Encoding Table Constraints in CLP(FD) Based on Pair-wise AC

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

Abstract. We present an implementation of table constraints in CLP(FD). For binary constraints, the supports of each value are represented as a finite-domain variable, and action rules are used to propagate value exclusions. The bit-vector representation of finite domains facilitates constant-time removal of unsupported values. For n-ary constraints, we propose pair-wise arc consistency (AC), which ensures that each value has a support in the domain of every related variable. Pair-wise AC does not require introducing new problem variables as done in binarization methods and allows for compact representation of constraints. Nevertheless, pair-wise AC is weaker than general arc consistency (GAC) in terms of pruning power and requires a final check when a constraint becomes ground. To remedy this weakness, we propose adopting early checks when constraints are sufficiently instantiated. Our experimentation shows that pair-wise AC with early checking is as effective as GAC for positive constraints.

1 Introduction

A table constraint, or extensional constraint, over a tuple of variables specifies a set of tuples that are allowed (called *positive*) or disallowed (called *negative*) for the variables. Recently there has been a growing interest in this format of constraints. This format is well suited to problems where relations are more easily given in extension than in intension such as configuration problems involving datasets (e.g., crossword puzzles). Another reason for the popularity of this format is that certain intensional constraints, especially nonlinear and global constraints, can be more cheaply maintained when tabulated. The table format has been used in the CSP solver competitions and a good collection of problem instances are available. Arc consistency has been generalized for table constraints [5, 12] (called *GAC*) and several data structures have been proposed for maintaining GAC for table constraints [2, 3, 6, 7, 9, 10].

No previous work has been reported on introducing table constraints into CLP(FD). Because of the lack of sophisticated data structures such as multi-dimensional arrays and the necessity of manipulation of tagged data in CLP(FD), an efficient data structure designed for a low-level language may not be suited

to CLP(FD). In this paper, we propose an encoding for table constraints in B-Prolog, a CLP(FD) system.

For a binary table constraint, the supports of each value are represented as a finite-domain variable. When either variable in the constraint is bound to a value, the other variable is unified with the finite-domain variable that represents the supports of the value. Whenever a value is excluded from the domain of a variable in the constraint, the supports of the value in the domain of the other variable are examined and those values that are no longer supported are excluded. As bit vectors are used to represent finite domains, the basic operations required in propagation can be performed efficiently [11].

For an n-ary table constraint, we propose pair-wise arc consistency (AC), which ensures that each value has a support in the domain of every related variable. As for binary constraints, supports of each value are also represented as a finite-domain variable. One of the advantages of pair-wise AC is that it, unlike binarization methods [1], does not introduce new problem variables. The newly introduced finite-domain variables are solely used as bit vectors to represent supports of values. Since supports are not updated during search, no events can occur in these new domain variables. This representation fits CLP(FD) since bits are not tagged individually. Another advantage of pair-wise AC is that constraints can be represented very compactly. Let n be the arity of an n-ary contraint and d be the size of the maximum domain. Supports of values can be represented with $O(n^2 \times d^2)$ space.

Nevertheless, pair-wise AC is weaker than GAC in terms of pruning power because, understandably, it is impossible to use $O(n^2 \times d^2)$ space to represent as many as d^n tuples. To remedy this weakness, we propose adopting early checks to enforce GAC when constraints are sufficiently instantiated. Early checking extends forward checking [8] because the number of variables contained in a constraint can be more than one when the constraint is checked.

For each variable X in a table constraint, a propagator is used to watch the `ins(`X`)` event which is posted when X is instantiated, and another propagator is used to respond to the `dom_any(`X`,`E`)` event which is posted whenever any element E is excluded from the domain of X. Propagators are described using action rules [16]. Our implementation propagates values like the AC-4 algorithm [13], and hence can be classified as fine-grained.

The contribution of this paper is twofold. First, this paper presents an encoding for table constraints which is suited to any CLP(FD) system that represents finite-domains as bit vectors and handles domain value exclusions as events. Second, this paper proposes pair-wise AC, which is a natural extension of AC but has never received much attention before, and proposes to remedy the weakness of pair-wise AC with early checking. We experimented with two different settings for early checking and our experimental results showed that pair-wise AC with early checking is as effective as GAC.

This paper is organized as follows: Section 2 overviews table constraints, consistency algorithms, CLP(FD), and action rules; Section 3 describes an encoding for binary constraints and gives the propagators as action rules; Section 4 gives the propagators for maintaining pair-wise AC; Section 5 proposes several improvements on the propagators; Section 6 describes early checking; Section 7 presents the experimental results; Section 8 discusses the related and future work.

