

AKL(FD) – A concurrent language for FD programming

Björn Carlson¹
Seif Haridi
Sverker Janson

Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
{bjornc, seif, sverker}@sics.se

Abstract

We consider a complete implementation of an extension of the finite domain constraint system FD, including antimonotone constraints, and its integration in AKL, a deep-guard concurrent constraint language. We present the language AKL(FD), together with associated programming techniques. In particular, we show how powerful symbolic constraints can be defined as AKL(FD) programs, which is partly made possible by the extension of FD with a condition combinator and a proper treatment of antimonotone constraints. The AGENTS implementation of AKL(FD) shows competitive performance in a complete programming environment.

1 Introduction

Concurrent constraint programming (CCP) is a general paradigm for concurrent programming, which is characterized by its elegant notions of communication and synchronization based on *constraints* [Mah87, Sar93]. A set of constraints, regarded as formulas in first-order logic, forms a *constraint store*. A number of *agents* interact with the store using the two operations *tell*, which adds a constraint to the store, and *ask*, which tests if the store either entails or disentails the asked constraint, otherwise waiting until it does. Telling and asking correspond to sending and receiving “messages”, thereby providing the basic means for communication and synchronization for concurrent programming.

Constraints also serve as primitives in problem solving programs. The *finite domain* constraint system FD is a framework for solving discrete constraint satisfaction problems in a CCP setting [VHSD91]. FD is suitable as a target for constraint compilers and can be used both for consistency checking and for entailment checking [CC94, CCD94]. We have extended FD with a conditional combinator, and we base our implementation on a proper treatment of the monotonicity of FD expressions.

¹Computing Science Department, Uppsala University, Box 311, 751 05 Uppsala

AKL (Agents Kernel Language) generalizes the basic CCP functionality using a small set of powerful combinators [JH91, Jan94]. The paradigm is that of agents communicating over a constraint store, but the combinators make possible also other readings, depending on the context, where agents compute functions or relations, serve as user-defined constraints, or as objects in object-oriented programs. A major point of AKL is that its paradigms can be combined. For example, it is quite natural to have a reactive process- or object-oriented top-level in a program, with other, encapsulated, components performing constraint solving using don't know non-determinism.

The AGENTS implementation of AKL [JMB⁺94], being developed at SICS, has rational trees as its basic constraint system, which is supported efficiently at the emulator level. Other constraint systems, such as FD are integrated using *generic variables* and *generic constraints*, which are objects offering methods for the services required by the emulator, such as variable binding, garbage collection, and global propagation. This enables a simple integration of arbitrary constraint systems with reasonable efficiency.

In this paper we illustrate by examples how nontrivial arithmetical and symbolic finite domain constraints can be defined as AKL(FD) programs, constraints that have to be provided as primitives in previous CLP systems. Some of the constraints use conditional and disjunctive reasoning, which is formulated completely within AKL(FD). We also give an example of how cardinality reasoning is captured in AKL(FD). Monotone FD rules are used as programming primitives when defining propagating finite domain constraints.

Furthermore, we show that both consistency and entailment checking versions of the constraints can be defined, again within AKL(FD). The entailment of cardinality and disjunctive constraints are thus checked by AKL(FD) programs, using antimonotone FD rules as primitives to implement entailment checking. We also stress the use of monotonicity reasoning when programming with FD, to control the suspension of constraints.

The language AKL(FD) is described, together with brief descriptions of AKL and FD, and we show how the constraint compiler of AKL(FD) can be controlled to generate entailment or consistency checking versions of the constraint compiled. The implementation is sketched, and a short evaluation is given of its performance. However, we emphasize the expressiveness of AKL(FD) in this paper.

The paper is structured as follows: first AKL is presented (Section 2), followed by a presentation of FD (Section 3). AKL(FD) is then described, which includes an overview of the implementation (Section 4). A section follows which contains several examples of how arithmetical and symbolic constraints are defined in AKL(FD), showing the specific programming techniques of AKL(FD) (Section 5). The paper is concluded with a brief performance evaluation (Section 6) and a summary (Section 7).

