Constraint Based Reasoning with Constraint Logic Programming and Array Based Logic

Christopher Turner
turner@qucis.queensu.ca

November 1996
External Technical Report
ISSN-0836-0227-
96-402

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared November 15, 1996
Copyright ©1996 Christopher Turner

Abstract

The paper describes how Constraint Based Reasoning (CBR) can be performed with two different paradigms, Constraint Logic Programming (CLP) and Array Based Logic (ABL). The author describes the operation of Constraint Logic Programming emphasizing CLP techniques for finite domain problems such as search strategies and consistency techniques. An explanation of Array Based Logic is presented including a description of methods for creating, joining and compressing ABL relations as well as an heuristic for building a system of relations in ABL. A familiar cryptogram is used as an example to demonstrate the operation of the two approaches for finite domain constraint problems. Some potential avenues of research are also presented.

1 Introduction

In recent years, there has been a substantial increase in interest in Constraint Based Reasoning (CBR) among computer scientists and engineers. The great potential of CBR as a problem solving tool is becoming increasingly apparent. Known areas of application for CBR are as surprising as they are diverse and range from operations research to vision. Moreover, any combinatorial search problem may be amenable to a CBR solution. Constraint Logic Programming (CLP) and the lesser known Array Based Logic (ABL) are two methods for performing CBR. CLP developed from and is a superset of Logic Programming. Similarly, ABL for general constraint domains developed from earlier work on array based boolean logic. In this paper we describe these two different paradigms, with emphasis on their operation on the finite domain constraint domain. The first section introduces the reader to important background issues such as CBR, constraint domains, and Constraint Satisfaction Problems. Next, an overview of CLP is presented. The next section is a description of ABL. The paper concludes with a section devoted to a discussion of interesting issues regarding the two approaches and some suggestions for possible avenues of research.

1.1 Constraint Based Reasoning

Constraint Based Reasoning involves the formulation of problems as sets of constraints on variables. A solution to a problem thus formulated is an assignment to the variables satisfying all the constraints. The term *constraint* has been used in a wide variety of ways in the Artificial Intelligence literature. Informally speaking, a constraint states a relationship among variables. (Note that in the next section we give a more formal definition of constraints as part of our treatment of constraint domains.) Constraints can be expressed in many ways, including as mathematical relations, as inequalities, as logical formulae and as sets of tuples. Some examples of constraints are : $X+Y \leq 10$; $(t \wedge u) \Rightarrow v$; $F = \frac{9}{5}C + 32$ and $\{(x,y,z) \mid x,y,z, \in \mathcal{N}, 0 < x < y < z < 10\}$. Applications of CBR include circuit analysis [51], diagnosis [21], job shop scheduling [48], car production [12], spatial reasoning [22, pp. 5-16] and scene interpretation [57, 58] to name but a few. For an introduction to CBR, which includes an explanation of the concept of *dynamic constraints*, or constraints which themselves are constrained, see [22].

1.2 Constraint Domains

In this section we define constraint domains and give three important examples of them. Before doing so, we present some preliminary definitions and notation. A *signature* Σ is a set of function and predicate symbols. A Σ-*structure* $\mathcal{D}$ is a set D and an assignment of functions and relations on D to the elements of Σ where D is the set from where the values of the variables and constants involved in the constraints are drawn. A *primitive constraint* has the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots t_n$ are terms and p is a predicate symbol in Σ. A first order Σ-$formula$ is a first order formula

constructed from the primitive constraints and the logical connectives $\neg, \wedge, \vee, \Rightarrow, \Leftarrow$ and the quantifiers $\forall$ and $\exists$ [35, p.6].

A *constraint domain* is an ordered pair $(\mathcal{D}, \mathcal{L})$. $\mathcal{D}$ is referred to as the *domain of computation* and is a Σ-*structure* as defined above. $\mathcal{L}$ is the set of constraints for the domain. It is a set of first order Σ-*formulae* generated from the primitive constraints of the domain. It is usually a proper subset of all the possible Σ-*formulae* which can be generated in this manner.

The first constraint domain we consider is the Herbrand constraint domain. (This is actually the constraint domain used in the Logic Programming language Prolog.) Its signature Σ contains function symbols and the predicate $=$. D is the set of finite trees whose nodes are labelled with a function symbol or a constant term such that each node has k ordered children where k is the arity of the node. (In other words, D is the set of tree representations of terms.) The Σ-*structure* $\mathcal{D}$ generates trees by interpreting the finite trees of D as children of the function symbols (and of the predicate $=$) of Σ. The primitive constraints are equations between terms, *i.e.*, finite trees generated by $\mathcal{D}$ whose root is the equals predicate. The constraints $\mathcal{L}$ are the first order formulae generated from the primitive constraints. A sample constraint in $\mathcal{L}$ is $f(x, y) = h(x, y) \wedge f(g(y), x)$.

A second example of a constraint domain of importance to this paper is the boolean two-valued logic domain. Here, Σ contains $0, 1, \vee, \wedge, \oplus, \Rightarrow$ and the single predicate $=$. The set D is simply $true, false$. $\mathcal{D}$ treats the symbols of Σ as boolean functions. For example, $\vee$ is the boolean 'or' operator. The set of constraints $\mathcal{L}$ is the full set of first order logic formulae generated from the primitive constraints. $(\mathcal{D}, \mathcal{L})$ is $\mathcal{BD}$, the boolean constraint domain. Two examples of constraints from this domain are: $(x \Rightarrow y) \wedge z = 0$ and $a \oplus b = c \vee d$.

