unbound domain variables, and at least one of them is an unbound domain variable. The unbound domain variables are called the *constrainable variables*.

Definition. *MDER* will reduce a goal A for a domain predicate defined in a program *P* for an logic programming system using a computation model ℂ to a set of goals *S* where:

1. A is multiple domain checkable and x_1, , x_n are the constrainable variables with domains d_1, , d_n .
2. $e_j = \{a_j \in d_j \mid \exists\ a_i \in d_i,\ 1 \le i \le n,\ i \ne j\ [P \models_{\mathbb{C}} A\{x_1/a_1, , x_n\ /a_n\}]\} \ne \emptyset$.
3. z_j is c if $e_j = \{c\}$. z_j is a new domain variable with domain e_j otherwise.
4. *S* is $\{x_1{=}z_1,\ ,x\ \ _n{=}z_n\}$ if at most one z_i is a domain variable (trivially satisfied if $n{=}1$). *S* is $\{A\theta, x_1{=}z_1,$ $, x_n\ \ {=}z_n\}$ otherwise, where $\theta = \{x_1/z_1, , x_n\ \ /z_n\}$.

MDER is effectively the Looking-Ahead Inference Rule [25] except that it recognises the concurrent environment.

3.3 Ask domain constraint checking

Definition. *ADER* will reduce a call A for a domain predicate defined in a program *P* for a logic programming system using a computation model ℂ to *true* or *false* subject to the following conditions:

1. A is multiple domain checkable and x_1, , x_n are the constrainable variables with domains d_1, , d_n .
2. A is reduced to *true* if and only if $\forall 1 \le i \le n\ \forall a_i \in d_i,$ $[P \models_{\mathbb{C}} A\{x_1/a_1, , x_n\ \ /a_n\}]$.
3. A is reduced to *false* if and only if $\forall 1 \le i \le n\ \forall a_i \in$ $d_i, [P \models_{\mathbb{C}} \neg A\{x_1/a_1, , x_n\ \ /a_n\}]$.
4. A is unchanged otherwise.

This rule is essential for concurrent logic programming to evaluate the finite domain constraints appearing in the guards of the clauses.

3.4 Constraint declarations in FD-Parlog

In FD-Parlog the programmer may define any predicate as being subject to one of the special evaluation rules, *SDER* or *MDER*. If an *SDER* or *MDER* system predicate appears in guard, it will be automatically be evaluated by using the *ADER*.

4 Programming example

Here we show an example of weak-queens problem, which makes use of stream parallelism and lazy evaluation. The weak-queens problem is a variation of the *n*-queens problem. Instead of placing *n* queens on an *n*×*n* board, we shall place one queen on each column of an infinitely long band of finite width *n*. A queen can attack horizontally, vertically and diagonally, but the attack range of a queen is reduced to a distance *k*, *k* < *n*. Hence a queen in this problem are *weaker* than a normal queen. Hence queens can be put on columns *n*+1, *n*+2, , should *m* > *n*.

The following is a naïve (eager) FD-Parlog program to solve the weak-queens problem. This program accepts as input the width *N* of the band and the attack range *K* of a queen, and produces an infinite list *Band* as output. Here ≠ is an *SDER* constraint of disequality.

```
mode weak_queens(?,?,^).
weak_queens(N, K, Band) <-
        K < N:
        gen_list(Band), Band all_in (1..N), constraint(Band, K).
mode constraint(?,?).
constraint([X|T], K) <- safe(X, T, 1, K), constraint(T, K).
mode safe(?,?,?,?).
safe(X, _, N, K) <-
        N > K : true.
safe(X, [Y|T], N, K) <-
        N ≤ K : noattack(X, Y, N), N2 is N + 1, safe(X, T, N2, K).
noattack(X, Y, N) <- Y ≠ X, Y ≠ X + N, X ≠ Y + N.
mode gen_list(^).
gen_list([H|T]) <- gen_list(T).
```

