INCREMENTAL QUERYING IN THE CONCURRENT CLP LANGUAGE IFD-CONSTRAINT PANDORA

J.H.M. Lee and H.F. Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: {jlee,lhf}@cs.cuhk.edu.hk

Keywords: Incremental Execution, Concurrent Constraint Logic Programming

ABSTRACT

Lee and Leung propose the IQ-scheme [11], a framework for efficient incremental query processing in concurrent constraint logic programming. They show that the performance of the IQ-scheme is in general better than the scheme proposed by Van Hentenryck and Le Provost [22]. IFD-Constraint Pandora is an instance of the Constraint Pandora class of languages that supports finite domain and interval constraints. In this paper we show how the IQ-scheme can be incorporated in IFD-Constraint Pandora. We first describe the IFD-Constraint Pandora language, its operational semantics and constraint satisfaction mechanism. Then we present an implementation scheme for the IQ-scheme in the IFD-Constraint Pandora language, which is based on the concept of *constraint effective point*. Finally, we present some preliminary results of applying the IQ-scheme to an IFD-Constraint Pandora program that solves the map-coloring problem.

1 INTRODUCTION

According to the *incremental query processing* [19] or *incremental execution* [3, 4, 22] framework in logic programming, users enter queries by *increments*. After an increment is entered, Prolog displays an answer, which is the result of, not the increment but, the query accumulated so far. In this framework, variables with the same identifiers in different increments denote the same variable. Thus increments are continuations of the same query. The aim is to minimize re-execution by making use of information gathered in the execution of the previous query increments.

Van Hentenryck and Le Provost [22] propose a scheme (VHLP-scheme hereafter) for incremental query processing in the CLP framework [10]. In the VHLP-scheme, the increments are all constraints instead of goals: user is allowed to add new constraints to the query or delete old constraints from it, but the goals are not changed. Van Hentenryck and Le Provost argue that the VHLP-scheme, which is based on re-execution, avoids the thrashing behavior of backtracking due to bad backtrack points and passive use of constraints.

Lee and Leung [11] propose the backtracking-based IQ-scheme for incremental query processing in concurrent constraint logic programming. They show that the IQ-scheme is in general better than the VHLP-scheme in *concurrent* constraint logic programming as it avoids much re-execution that is needed in the VHLP-scheme. As we show in section 5 of this paper, the execution of a typical query in concurrent constraint logic program sets up far less number of choice points than its Prolog counterpart. Normally, hundreds of derivation steps are performed between two consecutive choice points. While the IQ-scheme is able to make use of a large portion of execution context existing in the system, the VHLP-scheme *re-builds* them when executing the new query.

In this paper, we present the IFD-Constraint Pandora language, an instance of the Constraint Pandora [13] class of languages. We then describe a scheme to implement the IQ-scheme in the language. IFD-Constraint Pandora is a concurrent constraint logic programming language that supports finite domain and interval constraints. Its constraint assimilation mechanism is based on an adaptation of the arc-consistency algorithm AC-3 [15]. We note that, with an abstract-machine implementation [1, 13], it is nontrivial to incorporate the IQ-scheme into the language implementation. The main reason is that, in the deletion of an old constraint, the IQ-scheme selects the best restart point to which the execution should be unwound, and then the constraint to be deleted is removed from the constraint store before execution is restarted from there. In an abstract-machine implementation, however, the constraint might have affected the constraint store before the execution reaches the selected restart point, and it is costly to undo its effect. In this paper we propose an efficient implementation method for the IQ-scheme, which *approximates* the ideal IQ-scheme by identifying the *effective point* of the constraint to be deleted. If the effective point is at or below the selected restart point, then we just follow the IQ-scheme. Otherwise execution is unwound to the first choice point above the constraint effective point.

This paper is structured as follows. Section 2 reviews the Constraint Pandora class of language. The IFD-Constraint Pandora language is presented in section 3. Section 4 reviews the IQ-scheme. In section 5, we present the implementation scheme with some preliminary results. Section 6 concludes the paper.