2 Preliminaries

2.1 Table constraints and consistency

A table constraint is either positive or negative. A positive constraint takes the form X `in` R and a negative constraint takes the form X `notin` R where X is a tuple of variables $(X_1, \ldots, X_n)$ and R is a table defined as a list of tuples of integers where each tuple takes the form $(a_1, \ldots, a_n)$. In order to allow multiple constraints to share a table, we allow X to be a list of tuples of variables. In theory, a negative constraint can always be represented as a positive constraint by complementing the table, but in practice this is not always viable since the resulting table can be prohibitively large.

A table constraint is said to be *binary* if each tuple has only two components, and *n-ary* if each tuple has more than two components. A table constraint degenerates into a domain constraint in CLP(FD) if each tuple has only one component.

Let (X_1, X_2) `in` R be a binary table constraint. A value x_1 in the domain of X_1 is said to be *supported* in the constraint if there exists a value x_2 in the domain of X_2 such that (x_1, x_2) is included in R. The constraint is said to be *AC (arc consistent)* on X_1 if every value in the domain of X_1 is supported. The constraint is said to be *AC* if it is AC on both X_1 and X_2.

Let $(X_1, \ldots, X_n)$ `in` R be an n-ary table constraint. Let R_{ij} denote the projection of the table R over the ith and jth columns $(i < j)$. The binary projection of the constraint over X_i and X_j $(i < j)$ is the binary constraint (X_i, X_j) `in` R_{ij}. The n-ary constraint is said to be *pair-wise AC* if all of its binary projections are AC.

Consider the n-ary constraint $(X_1, \ldots, X_n)$ `in` R again. A value x_i in the domain of variable X_i is *gac-supported* in the constraint if there exists a tuple in R whose ith component is equal to x_i. The constraint is said to be *GAC* if every value in the domain of every variable is gac-supported. This condition can be given more formally as:

$$\forall_{i \in \{1..n\}} \forall_{x_i \in X_i} \exists_{x_1 \in X_1, \ldots, x_{i-1} \in X_{i-1}, x_{i+1} \in X_{i+1}, \ldots, x_n \in X_n} (x_1, x_2, \ldots, x_n) \in R$$

where variables are used to denote their domains.

In general, pair-wise AC is a weaker condition than GAC. For example, consider the following constraint:

```
(X,Y,Z) in [(0,1,1),
            (1,0,1),
            (1,1,0)]
```

After the assignment `X=1`, `Y=1`, and `Z=1`, the constraint is not GAC but it is still pair-wise AC.

When a table constraint is generated, tuples of variables and values in the form $(X_1, \ldots, X_n)$ are all transformed into the form $\mathtt{t}(X_1, \ldots, X_n)$ that takes less memory to store and is easier to manipulate.

2.2 CLP(FD)

CLP(FD) [8] is a constraint language that enhances Prolog with built-ins for specifying domain variables, constraints, and strategies for assigning values to variables (called *labeling*). The unification operator is enhanced to deal with domain variables. For two domain variables X and Y, after unification $X = Y$ the elements that are not in both domains are removed and the two variables become aliases.

The following built-ins are used in the implementation of table constraints:

- X `in` D: restricts X to take on a value from D, where D is a set of integers.
- X `notin` D: forbids X to take on any value from D.
- `fd_dom(`X`,`D`)`: D is the list of integers in the domain of X.
- `fd_disjoint(`X`,`Y`)`: The domains of X and Y are disjoint.
- `fd_set_false(`X`,`E`)`: excludes integer `E` from the domain of X. It is equivalent to X`#\=`E but more efficient.

These built-ins are available in B-Prolog. Similar built-ins are also available in other CLP(FD) systems or can be implemented using other primitives.

2.3 Action rules and events

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [16]. It was originally implemented in B-Prolog and now has been introduced into other Prolog systems [4].

An action rule takes the following form:

$$Agent, Condition, \{Event\} => Action$$

where *Agent* is an atomic formula that represents a pattern for agents, *Condition* is a conjunction of conditions on the agents, *Event* is a non-empty disjunction of patterns for events that can activate the agents, and *Action* is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if *Event*

together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

A subgoal is called an *agent* if it can be suspended and activated by events. For an agent α, a rule "H, C, $\{E\}$ => B" is *applicable* to the agent if there exists a matching substitution θ such that $H\theta = \alpha$ and the condition $C\theta$ is satisfied. The reader is referred to [16] for a detailed description of the language and its operational semantics.