Another important constraint domain is finite domain ($\mathcal{FD}$). It is the domain associated with constraint problems with the integers. For finite domain, $D = \mathcal{Z}$ and $\Sigma = \{\{\in [m, n]\}_{m \leq n}, +, =, \neq, \leq\}$. For two integers m and n, $x \in [m, n]$ is the interval constraint meaning $m \leq x \leq n$. Here $\mathcal{L}$ is the set of constraints generated from the primitive constraints such that every variable has an interval constraint associated with it.[1] An example of a finite domain constraint is:

$$x \in [10, 32] \wedge y \in [1, 5] \wedge x + 10y \neq 76 \wedge x + y \leq 12$$

As an example of a finite domain constraint problem, consider the often cited cryptogram shown in Figure 1. Each of the letters may take on any integer value from 0 to 9. The columns must be added in the usual fashion. This results in one arithmetic constraint for each of the five columns. An additional constraint is that the letters must all be different within each solution. The five arithmetic constraints are shown below the cryptogram. The carry digits in the problem are represented by R1, R2, R3 and R4 and may be either 0 or 1.

Other constraint domains include pseudo-boolean constraints [6], order-sorted feature algebras [3], domains of functions expressed by lambda expressions [40] and do-

[1] Note that the constraint domain $\mathcal{BD}$ is actually included in $\mathcal{FD}$.

$$
\begin{array}{rr}
 & \text{SEND} \\
+ & \text{MORE} \\
\hline
 & \text{MONEY}
\end{array}
$$

$$
\begin{aligned}
\text{M} &= \text{R1} \\
\text{S} + \text{M} + \text{R2} &= \text{O} + 10 \times \text{R1} \\
\text{E} + \text{O} + \text{R3} &= \text{N} + 10 \times \text{R2} \\
\text{N} + \text{R} + \text{R4} &= \text{E} + 10 \times \text{R3} \\
\text{E} + \text{D} &= \text{Y} + 10 \times \text{R4}
\end{aligned}
$$

Figure 1: The Send More Money Cryptogram Problem

mains of finite sets [14]. For a full discussion of constraint domains, see [28, pp.510-516].

Closely related to information systems [50] is the concept of *constraint systems* [49] which are an alternate, more general formalization of constraints which have been proposed by Saraswat.

1.3 Constraint Satisfaction Problems

Concurrently with CLP research, a great deal of research has been done on *Constraint Satisfaction Problems* or CSPs. CSPs are commonly associated with the finite domain constraint domain. However, the constraints involved are usually not restricted to the arithmetic constraints discussed in Section 1.2. A CSP consists of a set of variables whose domains are finite and a set of constraints which restricts the values the variables can simultaneously take. A solution to a CSP is an assignment of values to the variables that satisfies all the constraints. Examples of CSPs are the graph colouring problem [52, pp. 19-21], as well as the cryptogram earlier described. In addition to the constraints being satisfied, some CSPs require that a quantity be optimized. These are known as *Constraint Satisfaction Optimization Problems* or CSOPs. As an example, consider the well known job shop scheduling problem [48]. The term *combinatorial search problem* is often used to refer to either CSPs or CSOPs.

We now give a formal definition of a particular CSP whose constraints are expressed explicitly as tuples. The Finite CSP [36] is a sextuple $(X, \Delta, \delta, C, \Sigma, \sigma)$. X is the set of variables found in the problem. Δ is the set of domains of the variables in X. δ is a function that maps each variable $x \in X$ to its domain $\delta(x) \in \Delta$. C is the set of constraints. Σ is a set of sets of variables in X such that $\bigcup \Sigma = X$. σ is a one-to-one function that maps each constraint $c \in C$ to a set of variables in Σ. $\sigma(c)$ is called the *scope* of the constraint c. It is the set of the variables in X to which the values in the constraint apply. Each constraint in C is the subset of the Cartesian product of the domains of the variables in its scope that the variables may attain. For example, consider the Finite CSP $P = (X, \Delta, \delta, C, \Sigma, \sigma)$ where:

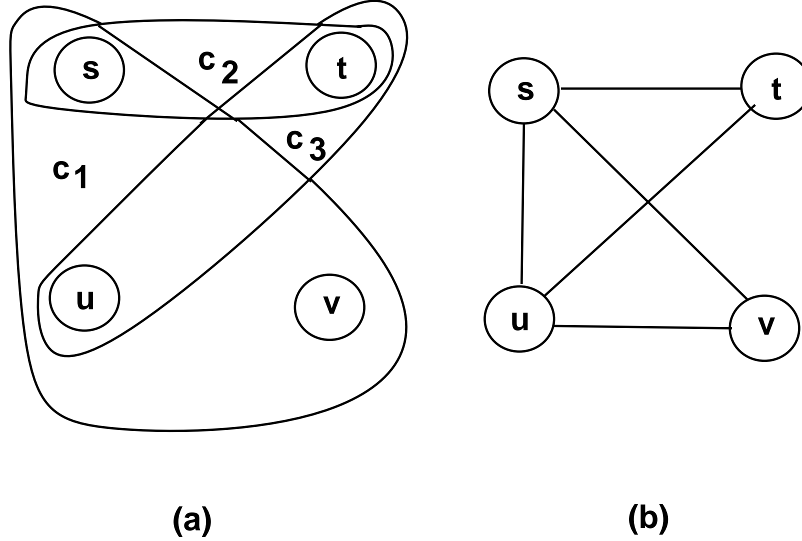

Figure 2: Graph Representations of a CSP

$X = \{s, t, u, v\}$
$\Delta = \{\{i \in \mathcal{N} \mid 0 \leq i \leq 9\}\}$
$\delta(s) = \{1, 2, 3, 4\},\ \delta(t) = \{3, 5, 7\},\ \delta(u) = \{6, 7, 8, 9\},\ \delta(v) = \{2, 4, 6, 8, 9\}$
$C = \{c_1, c_2, c_3\}$ such that $c_1 = \{(1, 7, 9), (4, 9, 2), (3, 6, 6)\}$,
$c_2 = \{(4, 7), (3, 3), (1, 5), (2, 5)\}$ and $c_3 = \{(7, 6), (5, 7), (5, 9)\}$
$\Sigma = \{s_1, s_2, s_3\}$ such that $s_1 = (s, u, v),\ s_2 = (s, t)$ and $s_3 = (t, u)$
$\sigma(c_i) = s_i,\ i = 1, \ldots, 3$

The variable assignment $s = 1,\ t = 5,\ u = 7,\ v = 9$ is a solution to the problem. Note that Finite CSPs are also known as Consistent Labelling Problems, Consistency Satisfaction Problems, Constraint Networks and Networks of Relations.