$S ::=$	$c(\bar{x})$	$constraint$
	$p(\bar{x})$	$call$
	S, S	$composition$
	$\bar{x} : S$	$hiding$
	$(C\% ; \dots ; C\%)$	$choice$
	$\Sigma(x, S, y)$	$aggregate$
$C\% ::=$	$\bar{x} : S \% S$	$clause (\% \in \{\rightarrow, ?, \})$
$D ::=$	$p(\bar{x}) := S.$	$definition$

Figure 1: Syntax of statements in AKL

2 AKL

This section gives a brief and informal summary of AKL, a deep-guard, don't know nondeterministic CCP language. For a complete definition see [JH91], [Fra94], or [Jan94].

Basic statements, such as constraints, calls, composition, and hiding, are as in other CCP languages [Sar93] (see Figure 1). The particular flavor of AKL is given by the deep-guard choice statements, and the family of aggregate statements (which is not discussed here). Choice statements are sequences of clauses. The first statement of a clause is the *guard* and the second the *body*. The operator $\%$ is one of \rightarrow , $?$, and $|$, and to these correspond *conditional* choice, *nondeterminate* choice, and *committed* choice statements, respectively.

The guards of choice statements execute with corresponding local stores. The stores in an execution state form a hierarchy. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, a choice statement fails.

Conditional choice corresponds to if-then-else in Prolog. If the first remaining guard is reduced to a store that is entailed by the union of external stores, the statement is replaced with the body of this clause and the constraints in its store are moved to the closest external store. (The clause is *promoted*).

Nondeterminate choice corresponds to disjunction in Prolog. If only one clause remains, and its guard is successfully reduced to a store which is consistent with the union of external stores, the choice statement is said to be *determinate*. The clause is then promoted. Otherwise, if there is more than one clause left, the choice statement is said to be *nondeterminate*, and it will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, even by subsequent telling in external stores, each of the remaining clauses may be promoted in different copies of the state. The alternative computation

$$\begin{aligned}
N & ::= x \mid i, \text{ where } i \in \mathcal{N} \mid \infty \\
T & ::= N \mid T + T \mid T - T \mid T * T \mid [T/T] \mid [T/T] \mid T \mathbf{mod} T \\
& \quad \mid \mathbf{min}(R) \mid \mathbf{max}(R) \\
R & ::= T..T \mid R \wedge R \mid R \vee R \mid R \Rightarrow R \mid -R \\
& \quad \mid R + T \mid R - T \mid R \mathbf{mod} T \\
& \quad \mid \mathbf{dom}(N)
\end{aligned}$$

Figure 2: Syntax of FD range expressions

paths are explored concurrently. The state reached is called *stable*, and the behaviour is a generalization of the Basic Andorra Model to deep guards.

Up to this point, the constructs introduced belong to the strictly logical subset of AKL, which has an interpretation in first-order logic both in terms of success and failure [Fra94].

Committed choice corresponds to guarded clauses in committed-choice languages. If any of the guards is successfully reduced to a store which is entailed by the union of external stores, the corresponding clause is promoted.

In Section 5 we will see how the choice statements of AKL can be used for constraint programming. A clausal syntax for definitions with choice statements will be used, which corresponds to the above syntax in a straightforward manner.

3 FD

The constraint system FD is based on simple *domain constraints* and functional rules called *indexicals* [VHSD91]. These rules may be thought of both as describing node and arc consistency propagation, and as describing the logical relation between variables.

A domain constraint is an expression $x \in I$, where I is a set of integers. The sets that are considered will always be finite unions of intervals. A set S of domain constraints is called a *store*. The expression x_S denotes the intersection $I_1 \cap \dots \cap I_n$ for all constraints $x \in I_k$ in S . If S does not contain a constraint $x \in I$, x_S is the set \mathcal{Z} of integers. A variable x is *determined in S* if x_S is a singleton set. Let $S_1 \sqsubseteq S_2$, for stores S_1 and S_2 , if for all variables x , $x_{S_2} \subseteq x_{S_1}$.

An indexical has the form $x \mathbf{in} r$, where r is a *range* (generated by R in Figure 2). When applied to a store S , $x \mathbf{in} r$ evaluates to a domain constraint $x \in r_S$, where r_S is the value of r in S (see below).