An obvious deficiency of this program is that it generates an infinite list and infinite number of constraints even if only the solution to the first *M* columns is needed. This problem can be solved by the technique of lazy-evaluation [9]. The following program shows such an approach. The predicate *lazy_weak_queens/3* is a lazy equivalence of *weak_queens/3*, which produces the solution to the first *M* columns one by one, on request from the consumer *consumes/2*.

```
mode consumes(?,^).
consumes(0, []).
consumes(M, [H|T]) <- M > 0 : M2 is M — 1, consumes(M2, T).
mode lazy_weak_queens(?,?,?).
lazy_weak_queens(N, K, Band) <-
        K < N :
        is_list(Band), Band all_in (1..N), constraint(Band, K).
mode constraint(?,?).
constraint([], K).
constraint([X|T], K) <- safe(X, T, 1, K), constraint(T, K).
mode safe(?,?,?,?).
safe(X, [], _, _).
safe(X, _, N, K) <-
        N > K : true.
safe(X, [Y|T], N, K) <-
        N ≤ K : noattack(X, Y, N), N2 is N + 1, safe(X, T, N2, K).
noattack(X, Y, N) <- Y ≠ X, Y ≠ X + N, X ≠ Y + N.
mode is_list(?).
is_list([]).
is_list([H|T]) <- is_list(T).
```

If we are only interested in the solution to, say, the first 6 columns of a band of width 5, and the attack range of a queen is 2, then the query is

```
?- lazy_weak_queens(5, 2, Band), consumes(6, Band).
```

Let *Band* be [*X1*, *X2*, *X3*, *X4*, *X5*, *X6*], then the following constraints are generated before the first (and only) deadlock.

```
X1 ≠ X2          X1 ≠ X2 + 1          X2 ≠ X1 + 1
X1 ≠ X3          X1 ≠ X3 + 2          X3 ≠ X1 + 2
X2 ≠ X3          X2 ≠ X3 + 1          X3 ≠ X2 + 1
X2 ≠ X4          X2 ≠ X4 + 2          X4 ≠ X2 + 2
X3 ≠ X4          X3 ≠ X4 + 1          X4 ≠ X3 + 1
X3 ≠ X5          X3 ≠ X5 + 2          X5 ≠ X3 + 2
X4 ≠ X5          X4 ≠ X5 + 1          X5 ≠ X4 + 1
X4 ≠ X6          X4 ≠ X6 + 2          X6 ≠ X4 + 2
X5 ≠ X6          X5 ≠ X6 + 1          X6 ≠ X5 + 1
```

Solutions will then be found by backtracking.

In real-life applications, a tricky consumer could generate more interesting scenarios. For example, the following tricky consumer can cause the first 4 rows of constraints be generated and X2 be solved before the last 5 rows are generated.

```
mode tricky_consumes(?,?,^).
tricky_consumes(0, _, []).
tricky_consumes(4, X, [H|T]) <- data(X) : consumes(3, H, T).
tricky_consumes(M, _, [H|T]) <-
      M > 0, M ≠ 4 : M2 is M — 1, consumes(M2, H, T).
?- lazy_weak_queens(5,2,Band), tricky_consumes(6,_,Band).
```

5 Discussions and comparisons with related work

An early attempt to parallelise finite domain constraint solving with was reported in [26], which enhances CHIP by the OR-parallelism in PEPSys [3]. It does not support search-parallelism and stream AND-parallelism [8] found in FD-Parlog.

FD-Parlog provides real direct support to *d*-variables, *d*-unification and domain evaluation rules. The *SDER* and *MDER* are powerful rules as they make use of logical deduction to eliminate invalid values. For example, as seen from the definition of *M D E R*, if $P^{\circ\circ}|\neq Å_m\{x_1/a_1, ,x_n\ /a_n\}$ for some a_1 in d_1 for any combination of a_i in d_i, $i \neq 1$, then a_1 will be eliminated from d_1. This support distinguishes FD-Parlog from the work of Bahgat [2], where domain constraints are only expressed and solved by explicit use of high-level programming techniques. It also distinguishes FD-Parlog from the work by Gregory *et al.* [13], where domains are represented by a new data-type, making even *d*-unification, a very basic operation, need to be programmed by users.