Constraint hypergraphs are convenient representations of CSPs. A *constraint hypergraph* is a hypergraph (V, E). The set of vertices V is the variable set X. The edges in the set E correspond to the constraint scopes in Σ. The hypergraph shows the interconnections of variables within constraints. The constraint hypergraph for the above problem is shown in Figure 2(a). Similar to constraint hypergraphs are *primal graphs*. The nodes of primal graphs are the CSP variables as in the hypergraph representation. For every constraint in the problem, there is an edge in the primal graph between each pair of variables in the scope of the constraint. The primal graph for the above finite CSP is shown in Figure 2(b).

We conclude this section with some remarks about CSPs. If a CSP has a solution, *i.e.*, if there is an assignment to its variables which satisfies all of its constraints, then it is said to be *satisfiable*. Otherwise, it is *unsatisfiable*. The *order of a constraint* is the number of variables involved in that constraint. The *order of a CSP* is the order of its highest order constraint. In particular, the order of the above Finite CSP is 3. A CSP of order 2 is said to be a *binary* CSP whereas those of higher order are often

called *general* CSPs. For binary CSPs the hypergraph representation discussed above is reduced to an ordinary graph. In addition, for binary CSPs the primal graph and the constraint hypergraph will be the same.

The Finite CSP, like most CSPs, is known to be NP-complete [36]. However, as we shall see in Section 2 heuristic problem reduction and search techniques have been developed with a view to solving CSPs efficiently. It is important to note that the term *constraint satisfaction* has been used casually within the Artificial Intelligence community to refer to problems which do not necessarily have the above specification of the CSP. In particular, some of these references are to problems whose variables have infinite domains. For a complete introduction to the techniques and issues regarding CSPs, the reader is referred to [52].

2 Constraint Logic Programming

Although effective for solving many problems in the Herbrand Universe, the Logic Programming paradigm has not proven to be a very practical way of solving problems in other constraint domains including the discrete combinatorial problems of finite domain such as the cryptogram given in Section 1.2. For these, a more general problem solving paradigm is necessary. Constraint Logic Programming incorporates Constraint Based Reasoning into the Logic Programming paradigm. It was devised by Joxan Jaffar and Jean-Louis Lassez [26, 27]. They noted that languages based on logic and definite clauses could be separated into classes according to the domain of computation and constraints on which they operate.[2] Based on this notion, a parameterized schema $CLP(\chi)$ where χ is a quadruple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$ has been adopted as a means of classifying CLP languages. Here, Σ, $\mathcal{D}$ and $\mathcal{L}$ are all as defined in Section 1.2. $\mathcal{T}$ is a Σ-*theory*, a collection of closed Σ-$formulas$. It is a set of axioms that apply to the language.

The reader should be aware that in practise CLP languages are usually designated simply by the constraint domain on which they operate, *e.g.*, CLP($\mathcal{FD}$) is the usual way of designating CLP languages for finite domains.[3] In this paper we describe in detail the operation of CLP($\mathcal{FD}$) languages. Other CLP languages include CLP($\mathcal{R}$) (real numbers) [29] and CLP(Σ^*) (regular sets) [55]. More generally, a CLP language may be defined on any of the constraint domains discussed in Section 1.2.

CLP languages have built-in predicates for expressing constraints in their constraint domain. In addition, many CLP systems incorporate constraints that are not strictly part of the domain on which they operate. Known as *complex constraints* [28, pp.528-529], in some cases they are essentially just boolean combinations of the primitive constraints of the constraint domain of the language. On the other hand, many CLP languages incorporate application specific constraints in an *ad hoc* manner that are not part of any general purpose constraint domain. The reader should note that the SLD resolution schema used in Logic Programming is maintained in CLP.

[2] See Section 1.2 for definitions of these concepts.

[3] Most existing CLP systems actually implement more than one class of CLP languages.

At each choice point, however, unification is replaced by the more general concept of *constraint solving.*

In this section we describe Constraint Logic Programming, focusing on those concepts most relevant to the understanding of CLP for finite domains. (For a complete survey of Constraint Logic Programming, with a bibliography containing more than 260 items, the reader is referred to [28].) We begin this section with a description of the Logic Programming paradigm. The structure of CLP programs is then discussed. Next comes a semantic description of how CLP programs execute. The important CLP concept of incrementality is the topic of the next section. A section devoted to important CLP operations follows. Lastly, we present a detailed description of the operation of Constraint Logic Programming for finite domains.

2.1 Logic Programming

In this section we introduce the reader to the essential concepts of Logic Programming, especially those which are most relevant to Constraint Logic Programming. Logic Programming is a declarative programming language paradigm which can be used to express problems in first order logic. Because of efficiency considerations, most programming language implementations of Logic Programming restrict themselves to a subset of full clausal logic. Prolog is the best known Logic Programming language. A Prolog program is a set of positive Horn clauses, also known as definite clauses. Execution of Prolog programs is a derivation based on a rule of inference called SLD[4] resolution [32].