The following event patterns are supported for programming constraint propagators:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `bound`(X): when a bound of the domain of X is updated. There is no distinction between lower and upper bounds changes.
- `dom`(X,E): when an *inner* value E is excluded from the domain of X. Since E is used to reference the excluded value, it must be the first occurrence of the variable in the rule.
- `dom`(X): same as `dom`(X,E) but the excluded value is ignored.
- `dom_any`(X,E): when an arbitrary value E is excluded from the domain of X. Unlike in `dom`(X,E), the excluded value E here can be a bound of the domain of X.
- `dom_any`(X): equivalent to the disjunction of `dom`(X) and `bound`(X).

Note that when a variable is instantiated, no `bound` or `dom` event is posted. Consider the following example:

```
p(X),{dom(X,E)} => write(dom(E)).
q(X),{dom_any(X,E)} => write(dom_any(E)).
r(X),{bound(X)} => write(bound).
go:-X :: 1..4, p(X), q(X), r(X), X #\= 2, X #\= 4, X #\= 1.
```

The query `go` gives the following outputs: `dom(2)`, `dom_any(2)`, `dom_any(4)` and `bound`.[1] The outputs `dom(2)` and `dom_any(2)` are caused by `X #\= 2`, and the outputs `dom_any(4)` and `bound` are caused by `X #\= 4`. After the constraint `X #\= 1` is posted, `X` is instantiated to 3, which posts an `ins(X)` event but not a `bound` or `dom` event.

[1] In the current implementation of AR, when more than one agent is activated the one that was generated first is executed first. This explains why `dom(2)` occurs before `dom_any(2)` and also why `dom_any(4)` occurs before `bound`.

3 Binary Constraints

Given a binary table constraint `(X,Y) in R`, we build a hashtable `Hxy` for supports of values of `X`.[2] For each value `Ex` in the domain of `X`, there exists an entry `(Ex,Sx)` in `Hxy` where `Sx` is a finite-domain variable that represents `Ex`'s set of supports in the domain of `Y`. If `Ex` has only one support, then `Sx` is the support itself. Similarly, we also build a hashtable `Hyx` for supports of values of `Y`. After `(X,Y) in R` is posted, it is made AC by excluding all unsupported values from the domains of `X` and `Y`. In the following, `Hxy` and `Hyx` are called *support tables*.

Consider, for example, the following constraint:

```
(X,Y) in [(1,2),(2,1),(3,4),(3,5),(4,4)]
```

The hashtable `Hxy` contains

```
(1,2),(2,1),(3,S3:[4,5]),(4,4)
```

and the hashtable `Hyx` contains

```
(1,2),(2,1),(4,S4:[3,4]),(5,3)
```

where `S3:[4,5]` and `S4:[3,4]` are two finite-domain variables. After the constraint is posted, `X`'s domain becomes `[1,2,3,4]` and `Y`'s domain becomes `[1,2,4,5]`.

The two hashtables `Hxy` and `Hyx` are essentially two tries [6]. Each trie requires, in the worst case, $O(|X| \times |Y|)$ space. This representation is compact because of the indexing effect and the use of bit vectors for domain variables. As will be shown below, supports are never updated during search. Therefore, the domain variables used to represent supports never post any event.

For a binary constraint over `(X,Y)`, we generate propagators to watch `ins` and `dom_any` events on `X` and `Y`. The propagation is very straightforward. When `X` is bound to an integer, `Y`'s domain is reduced to retain only those elements that are supported by `X`. Whenever a value is excluded from the domain of `X`, the supports of the value in the domain of `Y` are examined and those values that are no longer supported by `X` are excluded from the domain of `Y`.

The propagator `watch_ins(X,Y,Hxy)`, defined below, watches `ins(X)` events.

```
watch_ins(X,Y,Hxy),var(X),{ins(X)} => true.
watch_ins(X,Y,Hxy) =>
    hashtable_get(Hxy,X,Sx),
    Y=Sx
```

The propagator is suspended as long as `X` is a variable. The second rule is applied after `X` becomes ground, which unifies `Y` with the set of supports of `X`.

The propagator `watch_dom(X,Y,Hxy,Hyx)`, defined in Figure 1, watches `dom_any` events on `X`.

