten times faster than other approaches. We are currently evaluating this approach using job-shop scheduling problems.

Reference

[Dincbas 88] Dincbas, M., Simonis, H. and Hentenryck, P. V.: *Solving a Cutting-Stock Problem in Constraint Logic Programming*, Proc. of the Fifth International Conference and Symposium on Logic Programming, pp. 42-58 (1988).

Representing Situations in Forward Planning with Boolean Arrays

Neng-Fa Zhou
zhou@mse.kyutech.ac.jp
Faculty of Computer Science and Systems Engineering
Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka, Japan

1 Introduction

Among the methods for planning [2, 5, 7, 9, 10, 11, 12, 13], forward planning is the most straightforward one. It reasons forward from the initial situation, deriving new situations from the old ones until a situation in which the goal is satisfied is generated. Forward planning is also called progressive planning [3, 7] and bottom-up planning [5]. Compared with backward planning [7, 12], it is much more straightforward because the partial plan already constructed is accessible for constraint checking and no interaction between subgoals need to be considered. Compared with non-linear planning [9, 10, 11], it is simple to check whether a condition is satisfied at a situation [1].

There are two well used schemes, namely *implicit scheme* and *explicit scheme*, for representing situations in forward planning. The implicit scheme uses the partial plan already constructed and the initial situation to represent the current situation. It is easy for this scheme to apply a new operator or undo an old operator. It simply inserts the operator to or delete the operator from the end of the partial plan. However, it is difficult to search for an applicable operator at the current situation. In order to test whether or not a condition is satisfied at the current situation, it has to examine the partial plan and sometimes the initial situation. This is done by the *regression* operation [7, 8] or frame axioms [4, 5, 6].

Kowalski has pointed out the problem of using frame axioms [5]. He says that "It can be argued that this (using frame axioms) is unnatural and potentially inefficient. The alternative, when using depth-first search and reasoning forward from the initial state, is to store the current state explicitly." He suggests the explicit scheme that stores the current situation explicitly as a data base. It is easy for this scheme to test whether a condition is satisfied, because we have only to retrieve the current data base. However, this scheme does has its drawbacks. We have to update the data base when we apply a new operator. In addition, we have to restore the data base when backtracking occurs.

We propose a new scheme for representing situations in forward planning which is more efficient than both the schemes described above. The main idea is to store the current situation with several Boolean arrays. A Boolean array is a relation whose elements are associated with states. For each type of conditions, we define a Boolean array whose elements correspond to the conditions. In order to check whether a condition in an operator is true, we check whether its corresponding element is true. After applying an operator, we simply set the corresponding elements of the facts in the add-list of the operator to be true and set the corresponding elements of the facts in the delete-list of the operator to be false. When backtracking occurs, we simply restore the Boolean arrays to the state at a previous situation.

2 Method

A Boolean array is a relation whose elements are associated with states. The predicate

```
boolean_array(A,S)
```

defines several Boolean arrays where A is an array element and S is the state associated to A. Each array element takes the form of $f(E_1, \ldots, E_n)$ where f denotes the name of the array and E_i's are atomic terms. The state of an array element is either *true* or *false*.

The call select(f($E_1, \ldots, E_{n-1}, X_n$)), where all E_i's are atomic terms and X_n is a variable, selects a *true* element $(E_1, \ldots, E_{n-1}, E_n)$ from the array named f and binds X_n with E_n. The call set_true(f($E_1, \ldots, E_n$)) (set_false(f($E_1, \ldots, E_n$))) changes the state of the element $(E_1, \ldots, E_n)$ in the array named f into *true* (*false*). The call true(f($E_1, \ldots, E_n$)) (false(f($E_1, \ldots, E_n$))) succeeds if the element $(E_1, \ldots, E_n)$ in the array is *true* (*false*).

We consider how to transform the STRIPS [2] representation of a problem to a program in an extended Prolog with the Boolean array type [14]. For each type of conditions, we define a Boolean array whose elements corresponds to the conditions. A formula in the preconditon of an operator is transformed to a select or a true or a false call. The select call selects a true element corresponding to the formula, and the true and false calls test whether the corresponding element of the formula is true or false. A formula in the add-list is transformed to a set_true call. It sets the corresponding fact to be true. A formula in the delete-list is transformed to a set_false call. It sets the corresponding fact to be false.