The value of a range r in S , r_S , is a set of integers computed as follows. The expression $\mathbf{dom}(y)$ evaluates to y_S . The expression $t_1..t_2$ is interpreted as the set $\{i \in \mathcal{Z} : t_{1_S} \leq i \leq t_{2_S}\}$. The operators \vee and \wedge denote union and intersection respectively. The (new) conditional range $r \Rightarrow r'$ equals r'_S if $r_S \neq \emptyset$ and \emptyset otherwise. The conditional range together with union

x_S related to r_S	r monotone	r antimonotone
$x_S \cap r_S = \emptyset$	inconsistent	may become entailed
$x_S \subseteq r_S$	may become inconsistent	entailed
$x_S \not\subseteq (x_S \cap r_S) \neq \emptyset$	may become inconsistent	may become entailed

Table 1: Entailment/Inconsistency of x **in** r in a store S

make constructive disjunction and implication of finite domain constraints definable in FD (see sections 5.3 and 5.4). The expressions $r + t$, $r - t$, and r **mod** t denote the integer operators applied pointwise, where t cannot contain **max** or **min** terms. Finally, the value of $-r$ in S is the set $\mathcal{Z} \setminus r_S$.

The value of a term t in S , t_S , is an integer computed as follows. A number is itself. A variable is evaluated to its assignment, if it is determined in S . The interpretation of the arithmetical operators is as usual. The expressions **min**(r) and **max**(r) evaluate to the infimum and supremum values of r_S , possibly $-\infty$ (∞).

In the following we use $t..$ and $..t$ as shorthand for $t..\infty$ and $-\infty..t$, **min**(x) and **max**(x) as shorthand for **min**(**dom**(x)) and **max**(**dom**(x)). Where nonambiguous, we use t instead of $t..t$.

Let S be a store, and let c be x **in** r .

- S entails c if r_S is defined, and $x_{S'} \subseteq r_{S'}$, for any S' such that $S \sqsubseteq S'$.
- c is consistent in S if for some S' , $S \sqsubseteq S'$, S' entails c .
- c is inconsistent in S if c is not consistent in S .

A range r is *monotone* if for every pair of stores S_1 and S_2 such that $S_1 \sqsubseteq S_2$, $r_{S_2} \subseteq r_{S_1}$, and r is *antimonotone* if for every pair of stores S_1 and S_2 such that $S_1 \sqsubseteq S_2$, $r_{S_1} \subseteq r_{S_2}$. Note that a constant range such as $1..2$ is both monotone and antimonotone. Furthermore, note that a range such as **max**(y)..**min**(y) is antimonotone in any store, and monotone in any store where y is determined. We define x **in** r as monotone (antimonotone) if r is monotone (antimonotone).

The consistency and entailment of x **in** r in a store S is checked by considering the relationship between x_S and r_S , together with the monotonicity of r (see Table 1). Whenever it cannot be decided whether x **in** r is entailed or inconsistent, the indexical acts like a reactive agent, suspending until more information is added to the store, thereby reapplying the test. If r is monotone and $x_S \cap r_S \neq \emptyset$, $x \in r_S$ is added to S , since in any store S' , such that $S \sqsubseteq S'$ and S' entails x **in** r , it holds that $x_{S'} \subseteq r_{S'}$, and thus, by the definition of entailment and the monotonicity of r , $x_{S'} \subseteq r_S$. Hence, monotone indexicals *propagate* domain constraints.

In Section 5 we will see how the reasoning in Table 1 can be used for controlling the execution of constraints in AKL(FD). The basic principle is that, by the above, when an indexical x **in** r is evaluated in a store S ,

where S does not entail x **in** r and r is not monotone, the indexical must suspend. Suppose S' is such that $S \sqsubseteq S'$ and r is monotone in S' . Then, by the above, the domain constraint $x_{S'} \in r_{S'}$ is propagated by x **in** r . Hence, the indexical suspends until it is monotone and then it starts producing domain constraints. By appropriate coding, an indexical can thus be made to suspend in a controlled fashion.

4 AKL(FD)

In AKL(FD), a programmer may use indexicals directly, and has also the option to rely on the AKL(FD) constraint compiler to produce appropriate indexicals for given linear constraints [CC94]. These may be equalities or inequalities in which each term must be linear. For a linear constraint c , a statement `fd(c)` is compiled to monotone indexicals, suitable for consistency checking, and a statement `fd_ask(c)` is compiled to an antimonotone indexical, suitable for entailment checking.