2 THE CONSTRAINT PANDORA SCHEME

2.1 The Language

Constraint Pandora [13] defines a family of non-deterministic concurrent CLP languages. A Constraint Pandora program syntactically resembles a Pandora [1] program, except that constraints are allowed in clause guards as well as clause bodies. Using Maher's [16] terminology, the constraints in clause guards are *ask*-constraints and those in clause bodies are *tell*-constraints.

A clause in Constraint Pandora is of the form

$$H \leftarrow C_{ask} : B, C_{tell}$$

where H is an atom called the *clause head*, C_{ask} a (possibly empty) conjunction of ask-constraints, B a (possibly empty) conjunction of atoms called the *body goals*, and C_{tell} a (possibly empty) conjunction of tell-constraints. C_{ask} are collectively called the *guard* and B and C_{tell} are collectively called the *body*.

2.2 Operational Semantics

Constraint Pandora belongs to the family of concurrent logic programming languages [5] and is related to the ALPS framework [16]. As in Pandora, "goal" is synonymous with "process" in the constraint version. The execution of Constraint Pandora programs can be abstracted to a sequence of *derivation steps*. A derivation step is a transition from an *execution state* to another. An execution state of Constraint Pandora consists of a *process pool* P, which is a multiset, and a *constraint store* S, which is a set.

A Constraint Pandora query is of the form

$$\leftarrow B, C_{tell}$$

where B is a conjunction of body goals, and C_{tell} is a conjunction of tell-constraints. Initially, the process pool and the constraint store contain all the goals and all the constraints in the query respectively.

There are two kinds of derivation steps. When a process executes, it "commits" to a candidate clause (for don't-know processes, *the* candidate clause), generates the tell-constraints in the clause body, spawns the processes in the clause body, and then terminates. The newly generated constraints and processes are added to the constraint store and process pool respectively. The old process is removed from the process pool.

Another kind of derivation steps occurs when every process in the system suspends. The execution state is then said to *deadlock*. If the system deadlocks and there is no don't-know process in the system, a failure occurs. Otherwise, a deadlock handler is invoked to select one of the don't-know processes, based on which a *choice point* is set up. The candidate clauses of the selected don't-know processes are then tried in turn. We assume that the deadlock handler employs a fixed computation rule and clause ordering [14] in the selection of body goals and the trying of rules at each step of the computation. When failure occurs, the execution state will be restored to that of the most recent choice point, and an untried candidate clause will be tried. If all candidate clauses at the choice point have been tried, then a new failure occurs.

The execution of a Constraint Pandora query $Q = \leftarrow B, C_{tell}$ *succeeds* if and only if the constraint store is consistent and the process pool is empty. Let P be the Constraint Pandora program for the query and $\mathcal{D}$ the underlying constraint domain. Suppose the constraints in the constraint store at the end of a successful execution is S'. The soundness of Constraint Pandora [13] guarantees that

$$P, \mathcal{D} \models (\forall)(S' \Rightarrow C_{tell} \wedge B).$$

We call S' a *solution* to the query Q.

2.3 Constraint Pandora Search Tree

A *Constraint Pandora search tree* (CPS-tree) is a tree in which each node is an execution state. The root of the tree is the initial execution state corresponding to the query. An execution state is a child of another if and only if the former is obtained from the latter by a derivation step. Obviously, every node in the tree has at most one child, except that a deadlock state might have more than one child.

There are three possible kinds of leaves. A leaf with an empty process pool and a consistent constraint store corresponds to a successful execution. The solution of this successful execution is the constraint store in the leaf. A leaf with an inconsistent constraint store, or a leaf with only suspended don't-care processes in its process pool corresponds to a failure.

A *skeleton Constraint Pandora search tree* (skeleton CPS-tree) is a "reduced" CPS-tree. Intuitively, a skeleton CPS-tree is obtained from a CPS-tree by concentrating on the root, the deadlock nodes, and the leaves, and ignoring the rest. A skeleton CPS-tree T' of a CPS-tree T can be defined as follows:

- The root of T' is the same as that of T.
- If N is the root or a deadlock node in T', and $(N, N_1, \ldots, N_k, M)$ is a path in T, and $N_1, \ldots, N_k$ ($k \geq 0$) are not deadlock nodes, and M is a deadlock node or a leaf in T, then M is a child of N in T'.
- There are not any other nodes in T'.