[2] Hashtables are not available in ISO-Prolog. The built-in `hashtable_get(H,K,Val)` in B-Prolog retrieves from the table `H` the value `Val` with the key `K`. In the real implementation, a hashtable talored to tuples is used when the table is sparse or contain negative integers, and a structure is used otherwise.

```
watch_dom(X,Y,Hxy,Hyx),var(X),{dom_any(X,Ex)} =>
    hashtable_get(Hxy,Ex,Sx), % Sx supports Ex
    fd_dom(Sx,Eys),
    find_support(X,Eys,Y,Hyx).
watch_dom(X,Y,Hxy,Hyx) => true.

find_support(X,[],Y,Hyx).
find_support(X,[Ey|Eys],Y,Hyx):-
    hashtable_get(Hyx,Ey,Sy), % Sy supports Ey
    (fd_disjoint(X,Sy)->fd_set_false(Y,Ey);true),
    find_support(X,Eys,Y,Hyx).
```

Fig. 1. The propagator that watches `dom_any` events.

Whenever a value `Ex` is excluded from the domain of `X`, `Ex`'s set of supports `Sx` is retrieved from `Hxy`. The predicate `find_support` examines every element in `Sx`, and excludes it from the domain of `Y` if it is no longer supported by `X`. In the real implementation, `find_support` is encoded in C which uses bit-wise operations to iterate through the elements of `Sx`.

4 Pair-wise AC for n-ary Constraints

Given an n-ary table constraint `Vars in R`, we build two hashtables `Hxy` and `Hyx` for each pair of variables `X` and `Y` in `Vars`, and generate propagators to watch `ins` and `dom_any` events. In this way, pair-wise AC is maintained.

Since pair-wise AC does not guarantee GAC, an n-ary constraint needs to be checked after it becomes ground. Let `HashR` be the hashtable representation of the table `R`. This *final check* is described as follows:

```
final_check(HashR,Vars),n_vars_gt(1,0),{ins(Vars)} => true.
final_check(HashR,Vars) => hashtable_get(HashR,Vars,_).
```

The condition `n_vars_gt(1,0)` means that the last argument (namely `Vars`) has more than 0 variables.[3] The subgoal is suspended while at least one of the variables in `Vars` is free. After all the variables are instantiated, `hashtable_get` checks if the tuple `Vars` is included in `HashR`.

[3] In general the built-in `n_vars_gt(m,n)` in B-Prolog means that the number of variables in the last `m` arguments of the head is greater than `n`, where both `m` and `n` are integer constants. Notice that the arguments are not passed to the built-in. The system always fetches those arguments from the current frame. This built-in is well used in constraint propagators to change the action when the number of variables in the constraint reaches a certain threshold.

```
register_pair(X,Term):-        % Term=pair(Y,Hxy,Hyx)
     get_attr(X,pairs,L),!,
     attach(Term,L).           % attach Term to the end of L
register_pair(X,Term):-
     L=[Term|_],               % create an open-ended list
     put_attr_no_hook(X,pairs,L), % no default hook on X
     watch_ins(X,L),
     watch_dom(X,L).

watch_ins(X,L),var(X),{ins(X)} => true.
watch_ins(X,L) =>
     ...   % for each term bin(Y,Hxy,Hyx) in L, enforce AC on Y
     ...   % after X is instantiated.

watch_ins(X,L),var(X),{dom_any(X,Ex)} =>
     ...   % for each term bin(Y,Hxy,Hyx) in L, enforece AC on Y
           % after Ex is excluded from the domain of X.
watch_ins(X,L) => true.
```

Fig. 2. The registration procedure.

5 Improvements

In the encoding described above, two propagators are used for each variable in a pair of variables, one watching `ins` and the other watching `dom_any` events on the variable. When a variable is involved in n pairs, $2 \times n$ propagators are generated. One implementation technique for speeding-up propagation in CLP(FD) is to combine the propagators that watch the same event and take similar actions.[4] This technique can be used to speed-up propagation for table constraints too.

For a pair of variables `(X,Y)`, let `Hxy` and `Hyx` be the two support tables. The term `pair(Y,Hxy,Hyx)` is created and registered onto `X` under the attribute name `pairs`. If the attribute `pairs` does not exist yet, the attribute is created and two propagators are generated; if the attribute already exists, then the term is attached to the end of the attribute value, which is an incomplete list with an open end. Figure 2 gives part of the registration procedure. Similarly, the term `bin(X,Hyx,Hxy)` needs to be registered onto `Y`.