In AGENTS, indexicals are compiled to byte code for a simple stack machine that evaluates range expressions. If during execution it is established (by analysis as in Table 1) that an indexical is entailed, it is dismissed. If it is inconsistent, it causes failure. In all other cases, it suspends on all undetermined variables, annotating the suspensions with the type of dependency (MIN, MAX, MINMAX, or ANY). When the domain of a variable is further restricted, relevant suspensions are reconsidered. Suspended indexicals belonging to the current store or to external stores are reexecuted first. Those belonging to stores below the current one require reinstallation, which is performed lazily, and is a generic service for any constraint system. Local updates of variables belonging to external stores are trailed, enabling execution to move in and out of the local store, undoing and redoing updates. This is, of course, not a complete description; for such we refer to [Car94].

5 Programming in AKL(FD)

The expressiveness of AKL(FD) is illustrated by examples, ranging from the trivial to those using the full potential of AKL(FD). For each example we define both a consistency checking and an entailment checking version of the constraint.

Arithmetical constraints such as $x = y + 1$ are executed by conjunctions of indexicals which are either defined by the user or generated by the compiler.

The traditional method of constrain-and-generate that is prevalent in constraint logic programming is easily adapted to AKL(FD), which is illustrated by the `queens/2` program.

The guard operators \rightarrow and $|$ of AKL enable us to implement constraint-propagating symbolic constraints, as is shown by the `count/3` and `and/3` examples below.

Conditional reasoning can be defined either in terms of the \rightarrow and $|$ operators, or by indexicals with conditional ranges, as exemplified by the `eq_iff/3` predicate below.

Furthermore, we exploit the reasoning of Table 1 for entailment checking and for controlling the execution of indexicals. For example, in the `eq_iff/3` program we use antimonotone indexicals to check the entailment of arithmetical equalities, as well as nonmonotone indexicals to suspend the computation of some constraints.

Programming with nonmonotone expressions is unorthodox in the world of constraint programming. However, we claim that since it is possible to implement monotonicity checking of indexicals efficiently [CCD94], and since the implementation of Table 1 can be made efficient (Section 6), there is a large potential in programming with nonmonotone indexicals.

Indexicals can be used for implementing constructive reasoning with disjunctions, exemplified by the `element/3` program below. This is possible using the range operators \vee and \Rightarrow , basically translating disjunctions to unions of conditional ranges.

AKL(FD) makes different formulations of the same constraint possible. We see this as an asset of the language, since different formulations have different efficiency and different deductive power. The programmer can thus experiment with alternatives, tailoring the constraints to fit the particular application.

User-defined constraints have been recognized as crucial for the versatility of a constraint programming language [DC93b, VHSD91, ECR93, ILO93]. We claim that AKL(FD) is a language where complex finite domain constraints can be defined purely within the concurrent constraint framework, and still execute efficiently both when used for consistency checking and when used for entailment checking.

5.1 Example: arithmetics

First we show how a simple constraint such as $x = y + 1$ can be defined in AKL(FD).

```
'x=y+1'(X,Y) :-
    X in dom(Y)+1,
    Y in dom(X)-1.
```

Hence, X is constrained by the set of possible values of Y pointwise incremented by 1, and Y is constrained by the set of possible values of X pointwise decremented by 1. Since $X=Y+1$ is linear, the constraint can be compiled by the AKL(FD) compiler for both consistency and entailment checking.

Compiling $X=Y+1$ for consistency checking results in the indexicals

```
X in min(Y)+1 .. max(Y)+1,
Y in min(X)-1 .. max(X)-1.
```

The compiler thus generates interval reasoning indexicals. Compiling $X=Y+1$ for entailment checking generates the antimonotone indexical

$$1 \text{ in } \max(X)-\min(Y).. \min(X)-\max(Y)$$

which succeeds when $X-Y=1$ is true.