3 IFD-CONSTRAINT PANDORA

Lee and van Emden [12] show that the domain reduction operator used for interval domain constraint is an instance of LAIR of CHIP [20] and that the lookahead-efficient computation rule is a constraint relaxation algorithm similar to AC-3 [15]. Following this approach, we generalize FD-Constraint Pandora [13] to handle also interval domain constraints. We call the extended language *IFD-Constraint Pandora*, which can be regarded as a concurrent version of ICHIP [12]. IFD-Constraint Pandora features the notion of domains and domain variables [20], abbreviated as *d-variables*. A *domain* is either a non-empty finite set of constants or a non-empty interval of real numbers, whose bounds are floating-point numbers. Each d-variable has the form X^D, where D is the associated domain of X^D. A constraint in IFD-Constraint Pandora contains only ground terms and d-variables. Figure 1 contains a IFD-Constraint Pandora program that solves the 4-queens problem. The <=/2 test predicate [7] in the guard is for input matching (or one-way unification). The ::/2 predicate declares d-variables and their associated domains. In this example, we declare the domains of all d-variables as $\{1, 2, 3, 4\}$.

Each derivation step in Constraint Pandora consists of two substeps: (1) generation of new processes and constraints and (2) assimilation of the new constraints into the constraint store. Substep (1) is shared by all instances of Constraint Pandora. Each distinguished instance of Constraint Pandora, however, has a unique substep (2). In the

```
queens(L) :-
        [X1,X2,X3,X4] <= L :
        X1 :: 1..4, X2 :: 1..4, X3 :: 1..4, X4 :: 1..4,
        constrain([X1,X2,X3,X4]), indomain(X1),
        indomain(X2), indomain(X3), indomain(X4).

constrain(L) :- [] <= L : true.
constrain(L) :- [X|T] <= L :
        safe(X, T, 1), constrain(T).

safe(X, L, N) :- [] <= L : true.
safe(X, L, N) :- [Y|T] <= L :
        noattack(X, Y, N), N2 is N + 1, safe(X, T, N2).

noattack(X, Y, N) :-
        X ≠ Y, X ≠ Y + N, X ≠ Y - N.

indomain(X) :- X = 1.
indomain(X) :- X = 2.
indomain(X) :- X = 3.
indomain(X) :- X = 4.
```

Figure 1: The 4-queens problem in IFD-Constraint Pandora.

following, we use $\vec{D}$ to denote a vector $(D_1, \ldots, D_n)$ of n elements. The constraint assimilation step of IFD-Constraint Pandora is shown in algorithm 1, which is executed once in each derivation step after new constraints are generated. Algorithm 1 is an adaptation of the consistency algorithm AC-3 [15]. The core of the algorithm is the *domain restriction operation* `REVISE()`. Given a constraint p, the domain restriction operation removes inconsistent values from the domains of d-variables appearing in p. The `REVISE()` subprogram differs for different kinds of constraints. The subprogram can be as simple as functions that enforce node- and arc-consistency [15], or it can also be as complex as the algorithms behind the *atmost* and *accumulate* constraints in CHIP [21, 23, 6]. For example, with node-consistency:

$$\texttt{REVISE}(even(X^{\{1,2,3,4,5\}}), (\{1,2,3,4,5\})) = (\{2,4\})$$

and with interval arc-consistency:

$$\texttt{REVISE}(X^{[1,5]} + Y^{[3,6]} = 10, ([1,5],[3,6])) = ([4,5],[5,6])$$

, where the notation "$[x, y]$" denotes a close interval between bounds x and y. We leave the implementation of `REVISE()` unspecified and assume appropriate implementation that satisfies *contractance*, *correctness*, *monotonicity*, and *idempotence* defined in the sense of Benhamou and Older [2]. Algorithm 1 is thus an efficient procedure for coordinating the application of `REVISE()` on constraints in a given set of constraints.