The registration procedure is further improved as follows. For a pair `(X,Y)`, if the support tables `Hxy` and `Hyx` represent the Cartesian product of the domains of the variables, it is unnecessary to do the registration at all because every value is guaranteed a support no matter how the variables are instantiated.

Moreover, if a pair has been registered already, we merge the old support tables with the new ones. Let `pair(Y,Hxy,Hyx)` be the term to be registered onto `X`, and `pair(Y,OldHxy,OldHyx)` be a term that has been already registered

[4] This technique is implemented in B-Prolog for constraints such as disequality constraints over two variables.

on `X`. We construct two new support tables `NewHxy` and `NewHyx` where `NewHxy` is the intersection of `OldHxy` and `Hxy`, and `NewHyx` is the intersection of `OldHyx` and `Hyx`. Then we use the built-in `setarg/3` to replace `OldHxy` with `NewHxy` and `OldHyx` with `NewHyx`. This improvement allows inter-constraints sharing.

6 Early Checking

As shown above, pair-wise AC is weaker than GAC in terms of pruning power. The propagators for maintaining pair-wise AC resort to a final check to ensure that a constraint is indeed supported when it becomes ground. To remedy the weak pruning power of pair-wise AC, we can advance this final check to a point when the constraint still contain variables. An early check ensures that every value in the domain of every variable has a supporting tuple. We consider early checking for positive constraints, and similar ideas can be applied to negative constraints too.

There are two possible approaches to checking a constraint: One is to iterate through the values of the domains of the remaining variables in the constraint and, for each combination, we check if it is included in the table; the other is to iterate through the tuples in the table. Since the number of tuples in a given table is normally significantly smaller than the possible combinations of domain values when the number of variables is large, we follow the later approach.

To make it fast to iterate through the tuples in a table, we convert the table to a trie such that common prefixes of the tuples need not be examined more than once for each traversal. We only use one trie per table. The tuples are indexed on the first argument first, then second, and so on. The following defines a propagator that maintains GAC when variables are instantiated.

```
early_check(Trie,Vars),
    {generated,ins(Vars)}
=>
    enforce_gac(Trie,Vars).
```

The predicate `enforce_gac(Trie,Vars)` is also called when the propagator is first created. It first walks through the trie to record all the values that are supported, and then it examines each value in the domain of each variable in `Vars` and excludes it from the domain if it has no supporting tuple.

Since `enforce_gac` does not respond to domain value exclusions, pair-wise AC is still weaker than GAC even with this early checking. To always enforce GAC, we could call `ensure_gac` whenever a change occurs to the domain of any variable.

```
early_check_gac(Trie,Vars),
    {generated,ins(Vars),bound(Vars),dom(Vars)}
=>
    enforce_gac(Trie,Vars).
```

In addition to the `generated` and `ins(Vars)` events, this rule also watches `bound(Vars)` and `dom(Vars)` events. Recall that `bound(Vars)` is posted whenever a bound of the domain of any variable in `Vars` is changed and `dom(Vars)` is posted whenever an inner value is excluded from the domain of any variable. The two events `bound(Vars)` and `dom(Vars)` can be equivalently encoded as `dom_any(Vars)`.

Since `ensure_gac` is expensive, calling it on every change may not pay off. One compromise is to enforce GAC on domain value exclusions only when the constraint contains a certain number of variables. The following gives the refined propagator for early checking.

```
early_check_compromise(Trie,Vars),
    n_vars_gt(1,2),
    {generated,ins(Vars)}
=>
    enforce_gac(Trie,Vars).
early_check_compromise(Trie,Vars) =>
    early_check_bin(Trie,Vars).

early_check_bin(Trie,Vars),
    {generated,ins(Vars),bound(Vars),dom(Vars)}
=>
    enforce_gac(Trie,Vars).
```

Once the number of variables contained in the constraint is 2 or less (the condition `n_vars_gt(1,2)` fails), the propagator is replaced with `early_check_bin` which watches all changes to the domains of the variables.