5.2 Example: n -queens

The CLP technique of first constraining the solution set and then searching through it can be used in AKL(FD). The well-known n -queens program is simply coded as

```
queens(N,L) :-
    construct_domains(N,L),
    constrain(L),
    labeling(L).

constrain([]) :-
    → true.
constrain([D|R]) :-
    → constrain_each(R, D, 1),
    constrain(R).

constrain_each([], _D, _S) :-
    → true.
constrain_each([E|R], D, S) :-
    → no_threat(D, E, S),
    constrain_each(R, D, S+1).

no_threat(X, Y, N) :-
    X in -dom(Y)^(dom(Y)+N)^(dom(Y)-N),
    Y in -dom(X)^(dom(X)+N)^(dom(X)-N).
```

Since the expression $-\text{dom}(Y)(\pm N)$ is antimonotone, the effect of the formulation is that `no_threat/3` suspends until either X or Y is determined, thus making the two constraints monotone (ignoring the fact that $X \neq Y \neq Y \pm N$ may become true during the execution for the time being). Hence, we achieve the intended propagation of \neq [VH89] through the monotonicity reasoning depicted in Table 1. Note that the same constraint in CLP(FD) is defined by the use of a special `value` range function [DC93b], which, thus, we do not need.

Also note that `labeling/1` can be called anywhere in the `queens/2` clause, since the stability condition of AKL guarantees that all determinate work is performed before any don't know nondeterministic step is taken. Hence, the search is not initiated until all constraint propagation is completed.

`no_threat/3` can alternatively be defined by

```

no_threat(X, Y, N) :-
    fd(X≠Y),
    fd(X≠Y+N),
    fd(X≠Y-N)

```

which is a slightly less efficient formulation, since three constraints are used instead of one.

5.3 Example: Atmost

In constraint logic programming systems such as CHIP and cc(FD), cardinality reasoning is performed by builtins [DvHS⁺88, VHSD92]. We give an example of how cardinality reasoning can be programmed in AKL(FD), using the committed choice operator. The cardinality combinator can similarly be defined in AKL(FD) by using the abstraction notion of AKL [JMB⁺94], thereby calling the constraints as predicates.

Consider the constraint *atmost*(*u*, *l*, *v*) which is true iff at most *u* elements in *l* are equal to *v*, where *l* = [*x*₁, ..., *x*_{*k*}]. The constraint can be defined by the formula

$$\sum_{i=1}^k (b)_i \leq u$$

where $(b)_i$ is 1 iff $x_i = v$ is true and 0 iff $x_i \neq v$ is true. The following is an encoding of the formula in AKL(FD).

```

atmost(U, L, V) :-
    N in 0..U,
    count_V(N, L, V).

```

```

count_V(N, [], _) :-
    → N in 0.

```

```

count_V(N, [X|L], V) :-
    → B in 0..1,
    eq_iff(X, V, B),
    fd(B+M=N),
    count_V(M, L, V).

```

```

eq_iff(X, V, B) :- fd_ask(X≠V) | B in 0.
eq_iff(X, V, B) :- fd_ask(X=V) | B in 1.
eq_iff(X, V, B) :- fd_ask(B=0) | fd(X ≠ V).
eq_iff(X, V, B) :- fd_ask(B=1) | fd(X=V).

```

Note the use of the commit operator, which is crucial to the functionality of the predicate. The test $X \neq V$ may well suspend in a situation where $B=0$ succeeds, or vice versa. The use of \rightarrow could unnecessarily suspend the constraint.

`eq_iff/3` can alternatively be defined using \Rightarrow as

```

eq_iff(X, V, B) :-
  X in (max(B)..min(B) ∧ 1) ⇒ dom(V) ∨ (max(B)..min(B) ∧ 0) ⇒ -dom(V),
  V in (max(B)..min(B) ∧ 1) ⇒ dom(X) ∨ (max(B)..min(B) ∧ 0) ⇒ -dom(X),
  B in (max(X)-min(V)..min(X)-max(V) ∧ 0) ⇒ 1 ∨
      -(min(X)-max(V)..max(X)-min(V) ∧ 0) ⇒ 0.

```

Since $\max(B)..min(B)$ and $-(min(B)..max(B))$ are antimonotone ranges, the first two indexicals suspend until B is determined. Similarly the last indexical suspends until X and V are determined. We thus achieve sufficient control of the execution of a user-defined constraint by exploiting the monotonicity of the defining indexicals. Note that the indexicals of the latter `eq_iff/3` behave *exactly* as the builtin range functions `x_to_b` and `b_to_x` of CLP(FD) [DC93b] by the combination of conditional ranges and monotonicity reasoning.