Algorithm 1 resembles a classical iterative numerical-approximation technique called "relaxation" [17], which was adopted in SKETCHPAD [18]. In relaxation, we make an initial guess at the values of the unknowns and then estimate the error of the guess. New guesses are then made using the original guess and the error estimate. This process is repeated until the error is sufficiently small or satisfies a certain termination criterion. In algorithm 1, the initial guess is the input domains. Errors are values in the domains that cannot satisfy the relation associated with the constraint. New domains are computed by removing the inconsistent values. These pruned domains are better approximations of the originals in the sense that they provide more accurate inclusions of the answers. Numerical relaxation may fail to converge or terminate even when the constraints have a solution. Algorithm 1 does not suffer from this problem.

A: the *active set* of constraints
P: the *passive set* of constraints

Set A to contain all new constraints
Set P to contain all old constraints in the store
while A is not empty
  Remove p from A with associated domains $\vec{D}$
  $\vec{D}' = \texttt{REVISE}(p, \vec{D})$
  if any domain in $\vec{D}'$ is $\emptyset$ **then**
   Exit with failure
  else
   if $\vec{D} \neq \vec{D}'$ **then**
    foreach d-variable X^{D_i} in p s.t. $D_i \neq D'_i$
     Generate the substitution $\theta = \{X^{D_i}/Y^{D'_i}\}$
     Apply θ to A and P
     foreach constraint q in P
      if q contains X^{D_i} **then**
       Move q from P to A
      endif
     endforeach
    endforeach
   endif
  endif
  Add p to P
endwhile

Algorithm 1: A constraint assimilation algorithm.

Theorem 3.1: Algorithm 1 always terminates.

4 INCREMENTAL QUERYING AND IQ

4.1 Incremental Queries and Preliminaries

In this subsection, we review the main results of the IQ-scheme and relevant terminology. We define an *incremental query sequence* to be a sequence of queries of the form

$$(\leftarrow B, C_1), (\leftarrow B, C_2), \ldots, (\leftarrow B, C_i), (\leftarrow B, C_{i+1}), \ldots$$

where, without loss of generality, C_{i+1} is obtained from C_i by either adding or deleting *one* constraint. Given a program P and an incremental query sequence $Q_1, Q_2, \ldots, Q_n$, by *incremental queries*, we mean finding the first solutions $\theta_1, \theta_2, \ldots, \theta_n$ to $Q_1, Q_2, \ldots, Q_n$ respectively in such a way that computation results from $Q_1, Q_2, \ldots, Q_{i-1}$ are *reused* as much as possible to obtain the solution to each Q_i $(2 \leq i \leq n)$.

We number each clause in a program. Let T be a (skeleton) CPS-tree. The nodes of T can then be *labeled* as follows:

- The root node is labeled with the empty string ϵ.
- If T has node N with label L and N has $m \geq 1$ children associated with clauses $i_1, \ldots, i_m$, we assign $L.i_1$, ..., $L.i_m$ as labels to the children respectively, where "." is the string concatenation operator.

A string L_1 is *lexicographically smaller than or equal to* string L_2, denoted by $L_1 \preceq L_2$, if and only if one of the following three conditions holds:

- L_1 is ϵ.

- $L_1 = n.L_1'$, $L_2 = m.L_2'$, and $n < m$.
- $L_1 = n.L_1'$, $L_2 = n.L_2'$, and $L_1' \preceq L_2'$.

Let P be a program, Q a query, and T the (skeleton) CPS-tree of Q with respect to P. The *first* success leaf in T for Q with respect to P has label L such that for any success leaf with label L', $L \preceq L'$. Since the label of a node in a (skeleton) CPS-tree denotes the computation path from the root to that node, we abuse notation by identifying a node with its label.

The following property is useful in the design of the IQ-scheme.

Property 4.1: Let L be the first success leaf of query $Q = \leftarrow B, C_1$. If $\mathcal{D} \models (\forall)(C_2 \Rightarrow C_1)$, then there is no solution of $Q' = \leftarrow B, C_2$ with label $L' \preceq L$. ∎

4.2 The IQ-Scheme

The IQ-scheme is based on backtracking. In a backtracking-based language, execution context is saved on the trail along an execution path. Execution can be unwound to a particular point upon backtracking. In the following, the discussion on the IQ-scheme is based on the skeleton CPS-tree since backtrack points appear only in deadlock nodes.