It is easily observed that FD-Parlog programs are syntactically similar to their CHIP counterparts, despite of their differences in operational semantics. The readily observable syntactic difference is the omission of labelling predicates in the FD-Parlog programs. This makes FD-Parlog programs more declaratively readable. However, what is more important is the reason behind the omission of the labelling predicates. The Firebird computation model in fact provides the same functionality as *generalised forward checking* [25]. In CHIP, generalised forward checking is achieved by ordering the labelling predicate after the constraints, and thus causing different values of the domain variables to be tried in each OR-branch after all constraints have been generated. Explicit programming effort must be put to achieve this desired effect. In FD-Parlog, domain variables will not be labelled until all constraints have been generated, hence the technique of generalised forward checking becomes an intrinsic feature of the language. The *first-fail heuristics*, for instance, which labels the variable with the smallest domain first, can be easily realised in FD-Parlog by a smart deadlock handler.

In fact, it can easily be shown that, to solve the same problem, the number of backtracking in FD-Parlog due to deadlock/partitioning will not be more than that in CHIP due to labelling (or, more precisely, due to the nondeterminism of *indomain/1* in CHIP), as any heuristics applicable in labelling applies equally well in the design of the deadlock handler. In FD-Parlog we still have a further flexible alternative of executing the partitions in parallel [24].

FD-Parlog is a concurrent language. Therefore, whereas the order of goals is important in sequential languages such as CHIP, it is irrelevant in FD-Parlog. Moreover, FD-Parlog supports stream AND-parallelism. The availability of stream AND-parallelism provides a higher degree of flexibility for programming, as shown in the *weak-queens* example. Achieving the effect of corouting in sequential languages such as CHIP requires much programming effort and a number of control structures, such as the *if then else* , *delay* and *demon* mechanisms in CHIP. These could all be taken as granted in FD-Parlog.

cc(FD) [27] is a nondeterministic constraint logic programming language over finite domains and intervals in the cc framework [22]. Basically, FD-Parlog and cc(FD) are two orthogonal approaches to finite domain constraint handling in concurrent logic programming languages. cc(FD) provides a number of interesting higher-level constraint combinators for constraint handling in the cc framework, while FD-Parlog aims at coherent incorporation of finite domain constraints into the concurrent logic programming paradigm.

Both languages provide the basic constraint processing facilities and nondeterminism. cc(FD) supports don t know nondeterminism by adopting the Andorra Model [28], but FD-Parlog supports nondeterminism by allowing choice points to be set up using domain variables. These two kinds of nondeterminism is completely different in nature.

The cc(FD) language supports both finite domain constraints and interval constraints, while only the former is supported in FD-Parlog. However, as pointed out by Lee [16], there is no significant difference between these two kinds of constraints.

GDCC [23] is a new concurrent constraint logic programming language based on KL1 [5]. The constraints could be in the algebraic, Boolean or linear integer domains, but finite domain constraints are not supported. Therefore it is more related to the work of [18] than the work reported in this paper.

6 Conclusions and future work

In this paper we presented the operational semantics of the Firebird computation model for concurrent finite domain constraint logic programming. One of the unique features of the Firebird computation model is that choice points are set up using unbound domain variables. Like the computation models for most concurrent logic programming languages, the Firebird Computation Model supports stream AND-parallelism and search-parallelism. As the computation model provides the facilities of a kind of implicit OR-parallel labelling, the clauses in a program retain their logical meaning. The extra-logical labelling operation needs not be explicitly specified within the clause bodies.

A prototype implementation of this model in Parlog, the FD-Parlog language, has been built by modification to the Parallel Parlog system. We are implementing a Firebird language, which is a refinement of FD-Parlog, on a DECmpp Sx-100 massively parallel SIMD computer with a 8,192 processor element back-end data-parallel unit [24]. This machine is equivalent to a MasPar MP-1 [4]. OR-parallelism is exploited in the parallel execution of partitions. We are also building an implementation of the Firebird language on UNIX workstations.

References

[1] Bahgat, R. & Gregory, S. 1989 Pandora: Non-deterministic Parallel Logic Programming. In *Logic Programming: proceedings of the Sixth International Conference*, Lisbon, (ed. G. Levi & M. Martelli), 471—486. Cambridge: The MIT Press.

[2] Bahgat, R. 1992 *Non-Deterministic Concurrent Logic Programming in Pandora*. Singapore: World Scientific Publishing.

[3] Baron, U., de Kergommeaux, J. C. & Hailperin, M. 1988 The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988*, Tokyo, 841—850.

[4] Blank, T. 1990 The MasPar MP-1 Architecture. *Proc. of the IEEE COMPCON Spring 1990*, San Francisco.