Derivations based on resolution are proofs by contradiction of existential queries. Starting with the initial goal or query, the current goal and the head of one the definite program clauses are repeatedly resolved. The most general unifier of the two is determined and maintained. The result of the resolution is an inference which becomes the new goal. Conceptually speaking, the set of all possible resolution inferences can be thought of as a tree and the derivation as a search of the tree. In general there is a choice as to which clauses to resolve and as to which subclause becomes the new goal. For this reason, nodes in the inference tree are known as *choice points*. The tree is usually searched in depth first manner. If at any step in the search resolution cannot be performed, the system backtracks to a previous choice point and another clause is selected for resolution.

The original resolution principle is due to Robinson [46]. For a full description of Logic Programming, see, for example, [35]. For a slightly anecdotal paper by Robinson himself about Logic Programming, including historical background and detailed descriptions of resolution and unification, see [47].

2.2 Program Structure

In this section we discuss the structure of CLP programs. The term constraint is used as it is defined in Section 1.2. Π is the set of predicate symbols usable by the

[4] *S*elective *L*inear Resolution for *D*efinite Clauses

program. An *atomic formula* or *atom* is a predicate having the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $p \in \Pi$. A CLP program [28, p.509] is a sequence of rules having the form $a \leftarrow b_1, \ldots b_n$ such that a is an atom and the b_i are atoms or constraints. a is called the *head* of the rule whereas $b_1 \ldots b_n$ is called the *body.* A *goal* or *query* G is a conjunction of constraints and atoms. A *fact* is a rule $a \leftarrow c$ where c is a constraint.

2.3 Program Execution

In this section, we present the operational semantics of the top-down method of CLP program execution as well as a description of CLP derivations.[5] The semantics of top-down execution are defined in terms of transitions on computational states. A computational state of a CLP program is a triple $\langle A,\ C,\ S\rangle$. A is a multiset of atoms and constraints; C and S are both multisets of constraints. C and S collectively make up the *constraint store.* A constraint contained in C is said to be *active* or *awake*; one in S is said to be *passive* or *asleep.* (In Logic Programming, there are no passive constraints and the only active constraint is the current most general unifier.) There is a special computational state called $fail$. In addition, there is a predicate *consistent* and a function $infer$.

The initial computational state is $\langle G,\ \emptyset,\ \emptyset\rangle$ where G is the initial goal or query of the computation.[6] Associated with each CLP system is a *computational rule.* The computational rule governs the execution of the CLP program running on the system. Given the sequence of previous computational states and transitions in the computation, the rule selects the transition which determines the next state.

There are four kinds of transitions for top-down execution. They are: resolution, constraint transfer, constraint store management, and a consistency test. The semantics of each type of state transition and a brief description are given below.

Resolution:

A resolution transition is defined as follows:

$$\langle A \cup \{a\},\ C,\ S\rangle \rightarrow_r \langle A \cup B,\ C,\ S \cup \{s_1 = t_1, \ldots, s_n = t_n\}\rangle$$

where a is the atom chosen by the computational rule, $h \leftarrow B$ is a rule of the program, and $h = p(t_1, \ldots, t_n)$ and $a = p(s_1, \ldots, s_n)$. Alternatively, resolution is defined as

$$\langle A \cup a,\ C,\ S\rangle \rightarrow_r \ fail$$

if the computational rule selects the atom a and for every rule $h \leftarrow B$ the predicate symbols of h and a differ.

[5]For a description of its less common alternative called bottom-up execution, see [28, pp.523-525].
[6]Note that constraints can be used to specify the goal G.

Constraint Transfer:

$$\langle A \cup c,\ C,\ S\rangle \rightarrow_c \ \langle A,\ C,\ S \cup c\rangle$$

Here a constraint is transferred from the current goal to the passive store.

Constraint Store Management:

$$\langle A,\ C,\ S\rangle \rightarrow_i \langle A,\ C',\ S'\rangle$$
$$\text{if } (C',\ S') = infer(C,\ S)$$

Under this transition, new constraints are inferred from the previous ones in the store.

Consistency:

The consistency transition determines if the active constraints are consistent or satisfiable:

$$\langle A,\ C,\ S\rangle \rightarrow_s \langle A,\ C,\ S\rangle$$
$$\text{if } consistent(C)$$

or

$$\langle A,\ C,\ S,\rangle \rightarrow_s fail$$
$$\text{if } \neg consistent(C)$$

A CLP *derivation* is a sequence of transitions $\langle A_1,\ C_1,\ S_1\rangle \rightarrow \ldots \rightarrow \langle A_i,\ C_i,\ S_i\rangle \rightarrow \ldots$. If a state cannot be further rewritten, it is called a *final state*. A *successful* derivation is finite and has a final state of the form $\langle \emptyset,\ C,\ S\rangle$. If G is a query with free variables $x_1, \ldots x_n$, which causes a successful derivation and final state $\langle \emptyset,\ C,\ S\rangle$, then the existential closure of $C \wedge S$, except for the variables $x_1, \ldots x_n$, is said to be the *answer constraint* of the query. On the other hand, a derivation is *failed* if it is finite with final state *fail*. A derivation is said to *flounder* if it is finite and the final state is of the form $\langle A,\ C,\ S\rangle, A \neq \emptyset$.

The *computational tree* of a goal G in a CLP system for a program P is a tree whose nodes are computational states and whose edges are transitions. Those nodes whose outgoing edges are labelled with $\rightarrow_c$, $\rightarrow_i$ or $\rightarrow_s$ transitions have exactly one child; those with an outgoing edge $\rightarrow_r$ have as many children as there are rules in P. The root has a state label $\langle G,\ \emptyset,\ \emptyset\rangle$. Every branch of the tree constitutes a separate derivation. The computational rule determines the tree for the query and program. The process of finding solutions to CLP queries is equivalent to searching a computational tree. Most CLP systems do so in a depth-first manner with simple backtracking.