4.2.1 Addition of Constraints

Assume the new goal $Q_{i+1} = \leftarrow B, C_{i+1}$ is obtained from Q_i by adding one constraint, i.e., $C_{i+1} = C_i \cup \{c\}$. Suppose θ_i is the first solution of Q_i. Further suppose the length of the execution path of Q_i is l and the content of the constraint store after the k-th ($0 \leq k \leq l$) derivation step is S_k. Starting from S_l, we unwind the trail to the first S_k such that $S_k \cup \{c\}$ is consistent. If $k < l$, standard execution resumes from the most recent backtrack point of the $(k+1)$st derivation step. If $k = l$, then nothing needs to be done.

4.2.2 Deletion of Constraints

Assume $C_{i+1} = C_i \setminus \{c\}$. The first success of query Q_i corresponds to the traversal from the root of the (skeleton) CPS-tree to a particular leaf. We define the label of this leaf to be the *oracle* of query Q_i, denoted $O(C_i)$. We compute an oracle O prior to $O(C_i)$ that has the best pruning, i.e.

$$O = \max_{j=1}^{i-1}\{O(C_j) \mid C_j \subseteq C_{i+1}\},$$

where "max" uses the lexicographical ordering on strings. Property 4.1 guarantees that there is no solution to the new goal with a computation path smaller than O. Suppose $O(C_i) = L_1.p.L_2$ and $L_1.p$ is the common prefix of O and $O(C_i)$. The IQ-scheme unwinds along $O(C_i)$ up to the execution context of p. Suppose that the constraint store at that point is $S = C \cup C_i$. We replace C_i by C_{i+1} in S and then proceed with oracle O in the same way as for additions.

5 IQ IN IFD-CONSTRAINT PANDORA

5.1 Effective Points for Constraints

"All constraints are equal, but some constraints are more equal than others."

In a constraint logic programming system, constraints are generated on the fly during execution. Once generated, the new constraints are added to the constraint store and assimilated. In IFD-Constraint Pandora, a consistency algorithm [15] is used for constraint assimilation, which works by actively applying the domain restriction operation on constraints in the constraint store to prune the domains of d-variables. Declaratively, all constraints in the constraint store share the same status: the constraints, as a conjunction, serve to define implicitly the solution space of the constraint satisfaction problem at hand.

Operationally, however, we differentiate between two types of constraints. A constraint *takes effect* if, during execution of algorithm 1, `REVISE()` has used it to prune domains of its associated d-variables, resulting in a substitution which might be propagated to the other constraints in the constraint store. These activities occur in lines 10–18 of algorithm 1. This notion of *effectiveness* of a constraint is important in the deletion of constraint. Suppose the constraint store denotes a conjunction of constraints C. We can compute $C \setminus \{c\}$ by simply removing c from the constraint store if c has not taken effect in the store. Removing a constraint that has taken effect, however, involves trailing of bindings and undoing substitutions.

To illustrate the concept of effectiveness, we use the 4-queens program in figure 1, which works by generating the necessary disequality constraints $\neq/2$ to prohibit the queens to be on the same row, positive diagonals, and negative diagonals. The *indomain*/1 predicate is then used to enumerate elements in the domains of d-variables and instantiate d-variables to find solutions, by augmenting tree searching with constraint propagation. By virtue of the test-and-generate paradigm in constraint programming, the *indomain*/1 predicate is defined to execute only after all other predicates either suspend or terminate. Suppose the initial query is "?- *queens([X1,X2,X3,X4])*." Let us examine the constraint store right before the first *indomain*/1 process starts execution. There are a total of 18 constraints in the store, listed as follows:

$X1 \neq X2$		
$X1 \neq X2 + 1$		
$X1 \neq X2 - 1$		
$X1 \neq X3$	$X2 \neq X3$	
$X1 \neq X3 + 2$	$X2 \neq X3 + 1$	
$X1 \neq X3 - 2$	$X2 \neq X3 - 1$	
$X1 \neq X4$	$X2 \neq X4$	$X3 \neq X4$
$X1 \neq X4 + 3$	$X2 \neq X4 + 2$	$X3 \neq X4 + 1$
$X1 \neq X4 - 3$	$X2 \neq X4 - 2$	$X3 \neq X4 - 1$