[5] Chikayama, T., Sato, H. & Miyazaki, T. 1988 Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988*, Tokyo, 230—251.

[6] Clark, K. L. & Gregory, S. 1985 Notes on the implementation of PARLOG. *J. of Logic Programming*, **2**, No. 1, 17—42.

[7] Clark, K. L. 1990 Parallel Logic Programming. The Computer J., **33**, No. 6, 482—493.

[8] Conery, J. S. 1987 *Parallel Execution of Logic Programs*. Boston: Kluwer Academic Publishers.

[9] Conlon, T. 1989 Programming in PARLOG. Addison-Wesley Publishers Ltd.

[10] Crammond, J., Davison, A., Burt, A., Huntbach, M., Lam, M., Cosmadopoulos, Y. & Chu, D. 1993 *The Parallel Parlog User Manual V1.5*. Parlog Group, Department of Computing, Imperial College of Science, Technology and Medicine. London.

[11] Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T. & Berthier, F. 1988 The Constraint Logic Programming language CHIP. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988*, 693—702.

[12] Gregory, S. 1987 *Parallel Logic Programming in PARLOG: The Language and its Implementation*. Wokingham: Addison-Wesley.

[13] Gregory, S. & Yang, R. 1992 Parallel Constraint Solving in Andorra-I. In *Procc of the Int. Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 843—850.

[14] Jaffar, J. & Lassez, J.-L. 1987 Constraint Logic Programming. In *Conf. record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, 111—119.

[15] Jaffar, J., Michaylov, S., Stuckey, P. J. & Yap, R. H. C. 1992 The CLP($\mathfrak{R}$) Language and System. ACM Trans. *on Programming Lang. and Syst.*, **14**, No. 3, 339—395.

[16] Lee, J. H. M. 1992 *Numerical Computation as Deduction in Constraint Logic Programming*. Ph.D. Dissertation, Technical Report DCS-198-IR, Department of Computer Science, University of Victoria. Victoria.

[17] Leung, H.-F. 1993 *Distributed Constraint Logic Programming*. Singapore: World Scientific Publishing.

[18] Leung, H.-F. & Clark, K. L. 1992 Constraint Solving in Distributed Logic Programming Languages. Presented at *2nd Int. Symp. on Artificial Intelligence and Mathematics*, Fort Lauderdale.

[19] Leung, H.-F., Tong, B.-M. & Clark, K. L. 1993 The Firebird Computation Model for Finite Domain Constraint Solving in Concurrent Logic Programming and its Realisation in FD-Parlog. In preparation.

[20] Maher, M. J. 1987 Logic Semantics for a Class of Committed-Choice Programs. In *Logic Programming: proceedings of the Fourth International Conference*, Melbourne, (ed. J.-L. Lassez), 858—876. Cambridge: The MIT Press.

[21] Santos Costa, V., Warren, D. H. D. & Yang, R. 1991 Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Proceedings of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*.

[22] Saraswat, V. A. 1989 *Concurrent Constraint Programming Languages*. Ph.D. Thesis, Carnegie-Mellon University.

[23] Terasaki, S., Hawley, D. J., Sawada, H., Satoh, K, Menju, S., Kawagishi, T., Iwayama, N. & Aiba, A. 1992 Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 330—346.

[24] Tong, B.-M. & Leung, H.-F. 1993 Constraint Logic Programming on Massively Parallel SIMD Computers. In *Logic Programming: proceedings of the 1993 International Symposium*, Vancouver, (ed. D. Miller). Cambridge: The MIT Press.

[25] Van Hentenryck, P. 1989 *Constraint Satisfaction in Logic Programming*. Cambridge: The MIT Press.

[26] Van Hentenryck, P. 1989 Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Logic Programming: proceedings of the Sixth International Conference*, Lisbon, (ed. G. Levi & M. Martelli), 165—180. Cambridge: The MIT Press.

[27] Van Hentenryck, P., Saraswat, V. & Deville, Y. 1993 Design, Implementations, and Evaluation of the Constraint Language cc(FD). Technical Report No. CS-93-02, Department of Computer Science, Brown University. Providence.

[28] Warren, D. H. D. 1988 The Andorra Model. Presented at Gigalips Project Workshop, University of Manchester. Manchester.