2.4 Incrementality

We now present a few words about incrementality. Jaffar and Maher [28, p.532] distinguish between two notions of incrementality for algorithms. First, incrementality can be used to refer to the *nature* of an algorithm. In this case, an algorithm is said to be incremental if it accumulates an internal store of information and each new data item is processed in conjunction with the existing data. On the other hand, algorithms can be said to be incremental in terms of *performance*. Roughly speaking, an algorithm is incremental in this sense if the time it takes to process a new input item is proportional to the size of the data item and not the size of the existing data combined with the item. An algorithm is nonincremental if running it on a new input is no faster than running it on the entire data set accumulated so far. In general, there will be a continuum of incrementality in this sense among algorithms which perform a given task. Since it is important for CLP implementations to be practical, it is desirable for CLP algorithms to be as incremental as possible in the second sense. Jaffar and Maher present a more formal description of this version of incrementality for CLP operations in [28, pp.532-533].

2.5 Operations

There are four tests and operations on constraints of critical importance in CLP languages [28, pp.513-514]. They are: a test for satisfiability, constraint entailment, projection and the detection of the grounding of variables by constraints. The first is the most important, and is needed in all languages, while the others may not be fully supported by some languages. We now briefly describe these operations and some characteristics of the algorithms which perform them.

Testing the constraint store for *satisfiability*[7] is the main job of the constraint solver. Specifically, given a set of constraints, a set of variables and their domains the problem is to determine if there exists an assignment to the variables that satisfies all the constraints. For most constraint domains this problem is NP-complete. Therefore, the efficiency of a satisfiability algorithm for a given constraint domain is most often assessed based on its average, and not worst case, behaviour. It is also crucially important for satisfiability algorithms to be incremental in the second sense we have discussed.

Constraint entailment is the process of determining what subset of a set of guard constraints is entailed by the conjunction of a satisfiable set of constraints and an additional constraint. More formally, given a satisfiable constraint set C, a set of guard constraints G such that no $g \in G$ is entailed by C, and a new constraint c, the problem is to find the subset G' of G entailed by $C \wedge c$. Once again, algorithms which detect entailment must be incremental in nature. In particular, there should be no reexamination of the complete set of guard constraints each time a constraint is added to the store.

The *projection* operation consists of finding the projection of a set of constrains

[7]In the CLP literature, the terms *consistency* and *satisfiability* are often used interchangeably. We make a distinction between the two in the context of the CSPs of CLP($\mathcal{FD}$) in Section 2.6.1.

with respect to *target variables* and expressing it in a usable symbolic form. This involves finding the existential closure [35, p.7] of the constraint set except for the non-target variables. For example, the projection of the constraints $s = u - 5$ and $t = u + 2$ with respect to s and t is $s = t - 7$. Note that the non-target variables cannot be involved in the projection expression. The projection operation is the basis for removing variables from the constraint store which will not be referred to again. Since they are closely related to the domain on which they operate, projection algorithms have little in common across different constraint domains.

The final important CLP operation determines that, given a variable and a constraint, there is only one value in the domain of the variable that satisfies the constraint. If this is the case, then the variable is said to be *grounded* or *determined* by the constraint. Note that detecting groundness to a *specific* value actually constitutes a constraint entailment problem.

For a complete description of algorithms for these and other CLP operations for various constraint domains, see [28, pp.534-545].

2.6 Constraint Logic Programming for Finite Domains

In this section, we describe the operation of Constraint Logic Programming for the finite domain constraint domain (CLP($\mathcal{FD}$)). CLP($\mathcal{FD}$) languages have predicates that allow users to create programs that express combinatorial search problems. The usual approach taken by CLP($\mathcal{FD}$) languages to solve these problems is to incorporate techniques developed for solving CSPs and CSOPs into the SLD resolution format. In this approach, *general* techniques for solving CSPs are used. Techniques also exist for solving *specific* CSPs. For example, see [48] for a description of how the well known job shop scheduling problem can be modelled and solved as a CSP.

One possible approach to solving combinatorial search problems is to search through the set of all elements of the Cartesian product of the domains of all variables for those which satisfy the conjunction of constraints that define the problem. This technique is known as *generate and test*. Unfortunately, for a sufficiently large problem this brute force approach can become infeasible. In particular, applying this technique to the cryptogram described in Section 1.2 would involve testing 10^8 8-tuples for satisfiability of the problem constraints. More generally, discrete combinatorial problems usually have intractable complexity. CLP($\mathcal{FD}$) languages therefore employ heuristic techniques developed for solving CSPs so that these problems may be solved efficiently in some cases. These heuristics come in two categories: consistency techniques and search strategies.

Consistency techniques are used to reduce the search space of discrete combinatorial problems. The basic idea is to remove inconsistent values from the domains of variables by analyzing the problem constraints. In this way, the constraints of the problem can be used to prune its search space in a dynamic manner. This approach is known as *constrain and generate*. van Hentenryck was the first to incorporate this technique in a CLP language [13, 53]. Search techniques are used to find the solutions to CSPs one at a time. Backtracking (as in Logic Programming) is the basic search technique employed in CSP problem solving. In the CSP context, it is a depth first

search of possible variable labellings. The search strategies used to solve CSPs make use of problem information to limit backtracking and the testing of inconsistent variable labellings. These strategies include, but are not limited to, lookahead, variable ordering and value ordering heuristics.

In general, there will be a tradeoff between the complexity of the consistency technique used and the resulting reduction in the complexity of the required search. On the one hand, there is the pure generate and test search algorithm used without consistency techniques. At the other extreme, there is Freuder's solution synthesis algorithm [17], which finds all solutions to a CSP at once using only consistency techniques and no searching. Consistency techniques and search strategies are in fact *complementary* and can be used together to solve CSPs. The challenge of CSP and by extension CLP($\mathcal{FD}$) research is to find clever ways of combining consistency techniques and search heuristics with a view to finding *efficient* methods of solving finite domain problems. In particular, a premium has been established for finding classes of problems that are amenable to backtrack free search and techniques which allow it.