None of these constraints can eliminate values from the domains of the associated d-variables. Thus, no constraints have taken effect yet. Each d-variable has associated domain $\{1, 2, 3, 4\}$. Suppose *indomain*$(X1)$ is now executed and causes $X1$ to be bound to 1. This instantiation triggers algorithm 1 to initiate a sequence of constraint propagation. For example, the value 1 is eliminated from the domain of $X2$ by `REVISE()` using the constraint "$X1 \neq X2$." This is the first time that the constraint "$X1 \neq X2$" prunes the domains of its d-variables. Thus "$X1 \neq X2$" takes effect and this effect is propagated to other constraints involving variable $X2$.

We are now ready to define the notion of effectiveness formally. Consider the execution of an IFD-Constraint Pandora program. Let N be a non-leaf node in a CPS-tree and N' be one of its children. If, during the execution of algorithm 1 in the derivation step from N to N', the domain of a d-variable in the constraint store is pruned as a result of application of `REVISE()` using a constraint c in the constraint store of N, then N is said to be the *action point* of the constraint c. If N is an action point of c and all of its ancestors in the CPS-tree are not, then N is said to be the *effective point* of c, and is denoted $\mathcal{E}_c$.

5.2 An Approximation to IQ

We assume an IFD-Constraint Pandora implementation scheme similar to those presented by Bahgat [1] and Leung [13]. In constraint addition, a modified trail-unwinding operation is required. At each unwinding step, we extract the constraint store at the choice points, submit the newly added constraint to the constraint store and exercise algorithm 1 for constraint assimilation. This unwinding process starts from the leaf node corresponding to the current success execution, and is repeated until either the root of the CPS-tree is reached or consistency is detected. Thus we can adhere to the IQ scheme faithfully with addition of constraints.

With deletion of constraints, we compute the common prefix of O and $O(C_i)$ and unwind the execution to the choice point N that corresponds to the end of this common prefix (note that N is a choice point as we are considering a skeleton CPS-tree). Assume that $C_{i+1} = C_i \setminus \{c\}$ and the constraint store S at N is $C \cup C_i$ for some set of constraints C. Since constraints in S are fully assimilated, we cannot simply remove c from S to obtain $C \cup C_{i+1}$ in general. It is safe to do so only if c has not taken effect at N. We present an *approximation to IQ* for constraint deletion:

- if one of the ancestors of N is an effective point $\mathcal{E}_c$ for constraint c, then
  - if $\mathcal{E}_c$ is not the root node of the CPS-tree, we further unwind the trail to the first choice point N' above $\mathcal{E}_c$, remove c from the constraint store,
  - otherwise, unwind to the root node and remove c from the constraint store,
- otherwise, remove c from the constraint store of N.

After the removal of c from the constraint store, we proceed with oracle O in the same way as for additions in the VHLP-scheme. To support the approximate IQ-scheme for deletion of constraints, we only have to tag the constraints when they take effect during execution.

Figure 2 illustrates the approximate IQ-scheme pictorially. Suppose $Q_i = \leftarrow B, C_i$ and $Q_{i+1} = \leftarrow B, C_{i+1}$, where $C_{i+1} = C_i \setminus \{c\}$. The branch ending with segments 1, 2, and 3 corresponds to the oracle $O(C_i)$. Without loss of generality, we assume that the first solution of Q_{i+1} occurs to the "left" of Q_i, thus ending with segments 1, 2, and 4. Therefore, the node connecting branch segments 2, 3, and 4 is the theoretically best re-start point. As explained, the IQ-scheme may not be able to locate the perfect re-start point. Suppose that the IQ re-start point is at the node labeled N_{IQ}. If the effective point $\mathcal{E}_c$ of constraint c occurs in segment 1, then we have to further unwind to the node N'_{IQ} before it is safe to remove c from the constraint store. If $\mathcal{E}_c$ occurs in either segment 2 or 3, then we can remove c from the constraint store at N_{IQ}.