The entailment of `atmost/3` is defined by replacing `eq_iff/3` by

```

eq_iff(X, V, B) :- fd_ask(X≠V) | B in 0.
eq_iff(X, V, B) :- fd_ask(X=V) | B in 1.

```

or alternatively by

```

eq_iff(X, V, B) :-
  B in (max(X)-min(V)..min(X)-max(V) ∧ 0) ⇒ 1 ∨
      -(min(X)-max(V)..max(X)-min(V) ∧ 0) ⇒ 0.

```

That is, `atmost(U, L, V)` succeeds when at most U equations $X_i=V$ are true, where $L=[X_1, \dots, X_k]$.

5.4 Example: Element

We now consider the definition of `element/3` [DSH88]. The constraint exploits constructive disjunction which can be captured by the use of the FD operators \Rightarrow and \vee . The constraint `element(i, [x1, ..., xk], v)` is true iff $\bigvee_{j=1}^k (i = j \wedge x_j = v)$ is true, i.e.,

$$i \in \{j : x_j \in \mathbf{dom}(v)\} \text{ and } v \in \{x_j : j \in \mathbf{dom}(i)\}$$

where we assume x_j is a constant, $1 \leq j \leq k$. Note that this definition gives the same pruning as the builtin `element/3` of CHIP and CLP(FD) does [DSH88, DC93b].

We derive the following program, this time using the \Rightarrow and \vee operators of FD.

```

element(I, L, V) :-
  constrain_I(L, 1, V, I),
  constrain_V(L, 1, I, V).

constrain_I([], _K, _V, I) :-
  → I in 0.

```

$\text{constrain_I}([X|L], K, V, I) :-$
 $\rightarrow K1 \text{ is } K+1,$
 $\text{constrain_I}(L, K1, V, I0),$
 $I \text{ in } (X \wedge \text{dom}(V)) \Rightarrow K \vee \text{dom}(I0).$

$\text{constrain_V}([], _K, _I, V) :-$
 $\rightarrow V \text{ in } \infty.$

$\text{constrain_V}([X|L], K, I, V) :-$
 $\rightarrow K1 \text{ is } K+1,$
 $\text{constrain_V}(L, K1, I, V0),$
 $V \text{ in } (\text{dom}(I) \wedge K) \Rightarrow X \vee \text{dom}(V0).$

An even stronger formulation could be used for i and v above, i.e. as

$$i \in \{j: \mathbf{dom}(x_j) \cap \mathbf{dom}(v) \neq \emptyset\} \text{ and } v \in \cup\{\mathbf{dom}(x_j): j \in \mathbf{dom}(i)\},$$

which is given by replacing $(X \wedge \text{dom}(V))$ in $\text{constrain_I}/4$ with $(\text{dom}(X) \wedge \text{dom}(V))$, and by replacing $(\text{dom}(I) \wedge K) \Rightarrow X$ in $\text{constrain_V}/4$ with $(\text{dom}(I) \wedge K) \Rightarrow \text{dom}(X)$. This is a proper extension to the traditional `element/3`, such that the elements of the list need not be determined at evaluation time.

The entailment of `element(I, L, V)` is defined by the statement

$$(I=1 \mid \text{fd_ask}(X_1=V) ; \dots ; I=k \mid \text{fd_ask}(X_k=V))$$

or alternatively by the indexical

$$0 \text{ in } (I \wedge 1) \Rightarrow X_1=V \vee \dots \vee (I \wedge k) \Rightarrow X_k=V$$

i.e., `element(I, L, V)` succeeds if there exists j , $1 \leq j \leq k$, such that $I = j$ implies $X_j = V$.

5.5 Example: And

A finite relation can naturally be defined as a set of tuples. However, in many cases the relation maintains a certain relationship between different arguments, e.g. consider boolean *and* for which $\text{and}(x, y, 1)$ is true only if $x = y = 1$. Such internal relationships introduce conditional reasoning which can be used for constraint propagation. We now show an example of how the commit operator of AKL can be used for the conditional reasoning of finite relations.