In this section, we first describe consistency techniques. Search strategies for solving CSPs, including some probablistic methods, are described. Next, we show how the cryptogram in Section 1.2 can be done in CLP($\mathcal{FD}$). We conclude this section with some remarks about complexity issues as they pertain to the consistency and search algorithms we present.

2.6.1 Consistency Techniques

Consistency techniques reduce the complexity of CSPs and in the process make them easier to solve. They detect redundant values of variables *i.e.*, those which cannot be part of any solution to the CSP. Given a CSP with $X = \{X_1, \ldots, X_n\}$, a partial labelling of variables $(X_1 = x_1, \ldots, X_i = x_i)$ is *locally consistent* [10] if it satisfies all the constraints in the subproblem restricted to the set $\{X_1, \ldots, X_i\}$. Consistency techniques can be thought of as partial constraint solvers since they detect locally inconsistent variable labellings. These labels can be removed from the domains of the variables in question. In addition, any constraint tuple which contains such a label can be removed from the CSP. The resulting problem will be equivalent to the original in the sense that it will have the same variables and the same set of solutions. If the application of a consistency technique reduces the domain of any variable or constraint to the empty set, then the CSP is unsatisfiable.

Waltz was a pioneer of consistency techniques. He used them in scene analysis to eliminate impossible labellings of junctions in line drawings [57, 58]. Mackworth [36] showed the techniques have more general applicability. The three basic notions of consistency are node consistency, arc consistency and path consistency. The names are derived from the constraint hypergraph representation of CSPs discussed in Section 1.3. As stated, the hypergraph representation devolves to a normal graph for binary CSPs. Since most of the algorithms for attaining these levels of consistency operate on binary CSPs, we define them in terms of binary CSPs. (The notions of hyperarc and hyperpath consistency have been conceived for general CSPs [8]. In addition,

an algorithm, GAC4 [43] has been devised that enforces arc consistency for general CSPs.)

The simplest concept of consistency is that of node consistency. A constraint network is *node consistent* [52, p.57] if and only if for all variables the values in its domain satisfy the (unary) constraints on that variable. What is required for constraint network node consistency is formally stated below using the notation used to formalize finite CSPs found in Section 1.3:

$$\forall u \in X \ ((c_u \in C \ \wedge \ \sigma(c_u) = u) \Rightarrow \forall x \in \delta(u) \ \ x \in c_u)$$

The node consistency algorithm, NC-1 [36], is quite straightforward. The domain of each variable is checked against the unary constraints on that variable. Any inconsistent values are removed.

Arc consistency is more complicated. For an arc (u, v) in a constraint network to be consistent from variable u to variable v, for each value in the domain of u there must be a value in the domain of v such that (u, v) is a tuple in the constraint associated with the arc. The notion of arc consistency from variable u to variable v is formalized below.

$$(u, v \in X \ \wedge \ c_{uv} \in C \ \wedge \ \sigma(c_{uv}) = (u, v)) \Rightarrow (\forall x \in \delta(u) \ \exists y \in \delta(v) \ \mid \ (x, y) \in c_{uv})$$

A constraint network is *arc consistent* if every edge (u, v) in it is consistent both from u to v and from v to u. At this point we describe an interesting phenomenon of consistency techniques. When a locally inconsistent value is removed from the domain of a variable in a CSP, this may result in the necessity of removing values from the domains of the variables involved in the same constraints as the initial variable. Removal of these values can in turn cause other variable values to be removed and so on. This chain reaction of variable domain reductions is called *constraint propagation.* Its effect is most observable in the binary arc consistency algorithms. For this reason, these algorithms usually maintain a queue of variables whose domains may have to be reduced. Arc consistency algorithms go by the name AC-k where k is a natural number.

Algorithms AC-1, AC-2 and AC-3 are due to Mackworth [36]. AC-1 is quite inefficient since it checks the domains of all the variables *every time* a variable domain is reduced. AC-2 and AC-3 are more clever, since they check only the domains of those variables which might be affected by any given variable domain reduction. AC-4 [42] takes this idea one step further, with the result being an even more efficient algorithm. The key idea is to maintain support sets for each label of each variable. The *support set* of a label is the list of the values of each variable that are compatible with the label. The number of supporting labels for each possible ordered pair consisting of a constraint and a label is also maintained. When a label is deleted, the labels that are on its support list are examined. The number of supports of each of those labels is decremented. If the number becomes zero, the label must be removed from the domain of its variable. We also mention AC-5 which is due to van Hentenryck *et. al.* [11, 54]. This generic algorithm has given rise to a set of algorithms for various types of constraints. It has been shown to be especially efficient for the useful *functional* constraints.

The notion of path consistency expands the idea of arc consistency to paths in constraint graphs. A path in a constraint network is consistent if for every value in the domains of the variables at its end points, there exist values in the domains of the other variables in the path such that the sequence of arc constraints that comprises the path is satisfied. A constraint network is *path consistent* if every path in it is consistent in this sense. The algorithms which enforce path consistency have names of the form PC-k. It was once again Mackworth's useful 1977 paper which produced PC-1 and PC-2 [36]. PC-1 produces path consistency by reducing constraints by means of *relations composition* [52, p.91]. PC-2 improves upon its complexity by being more selective about re-examining constraints when a change to a constraint occurs. PC-4 [24], which corrects slight errors in PC-3 [42], uses the approach of maintaining support sets used by AC-4 to improve upon PC-2.

An alternative notion of consistency exists for CSPs, namely k-consistency. A constraint network is said to be *k-consistent* [37] iff given any labelling of any $k-1$ variables satisfying all the constraints involving those variables it is possible to find an instantiation of any kth variable such that the k values taken together satisfy the constraints involving the k variables. (It should be noted that according to this definition, for binary CSPs node consistency is equivalent to 1-consistency, arc consistency enforces 2-consistency and path consistency is 3-consistency.) Cooper has devised an algorithm [9] to enforce k-consistency in arbitrary constraint networks.