5.3 Preliminary Results

We present here some results of applying the IQ-scheme to the map-coloring problem to illustrate the performance of the scheme. The problem is to find an assignment of colors to the nodes in a map such that the two nodes do not have the same color if they are connected by an edge. The map consists of 11 nodes and there are 4 different colors to be used. The program first declares for each node a d-variable with an initial domain that contains all four colors. These

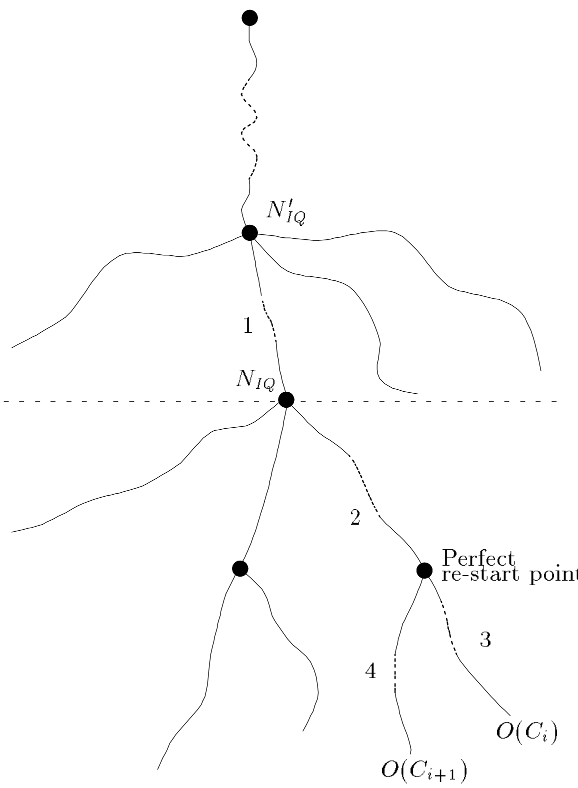

Figure 2: Constraint deletion in the approximate IQ-scheme.

d-variables are named $X1, X2, \ldots, X11$ and the colors 1, 2, 3 and 4 in the following presentation. Then the program sets up disequality constraints for each edge. After the solution to the initial query is found, we repeatedly perform addition and deletion of constraints to test the performance of the IQ-scheme. The results are summarized in table 1. The column labeled n_{VHLP} is the number of derivation steps needed to re-execute from the root of the tree to the restart point N_{IQ} [1] and n_{IQ} is the number of unwinding steps to reach N_{IQ}. It is shown in this table that the performance of the IQ-scheme is better than the VHLP-scheme.

The main reason for the difference between the performance of these two schemes is largely due to the unique characteristic of the Andorra class of concurrent logic programming languages: that is, there are usually a large number of deterministic derivation steps between two consecutive choice points. The overhead of unwinding one step backward is clearly much less than replaying these large number of derivation steps. In our map-coloring program, the execution proceeds for 327 derivation steps before the first choice point is created. This is especially true in concurrent constraint logic programming, in which most of the choice points are created for the "labeling" process, while only few choice points are set up during constraint establishment.

6 CONCLUSIONS

Efficiency of the IQ-scheme in an abstract-machine based implementation of (concurrent) constraint logic programming language [1, 13] depends on efficient implementation of the

[1] In this particular program, $N_{IQ} = N'_{IQ}$.

Query	Increment	Constraint	n_{VHLP}	n_{IQ}
Q_1	-	-	-	-
Q_2	add	$X3 = 2$	328	9
Q_3	add	$X9 = 4$	335	2
Q_4	add	$X10 \neq 2$	336	1
Q_5	delete	$X9 = 4$	335	0
Q_6	add	$X2 \neq 3$	328	7
Q_7	add	$X9 = 4$	335	1
Q_8	delete	$X2 \neq 3$	328	5
Q_9	delete	$X9 = 4$	332	0
Q_{10}	add	$X2 \neq 3$	328	7
Q_{11}	add	$X6 = 3$	331	5
Q_{12}	delete	$X2 \neq 3$	328	5

Table 1: Benchmark results.