We define $\text{and}(x, y, z)$, which is true iff $x \wedge y = z$, as

$\text{and}(X, Y, Z) :- \text{fd_ask}(X=0) \mid \text{fd}(Z=0).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(Y=0) \mid \text{fd}(Z=0).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(X \neq Z) \mid \text{fd}(Y=0).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(Y \neq Z) \mid \text{fd}(X=0).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(Z=1) \mid \text{fd}(X=1, Y=1).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(X \neq Y) \mid \text{fd}(Z=0).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(X=1) \mid \text{fd}(Y=Z).$

$\text{and}(X, Y, Z) :- \text{fd_ask}(Y=1) \mid \text{fd}(X=Z).$
 $\text{and}(X, Y, Z) :- \text{fd_ask}(X=Y) \mid \text{fd}(X=Z).$

This definition is to be compared with the similar definitions of *and* in CLP(FD) definition, and in cc(FD).

In CLP(FD), $\text{and}(x, y, z)$ is defined by a conjunction of indexicals, and only domain constraints of the kind $x \in \{0\}$ ($x \in \{1\}$) can be propagated [DC93a], which are weaker than constraints of the type $x = y$. Of course, the version of CLP(FD) specialized to boolean indexicals achieves very efficient propagation of domain constraints [DC93b], however, the point we want to make is that in AKL(FD) more general constraint propagation can be implemented.

In cc(FD), $\text{and}(x, y, z)$ is defined by a conjunction of implications, ($X=0 \rightarrow Z=0, \dots, X=Y \rightarrow X=Z$), similar to the clauses above (omitting the fd and fd_ask operators) [HSD92]. Thus the same constraints are propagated as in AKL(FD). However, in cc(FD) there is a problem of propagating redundant copies of constraints.

Consider, for example, the atom $\text{and}(X, Y, 1)$. In AKL(FD) this atom is replaced by $\text{fd}(X=1, Y=1)$. In cc(FD) it is replaced by $(X=0 \rightarrow Z=0, \dots, X=Y \rightarrow X=Z)$, which is further reduced to $(X=1, Y=1, X=1 \rightarrow Y=1, Y=1 \rightarrow X=1, X=Y \rightarrow X=1)$, which is finally replaced by $(X=1, Y=1, Y=1, X=1, X=1)$. That is, redundant constraints are propagated due to the lack of commit.

6 Performance Evaluation

For a simple performance evaluation of the system, we have timed four standard benchmark programs written in AKL(FD).

The queens program, finding one solution to the 16-queens problem, and collecting all 92 solutions to the 8-queens problem.

The alpha program [DC93b], which is a crypto-logical puzzle similar to the send-more-money problem, finding one solution.

The 20 equations problem [DC93b], which is a linear arithmetic problem, finding one solution.

The car sequencing program [DSH88], finding one solution to the standard instance of the problem. The cars program uses the definitions of `element/3` and `atmost/3` above, where we tried both the version of `eq_iff/3` using `|` and the version using `=>`. Furthermore, to give an indication of the efficiency of constraints implemented in AKL(FD), we also tried a C-version of `element/3` (we compile `element(I,L,X)` to a conjunction of indexicals, `X in element_x(I,L) & I in element_i(X,L)`,

example	AGENTS – labeling		CLP(FD)	ECLiPSe
	AKL (Run+Copy)	C		
16-queens	8210 (3390+4820)	2320	1140	9720
all 8-queens	1060 (460+600)	290	190	1320
alpha ff	810 (410+400)	240	120	2300
eq20	1200 (770+430)	500	220	750
cars C	260 (150+110)	90	50	130
cars \Rightarrow	900 (520+380)	260		
cars	950 (560+390)			

Table 2: Performance comparison

where $\text{element_x}(i, l) = \{l_j : j \in \mathbf{dom}(i)\}$ and $\text{element_i}(x, l) = \{j : l_j \in \mathbf{dom}(x)\}$, where $l = [l_1, \dots, l_k]$.

(For a copy of the example code contact agents-request@sics.se.)

All the timings are in milliseconds, and are computed on a SPARC-2 system. “ff” is used to indicate if the first-fail principle was used [VH89], and otherwise naïve labeling was used. We tried labeling procedures written in C and in AKL(FD). Since general don’t know nondeterminism in AGENTS is achieved by copying, we give the copy time for each example (see Table 2). The labeling procedure written in C uses trailing instead of copying.

We have included the performance figures of CLP(FD) and ECLiPSe run on the same programs and on the same machine as a comparison.