Unfortunately, k-consistency does not in general imply $(k-1)$-consistency. With this defect in mind, Freuder introduced a stronger notion of consistency. A constraint network is *strongly k-consistent* [18] if it is j-consistent for all $1 \leq j \leq k$. Closely related to the notion of consistency is that of satisfiability. A constraint network is k-satisfiable [52, p.54] if there exists a labelling of every k-subset of its variables that satisfies all the constraints involving those variables. A constraint network with n variables is satisfiable if it is n-satisfiable.

It is important to note that k-consistency does not necessarily imply CSP satisfiability if k is less than n. (Actually, k-consistency is neither a sufficient nor a necessary condition for problem satisfiability [52, p. 65].) For this, strong consistency is required. Strong k-consistency implies k-satisfiability. Therefore, proving strong n-consistency is sufficient to show problem satisfiability. However, as we shall see in Section 2.6.4, this can be extremely inefficient. Search strategies must therefore be used in conjunction with consistency techniques to prove CSP satisfiability and to find solutions to CSPs. We discuss these strategies next. A final observation of this section is that consistency techniques are considered to be constraint store management transitions as described in Section 2.3.

2.6.2 Search Strategies

A great deal of research has been done on finding efficient search techniques for solving CSPs, perhaps more than any other area of CSP research. Since a full discussion of this material is beyond the scope of this paper, we restrict ourselves to the most important CSP search techniques. For more information on this topic, the interested reader should consult [52, pp. 119-188]. Three basic search strategies used to solve

CSPs are simple backtracking, forward checking and lookahead. We discuss these first.

Simple backtracking (or *chronological backtracking* as it is also called) maintains the invariant that the variables currently instantiated are locally consistent. If the most recent variable instantiation has resulted in a locally inconsistent labelling, then the next value in its domain is tried. If there are no more possible values in the domain of the variable, then the previously instantiated variable is labelled with its next possible value.[8]

In addition to the restriction of simple backtracking, *forward checking* ensures that each unlabelled variable has at least one value in its domain that is consistent with the labels of the variables that have been instantiated. If a variable is instantiated with a label and there is an unlabelled variable with no possible consistent label, a new label is tried. If all possible labels have been exhausted, a new label is found for the previously labelled variable.

Lookahead techniques are even more stringent. In addition to the requirements of forward checking, these algorithms stipulate that there must be a labelling of each unlabelled variable that is consistent with the other unlabelled variables. For example, in the AC-lookahead algorithm [25] when a variable is labelled, all the labels that are inconsistent with the labellings made so far are removed from the domains of the unlabelled variables. Moreover, arc consistency is maintained among the unlabelled variables. If at any time the domain of an unlabelled variable becomes empty, the algorithm backtracks in the usual manner, by first attempting to relabel the current variable and then previously labelled variables if necessary. In a related paper, Freuder and Sabin [19] advocate maintaining full arc consistency at all times during a search.

Dependency directed backtracking or *intelligent backtracking* [5] is a general search strategy which attempts to choose the points to which backtracking occurs cleverly. The idea is to find the variable labellings that are most responsible for the need to backtrack. An example of this technique is *backjumping.* Under this strategy, whenever there is no domain value for a candidate variable that is consistent with the previously labelled variables, the algorithm does not backtrack to the most recently labelled variable. Instead, for each element of the domain of the candidate variable, the 'culprit' variable is identified. The culprit is the first labelled variable that is inconsistent with the domain value in question. The algorithm then backtracks to the most recent culprit variable and relabels it.

By strategically choosing the order in which variables are labelled, the efficiency of a CSP search can be increased. Many techniques exist for making this choice. They are called *variable ordering heuristics.* An example is the *minimal width ordering* heuristic [18]. Under this heuristic, the ordering of the CSP variables is found such that the width of the primal graph of the problem is minimized. The variables are instantiated according to this ordering. Another effective strategy is based on the notion that in order to reduce backtracking resulting from dead end searching, it is prudent to start with the variable whose labelling is most likely to fail. This general

[8] Note that this restriction is not made in the näive generate and test procedure, which searches all possible complete labellings in a depth first manner regardless of any intermediate locally inconsistent labellings.

search strategy is called the *fail first principle* or FFP. In particular, the technique of labelling the variables in increasing order according to the size of their domain has met with success. This technique can be used statically before a search or dynamically during, for example, a lookahead search.

Moreover, when labelling a variable it may be advantageous to try some of the values in its domain before others. Heuristics which decide the order in which the values of variables are selected as potential labels are known as *value ordering heuristics*. In contrast with variable ordering heuristics, value ordering heuristics attempt to find the values most likely to succeed so that backtracking will not be needed. The *min-conflict* heuristic [41] is an example. Under this heuristic, the values of a variable are ordered according to the degree of conflict they have with the unlabelled variables.

Probablistic algorithms also exist for solving CSPs. They can be faster than deterministic methods. Unfortunately, they suffer from the drawback of being incomplete, *i.e.*, they are not guaranteed to find all solutions to every CSP. Hill climbing [52, pp. 254-261] is the best known of these techniques. The *heuristic repair method* [41] is considered to be a hill climbing algorithm. *Simulated annealing* [1] is another, relatively unresearched, version of probablistic search. The well known *branch and bound search technique* (see for example [2]) is frequently used to search CSOPs.