constraint addition and deletion operations. Constraint addition amounts to testing the satisfiability (and also computing the solved form) of a previously consistent constraint store augmented with a new set of constraints. Since constraint addition is a frequent step in the normal execution, we can basically adopt the existing constraint addition and assimilation mechanism for constraint addition in the IQ-scheme. Constraint deletion concerns the (re)computation of a solved form of a previously consistent constraint store after a set of constraints is deleted from the store. The difficulty of this operation varies with the underlying constraint domain. Huynh and Marriott [9] and Helm *et al.* [8] develop efficient incremental constraint deletion algorithms for linear equalities and inequalities. In this paper we present the IFD-Constraint Pandora language and an efficient implementation method of the IQ-scheme in the language. For addition of constraints, the implementation scheme faithfully adheres to the IQ-scheme. For deletion of constraints, however, it approximates the ideal IQ-scheme by further unwinding the execution to the first choice point above the effective points of the effective point of the constraints to be deleted. We show that the implementation scheme compares favorably against the VHLP-scheme in the map-coloring problem using a prototype implementation.

ACKNOWLEDGEMENT

We thank the anonymous referees for their comments. This research is partially supported by the RGC Earmarked Grant CUHK 70/93E.

References

[1] Bahgat, R. *Non-Deterministic Concurrent Logic Programming in Pandora.* World Scientific Publishing, 1993.

[2] Benhamou, F., and Older, W. Applying interval arithmetic to real, integer and boolean constraints. *(to appear) Journal of Logic Programming* (1996).

[3] Chatalic, P. Incremental techniques and Prolog. Technical Report TR-LP-23, European Computer-Industry Research Centre, June 1987.

[4] Chatalic, P. IMPRO: An environment for incremental execution in Prolog. Technical Report TR-LP-42, European Computer-Industry Research Centre, May 1989.

[5] Clark, K. Parallel logic programming. *The Computer Journal 33*, 6 (1990), 482–493.

[6] Cosytec. *CHIP V4 User Manuals*, 1993.

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

[8] Helm, R., Huynh, T., Marriott, K., and Vlissides, J. An Object-Oriented Architecture for Constraint-Based Graphical Editing. In *Object-Oriented Programming for Graphics* (1995), C. Laffra, E. Blake, V. de Mey, and X. Pintado, Eds., Focus on Computer Graphics, Tutorials and Perspectives in Computer Graphics, Springer-Verlag, pp. 217–238.

[9] Huynh, T., and Marriott, K. Incremental constraint deletion in systems of linear constraints. *(to appear) Information Processing Letters* (1995).

[10] Jaffar, J., and Lassez, J.-L. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference* (Munich, January 1987), pp. 111–119.

[11] Lee, J., and Leung, H. IQ: Towards incremental query processing in non-deterministic concurrent constraint logic programming. In *Proceedings of the 1994 ACM Symposium on Applied Computing* (1994), pp. 405–409.

[12] Lee, J., and van Emden, M. Interval computation as deduction in CHIP. *Journal of Logic Programming 16*, 3 & 4 (1993), 255–276.

[13] Leung, H. *Distributed Constraint Logic Programming.* World Scientific Publishing, 1993.

[14] Lloyd, J. *Foundations of Logic Programming*, second ed. Springer-Verlag, 1987.

[15] Mackworth, A. Consistency in networks of relations. *AI Journal 8*, 1 (1977), 99–118.

[16] Maher, M. Logic semantics for a class of committed-choice programs. In *Proceedings of the Fourth International Conference on Logic Programming* (1987), J.-L. Lassez, Ed., The MIT Press, pp. 858–876.

[17] Southwell, R. *Relaxation Methods in Theoretical Physics.* Oxford University Press, 1946.

[18] Sutherland, I. *SKETCHPAD: a Man-Machine Graphical Communication System.* PhD thesis, MIT Lincoln Labs, Cambridge, MA, 1963.

[19] van Emden, M. Logic as an interaction language. In *Proceedings of the Fifth Conference on Canadian Society for Computational Studies in Intelligence* (1984), pp. 126–128.

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

[21] Van Hentenryck, P., and Deville, Y. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming* (1991), pp. 745–759.

[22] Van Hentenryck, P., and Le Provost, T. Incremental search in constraint logic programming. *New Generation Computing 9* (1991), 257–275.

[23] Van Hentenryck, P., Saraswat, V., and Deville, Y. Design, implementation, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Computer Science Department, Brown University, Providence, RI, USA, 1993.