Recall that the propagators for maintaining pair-wise AC already watch `dom_any` events. One may ask why we need to create a propagator to watch `dom` events here when the constraint becomes binary. The answer is that the support tables used in maintaining pair-wise AC are binary projections and they are never reduced while variables are instantiated. Consider the following example:

```
(X,Y,Z) in [(0,1,1),
            (0,2,2),
            (0,3,3),
            (1,1,2),
            (1,2,3),
            (1,3,1)]
```

The support table `Hyz` from `Y` to `Z` contains the following entries: `(1,S1:[1,2])`, (2,S2:[2,3]), and `(3,S3:[1,3])`. When `X` is bound to `0`, the support table should be reduced to contain `(1,1)`, `(2,2)`, and `(3,3)`. After that, when a value, say `2`, is excluded from the domain of `Y`, the support `2` should be excluded from the domain of `Z`. Nevertheless, because our solver does not reduce support tables, this effect couldn't be achieved without calling `enforce_gac(Trie,Vars)`.

7 Experimental Results

The two built-ins, `in/2` and `notin/2`, in B-Prolog have been extended to allow positive and negative table constraints.[5] For positive constraints, pair-wise AC is used with early checking, which maintains GAC when constraints become ternary. For negative constraints, pair-wise AC is used with forward checking, which maintains GAC only when constraints become unary. For negative constraints, support tables are constructed without complementing given relations.

Thanks to the availability of action rules, the extension was implemented with relative ease. The extension contains about 300 lines of code in Prolog (and action rules) and 1000 lines of code in C, most of which are for preprocessing tables.

We compared pair-wise AC with and without early checking on a selected set of benchmarks used for the N-ARY-EXT category in the CSP solver competitions.[6] The problem instances were translated from XML into Prolog format. The first 5 instances in each of seven selected problem classes were chosen. Each of the problem instances contains at least one positive constraint. Each instance was given a time limit of 500 seconds and no memory limit was imposed. The labeling strategy `ffc` (first-fail, breaking tie by selecting a most constrained variable) was used in all the runs. The machine used was a Pentium 3.0GHz with 1GB of RAM running Windows XP.

Table 1 shows the CPU times. In each row, the first column gives the name of a problem instance, the second column gives the maximum arity of the constraints in the instance, and each of the remaining columns gives the CPU time taken by each of the four different settings: PAC maintains pair-wise AC without early checking; PAC+ET1 triggers early checking after constraints become binary; PAC+ET2 triggers early checking after constraints become ternary; and GAC maintains GAC all the time as is done in `early_check_gac` shown above. Both PAC+ET1 and PAC+ET2 trigger early checking on `ins` events. PAC solved only 16 instances, PAC+ET1 solved 28 instances, and PAC+ET2 and GAC each solved 30 instances. In general, PAC alone is too weak, but it turned out to be the fastest on the bdd benchmarks. There is no remarkable difference between PAC+ET2 and GAC for most of the instances, and PAC+ET2 is faster than GAC on some for the instances such as `crossword_m1c_lex_vg10_11`. Four of the five instances of `renault` were not solved. Profiling the runs indicated that these instances were very memory demanding and most of the execution time was spent on garbage collection.

Table 2 shows the number of backtracks in each run. For those runs that were terminated by time-out events, the numbers were also recorded. For instances that contain only Boolean variables (`bdd` and `jnh`), there is no difference among different settings for early checking since no `dom` or `dom_any` event can occur on

[5] Table constraints are supported in version 7.3 and up.

[6] http://www.cril.univ-artois.fr/∼lecoutre/research/benchmarks/benchmarks.html

Table 1. Comparison on CPU time (seconds).