The figures indicate that AGENTS is on average 2 times slower than CLP(FD), using labeling in C (without a copying overhead), and on average 8 times slower when labeling in AKL(FD). The difference in speed, disregarding the copying overhead, is primarily explained by the fact that CLP(FD) compiles to C, and inlines FD in the emulator [DC93b].

CHIP [DvHS⁺88] is reported to be on average 2 times slower than CLP(FD) [DC93b], and hence AKL(FD) is comparable in speed to CHIP when labeling by trailing.

Running the problems in ECLiPSe [ECR93], indicates that AGENTS is comparable in speed including the copying overhead, and about 4 times as fast by labeling in C. ECLiPSe provides powerful constraint definition mechanisms, based on meta-terms and propagation rules, for example, the main part of the finite domain constraints are defined as source programs.

AGENTS is parameterized with constraints at the C level. We thus provide an alternative approach to user defined constraints, completely captured within the concurrent constraint framework, which is comparable in speed to state-of-the-art finite domain systems.

7 Concluding Remarks

We have presented AKL(FD), a deep-guard concurrent constraint programming language for finite domain constraint solving. By using entailment-checking nonmonotone indexicals and the choice statements of AKL or the new condition combinator for FD, powerful symbolic constraints can be defined in the spirit of the glass-box approach of FD.

AGENTS offers a complete implementation of AKL(FD), and is comparable in speed with state-of-the-art finite domain CLP languages. It is free for research and education purposes. (Contact agents-request@sics.se.)

Future research includes

- compiling implications and disjunctions of finite domain constraints, where disjunction is constructive [VHSD92]. Constraints are compiled differently whether used for consistency or entailment checking, and FD with conditional ranges will be used as target language [CC94].
- implementing constraint lifting, a generalization of constructive disjunction to nondeterminate agents. The lifting operation is typically union of finite domains, or anti-unification of terms.

Acknowledgments.

We would like to thank Torkel Franzén, Johan Montelius, and the other members of the Concurrent Constraint Programming group at SICS, Mats Carlsson of the Logic Programming and Applications group at SICS, and also other members of the ESPRIT Project 7195 (ACCLAIM), in particular Daniel Diaz at INRIA-Rocquencourt, for their contributions to this work.

References

- [Car94] Björn Carlson. The implementation of AKL(FD). Forthcoming research report, SICS, 1994.
- [CC94] Björn Carlson and Mats Carlsson. Compiling finite domain constraints. Forthcoming research report, SICS, 1994.
- [CCD94] Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of finite domain constraints. In *Proceedings of the International Conference on Logic Programming*. MIT Press, 1994.
- [DC93a] D. Diaz and P. Codognet. A boolean extension of clp(FD). In *Proceedings of the International Symposium on Logic Programming*. MIT Press, 1993.
- [DC93b] D. Diaz and P. Codognet. Compiling constraints in clp(FD). Research report, INRIA, 1993.

- [DSH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car sequencing problem in constraint logic programming. In *European Conference on Artificial Intelligence*, 1988.
- [DvHS⁺88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.
- [ECR93] ECRC. *ECLiPSe User Manual*, 1993.
- [Fra94] Torkel Franzén. Some formal aspects of the Andorra Kernel Language. SICS Research Report R94:10, Swedish Institute of Computer Science, May 1994.
- [HSD92] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.
- [ILO93] ILOG. *ILOG Solver User Manual*, 1993.
- [Jan94] Sverker Janson. *AKL—A Multiparadigm Programming Language*. Uppsala theses in computing science 19, Uppsala University, June 1994. (Also available as SICS Dissertation Series 14.).
- [JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.
- [JMB⁺94] S. Janson, J. Montelius, K. Boortz, P. Brand, B. Carlson, R. C. Haygood, B. Danielsson, and S. Haridi. *AGENTS User Manual*. SICS, 1994.
- [Mah87] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Logic Programming: Proceedings of the Fourth International Conference*. MIT Press, 1987.
- [Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. MIT Press, 1993.
- [VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Unpublished manuscript, 1991.
- [VHSD92] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint Logic Programming over Finite Domains: the Design, Implementation, and Applications of cc(FD). Technical report, Computer Science Department, Brown University, 1992.