For example, for the blocks world problems described in [8], we define the following four Boolean arrays for the four types of conditions : clear(X), ontable(X), holding(X), and on(X,Y).

```
boolean_array(clear(X),S):-
    block(X),
    (initial_state(clear(X))→S=true;S=false).
boolean_array(ontable(X),S):-
    block(X),
    (initial_state(ontable(X))→S=true;S=false).
boolean_array(holding(X),false):-block(X).
boolean_array(on(X,Y),S):-
    block(X),block(Y),
    (initial_state(on(X,Y))→S=true;S=false).
```

These clauses are self-explanatory. For example, the first clause defines a one-dimensional Boolean array named clear which consists of all blocks as its elements. If the fact clear(X) holds in the initial state, then the state of X is *true*; otherwise, the state is *false*.

Each operator in STRIPS representation is transformed into a predicate. For example, the try_pickup predicate defined below searches for an instance of the pickup(X) operator that are applicable at the current situation and applies it.

```
try_pickup(pickup(X)):-
    select(clear(X)),       % precondition
    true(ontable(X)),
    set_true(holding(X)), % add-list
    set_false(clear(X)),    % delete-list
    set_false(ontable(X)).
```

3 Discussion

Our scheme of representing the current situation in forward planning with Boolean arrays is much more natural and efficient than previous ones. Compared with the implicit representation that uses frame axioms, our scheme can test efficiently whether a condition is satisfied. It simply selects a corresponding array element of the condition or tests whether the state of the corresponding array element is true. Compared with the scheme of storing the current situation as a data base, our scheme can add a new fact or delete an old fact efficiently. It simply sets the state of the corresponding array element to be true or false.

Our scheme requires space for storing Boolean arrays and the states of array elements. However, this will not be a serious problem. For a world in which there are l objects and m kinds of facts, assume that each fact has averagely n arguments, then the total space required is $O(m \times l^n)$.

References

[1] Chapman, D. : Planning for Conjunctive Goals, Artif. Intell., Vol.32, pp.333-377, 1987.

[2] Fikes,R.E. and Nilsson, N.J. : STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving, Artif. Intell., Vol.2, pp.189-208, 1971.

[3] Georegeff, M.P.: Planning, Ann. Rev. Comput. Sci., Vol.2, pp.359-400, 1987. Also appear in Readings in Planning, Allen, J. Hendler, J., and Tate, A. (eds.), Mogan Kaufmann, 1990.

[4] Green, C.C. : Application of Theorem-Proving to Problem Solving, Proc. of IJ-CAI'69, pp.219-240, 1969.

[5] Kowalski, R. : Logic for Problem Solving, North Holland, 1979.

[6] McCarthy, J. and Hayes, P.J. : Some Philosophical Problems from the Standpoint of Artificial Intelligence, Machine Intelligence 4, pp.463-502, 1969.

[7] McDermott, D. : Regression Planning, Int. J. of Intelligent Systems, Vol.6, pp.357-416, 1991.

[8] Nilsson, N.J. : Principles of Artificial Intelligence, Tioga Publishing Co., 1980.

[9] Sacerdoti, E.D. : A Structure for Plans and Behaviour, Elsevier-North Holland, 1977.

[10] Tate, A. : Generating Project Networks, IJCAI-77, pp.888-893, 1977.

[11] Vere, S. : Planning, Encyclopedia of Artificial Intelligence, John Wiley & Sons, pp.748-758, 1986.

[12] Warren, D.H.D. : WARPLAN - a System for Generating Plans, DAI, Memo 76, Univ. of Edinburg, 1974.

[13] Wilkins, D.E. : Practical Planning - Extending the Classical AI Planning Paradigm, Morgan-Faufmann, 1988.

[14] Zhou, N.F. : An Extended Prolog with Boolean Array Type, submitted to 1992 Joint International Conference and Symposium on Logic Programming, 1992.