Problem instance	MAX-ARITY	PAC	PAC+ET1	PAC+ET2	GAC
bdd_21_133_18_78_10	18	1.50	2.70	2.82	2.67
bdd_21_133_18_78_11	18	9.25	31.00	31.00	31.00
bdd_21_133_18_78_12	18	7.85	24.00	24.00	23.00
bdd_21_133_18_78_13	18	9.32	69.00	69.00	69.00
bdd_21_133_18_78_14	18	11.00	33.00	33.00	33.00
crossword_m1c_lex_vg10_11	11	>500	2.96	2.96	12.00
crossword_m1c_lex_vg10_12	12	>500	0.61	0.59	2.61
crossword_m1c_lex_vg11_12	12	>500	0.32	0.32	0.89
crossword_m1c_lex_vg11_13	13	>500	0.12	0.12	0.25
crossword_m1c_lex_vg11_15	15	0.31	0.47	0.47	0.12
jnh01	14	0.15	0.15	0.15	0.15
jnh02	10	0.16	0.16	0.16	0.15
jnh04	11	0.20	0.20	0.20	0.20
jnh05	11	0.15	0.15	0.15	0.15
jnh06	11	0.11	0.11	0.11	0.94
rand_10_20_10_5_10000_0	10	>500	1.42	1.42	1.43
rand_10_20_10_5_10000_10	10	>500	1.29	1.28	1.29
rand_10_20_10_5_10000_11	10	>500	1.87	1.82	2.46
rand_10_20_10_5_10000_12	10	>500	1.31	1.31	1.87
rand_10_20_10_5_10000_13	10	>500	1.37	1.37	1.92
renault_mgd	10	2.50	2.57	2.54	2.57
renault_mod_0	10	>500	>500	>500	>500
renault_mod_10	10	>500	>500	>500	>500
renault_mod_11	10	>500	>500	>500	>500
renault_mod_12	10	>500	>500	>500	>500
ssa_0432_003	5	1.21	0.14	0.14	0.15
ssa_2670_130	5	>500	>500	>500	>500
ssa_2670_141	4	0.000	0.000	0.000	0.000
ssa_6288_047	6	0.40	0.40	0.40	0.40
ssa_7552_038	6	0.47	0.47	0.31	0.31
tsp_20_142	3	>500	>500	81.00	81.00
tsp_20_190	3	>500	286.00	11.00	11.00
tsp_20_193	3	>500	>500	36.00	36.00
tsp_20_1	3	>500	57.00	0.95	0.95
tsp_20_29	3	>500	2.31	0.29	0.29

Boolean variables. For instances that contain no constraint with more than 3 variables (`tsp`), there is no difference between PAC+ET2 and GAC.

We didn't directly compare our solver with other solvers for table constraints. The top ranked solvers, such as mddc, MDG, Mistral, and Abscon solved all the selected instances under a time limit of 1800 seconds.[7] We have to mention that the Windows-XP PC we used is probably slower than the Linux server used in the competition and our solver does not employ any restart strategy. Even under the same condition, it would be unfair to compare a CLP(FD) solver with a solver implemented directly in C or C++ because operations such dereferencing, tagging, and untagging incur measurable overhead in CLP(FD).

[7] http://www.cril.univ-artois.fr/CPAI08/results/results.php?idev=15

Table 2. Comparison on backtracks.

Problem instance	PAC	PAC+ET1	PAC+ET2	GAC
bdd_21_133_18_78_10	0	0	0	0
bdd_21_133_18_78_11	532563	9734	9734	9734
bdd_21_133_18_78_12	321883	13438	13438	13438
bdd_21_133_18_78_13	535481	11154	11154	11154
bdd_21_133_18_78_14	552313	10235	10235	10235
crossword_m1c_lex_vg10_11	91408432	675	675	193
crossword_m1c_lex_vg10_12	11401605	140	140	57
crossword_m1c_lex_vg11_12	10616917	61	61	22
crossword_m1c_lex_vg11_13	2100191	19	19	12
crossword_m1c_lex_vg11_15	0	0	0	0
jnh01	50	50	50	50
jnh02	8	8	8	8
jnh04	1611	1611	1611	1611
jnh05	41	41	41	41
jnh06	826	826	826	826
rand_10_20_10_5_10000_0	>268435455	1010	1010	999
rand_10_20_10_5_10000_10	>268435455	1000	1000	999
rand_10_20_10_5_10000_11	263869300	1003	1003	999
rand_10_20_10_5_10000_12	>268435455	1002	1002	997
rand_10_20_10_5_10000_13	>268435455	1882	1882	998
renault_mgd	13	0	0	0
renault_mod_0	>268435455	17614672	13761572	20704205
renault_mod_10	80724776	9859177	4931019	2200586
renault_mod_11	>268435455	8099239	3215349	1818967
renault_mod_12	251771657	9082875	6301406	2128999
ssa_0432_003	318288	10126	10126	10126
ssa_2670_130	72698460	23508987	23994708	24034014
ssa_2670_141	7	0	0	0
ssa_6288_047	23	23	23	23
ssa_7552_038	476	38	38	38
tsp_20_142	17326326	207966	15036	15036
tsp_20_190	25459227	366646	5750	5750
tsp_20_193	17622012	207826	2814	2814
tsp_20_1	21302231	65937	229	229
tsp_20_29	29304024	2723	31	31

8 Related and Further Work

The key operation used in GAC algorithms is to find a support tuple for a value y in the domain of a variable Y after a value x has been excluded from the domain of a related variable X ($X \neq Y$) [2]. Significant efforts have been made to speed-up this operation by skipping irrelevant tuples that can never been supports for a value [3, 6, 10]. Indexing is an effective technique. The trie data structure [6] indexes tuples such that tuples that have the same prefix share nodes in the trie. In order to facilitate propagating changes originated at every variable in an n-ary constraint, the solver reported in [6] needs to build n tries, one for each variable. An MDD (multi-valued decision diagram) [3] is more effective than a trie in the sense that tuples that have the same suffix also share nodes. The solver `mddc-solv` based on MDD was ranked top in the N-ARY-EXT category in the third CSP solver competition.

Our encoding of binary constraints is similar to the trie encoding. The difference is that the children (leaves) of each interior node are represented as a finite-domain variable rather than a list or an array. This representation fits CLP(FD) since bit-vectors are used in the representation of finite domains and bits in bit vectors are not tagged individually. Any data structure that requires tagging and untagging would incur considerable overhead.

It is well known that any n-ary constraint can be binarized by using a dual representation (i.e., treating each constraint as a variable) or introducing hidden variables for constraints [1]. Experiments have been done to compare various binarization schemes [14]. Our previous solver [17], like the early version of the Mistral solver [7]), introduces a new finite-domain variable for each n-ary constraint and encodes each tuple in the table as an integer. The main problem with that solver was that newly introduced variables could have very bigger domains and the solver could be flooded with events from these domains.

No previous work has been reported on introducing table constraints into CLP(FD). The case constraint in SICStus Prolog is used to implement the built-in `table/2`. Similar built-ins such as `fd_relation/2` in GNU-Prolog and `tuples_in/2` in SWI-Prolog have been implemented, but no detail of the implementation is published. None of these CLP(FD) systems directly supports negative table constraints.

In our solver, supports of values are not updated during search. This makes it possible for constraints to share tables and also renders it unnecessary to trail or copy supports of values. The drawback is that the support tables created for maintaining pair-wise AC for an n-ary constraint cannot be used to enforce AC when the constraint becomes binary. The early-checking propagators in our solver need to use the trie from the original table to enforce AC. Also the operation `fd_disjoint` does not become as cheap as it is supposed to be because the domain that represents supports of a value never shrinks. Recently, a new approach has been proposed that solves n-ary CSPs by reducing tables [10, 15]. It is worthwhile to investigate if this approach can be integrated into our approach.

Further work needs to be done to investigate when and how early checking should be performed. Our solver does not do any early checking on negative constraints. Further investigation should cover negative constraints as well.

9 Conclusion

We have presented an encoding for table constraints in CLP(FD) based on pair-wise AC. In the encoding, the supports of each value are represented as a finite-domain variable, and action rules are used to propagate value exclusions. The encoding is compact and requires no new problem variables. To remedy the weak pruning power of pair-wise AC, we proposed integrating pair-wise AC with early checking. Our experimental results showed that such an integration is effective. Our approach differs from the major GAC algorithms in that it is based on pair-

wise AC and is fine grained. More work remains to be done concerning when and how early checking should be performed.

References

1. Fahiem Bacchus, Xinguang Chen, Peter van Beek, and Toby Walsh. Binary vs. non-binary constraints. *Artif. Intell.*, 140(1/2):1–37, 2002.
2. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
3. Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *CP*, pages 509–523, 2008.
4. Bart Demoen and Phuong-Lan Nguyen. Two WAM implementations of action rules. In *ICLP*, pages 621–635, 2008.
5. Eugene C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11):958–966, 1978.
6. Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197, 2007.
7. Emmanuel Hebrard. Mistral, a constraint satisfaction library. In M.R.C. van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *Proceedings of the Second International CSP Solver Competition*, pages 35–42, 2008.
8. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
9. George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *CP*, pages 379–393, 2007.
10. Christophe Lecoutre. Optimization of simple tabular reduction for table constraints. In *CP*, pages 128–143, 2008.
11. Christophe Lecoutre and Vion Julien. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.
12. Alan K. Mackworth. On reading sketch maps. In *IJCAI*, pages 598–606, 1977.
13. Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
14. Kostas Stergiou and Nikos Samaras. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *J. Artif. Intell. Res. (JAIR)*, 24:641–684, 2005.
15. Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007.
16. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.
17. Neng-Fa Zhou. BPSOLVER 2008. In M.R.C. van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *Proceedings of the Third International CSP Solver Competition*, pages 83–90, 2008.