It is important to note that the general heuristics mentioned in this section will not work well for every problem instance. The reader is reminded that part of the essence of the NP-completeness of the finite CSP is that a problem instance can be found for every heuristic search technique that it will not solve efficiently. Determining which techniques are most effective for different types of CSPs is an important research topic. For a discussion of how problem specific features can be exploited to solve CSPs, see [52, pp. 189-251]. Research also has been done which attempts to determine effective combinations of the search techniques described here for various types of CSPs [52, pp.180-184]. As stated, consistency techniques are often used dynamically with the search heuristics. In some cases this is not a good strategy, however. For example, intelligent backtracking searches are ineffective when used in conjunction with consistency techniques. A last remark is that labellings of variables are considered to be resolution transitions as described in Section 2.3.

2.6.3 The Example Done in CLP($\mathcal{FD}$)

We now demonstrate the operation of CLP($\mathcal{FD}$) on the cryptogram problem of Section 1.2. A CLP($\mathcal{FD}$) program for solving the cryptogram is given in Figure 3. It has been reproduced from [53], in which van Hentenryck proposed the use of consistency techniques in the Logic Programming paradigm.[9] A commercial system named CHIP [13] was later developed based on this notion. It has achieved some success using the domain size variable ordering heuristic described in Section 2.6.2. For this reason, we

[9] We have removed the constraints $S \neq 0$ and $M \neq 0$ to make this example match the one given in Section 3.2.5 that demonstrates the operation of ABL for finite domains.

```
domain sendmory(0..9,0..1).
sendmory([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]) ←
    alldifferent([S,E,N,D,M,O,R,Y]),
    R1 = M,
    R2 + S + M = O + 10 × R1,
    R3 + E + O = N + 10 × R2,
    R4 + N + R = E + 10 × R3,
    D + E = Y + 10 × R4,
    labelling([R1,R2,R3,R4]),
    labelling([S,E,N,D,M,O,R,Y]).
```

Figure 3: A CLP($\mathcal{FD}$) Program for Solving the Cryptogram Problem

have chosen to describe at a high level[10] the operation of a CLP($\mathcal{FD}$) system which uses the domain size heuristic and simple backtracking on the cryptogram. The value ordering employed is simply to instantiate each variable in increasing order starting with the smallest value in its domain.

Given the goal

$S > 6$, $D < 6$, sendmory([S,E,N,D,M,O,1,7],[R1,R2,R3,R4])

top-down execution proceeds as follows. The query automatically instantiates R to be 1 and Y to be 7. The next smallest domain is that of S, which is initially set to be 7. However, this is in violation of the `alldifferent` constraint since $Y = 7$. S is therefore reset to 8. D is then set to 0. Next, the system arbitrarily attempts to instantiate E. The only value of E that is consistent with the last constraint of the program $E + D = Y + 10 \times R4$ is 7. But this once more violates the `alldifferent` constraint. Therefore, the system must backtrack and relabel D. Since $R = 1$, D clearly cannot be 1, and so the assignment $D = 2$ occurs.

By the last constraint, $E = 5$ and R4 is set to 0. The next variable chosen to be instantiated is N. On the one hand, the assignments $N = 1$ and $N = 2$ are not distinct from assignments to non-carry digits already labelled; on the other hand, the labellings $N = 0$ and $N = 3$ violate the second to last constraint, namely $R4 + N + R = E + 10 \times R3$, since $E = 5$, $R4 = 0$ and $R = 1$. Therefore the system lets $N = 4$ and R3 is set to 0.

Because $R3 = 0$, $E = 5$ and $N = 4$, the only labelling of O consistent with the constraint $R3 + E + O = N + 10 \times R2$ is $O = 9$, if $R2 = 1$. From the third and fourth constraints the system then concludes that $M = R1 = 0$ and the solution (8, 5, 4, 2, 0, 9, 1, 7, 0, 1, 0, 0) has been found. The other solution (8, 3, 2, 4, 0, 9, 1, 7, 0, 1, 0, 0) is then found by additional backtracking.

[10] Note that it is possible, but well beyond the scope of this paper, to describe the operation of a CLP system on the cryptogram problem using the computational states and state transitions given in Section 2.3.

The interested reader is encouraged to examine van Hentenryck's alternate description [53, pp.141-144] of how the cryptogram problem can be done in CLP($\mathcal{FD}$).

2.6.4 Complexity Issues

In this section, we discuss the complexity of some of the algorithms we have mentioned. Note that n refers to the number of variables in the CSP, e to the number of binary constraints and a is the size of the largest variable domain. The node complexity algorithm NC-1 is $O(an)$. The time complexities of the arc consistency algorithms are usually proportional to the product of the number of binary constraints and a polynomial function of the largest variable domain in the problem. In particular, the complexity of AC-1 is $O(a^3ne)$; that of AC-4 is $O(a^2e)$. The time complexity of AC-5 for functional constraints is $O(ea)$. PC-4 has time complexity $O(a^3n^3)$. For a full analysis of the complexity of the node, arc and path consistency algorithms, see [38, 39]. The complexity of Cooper's k-consistency algorithm [9] is

$$O(\sum_{i=1}^{k} \left(\begin{array}{c} n \\ i \end{array} \right) a^i)$$

The time complexity of Freuder's solution synthesis algorithm is also prohibitive at $O(2^n + na^{2n})$. The worst case complexity of any search algorithm used without consistency techniques where full backtracking is required is $O(a^n)$ whereas that of backtrack free search is $O(an)$.

3 Array Based Logic

Array Based Logic is an approach to logical reasoning that has been investigated at the Technical University of Denmark. Ole Franksen pioneered the approach for reasoning with boolean logic systems [15, 16]. Recently, Gert Møller has generalized Franksen's work for arbitrary variable and constraint domains [44]. The research has been conducted from an engineering point of view, with the goal of simulating real-time industrial and physical systems in mind. Indeed, a commercial software product based on the technology called *Beologic* [4] has been developed and marketed by the Danish company Bang & Olufsen for use in the control of real-time systems. ABL has been applied to a scheduling problem in production planning [45] and to the design of a train routing system for the Danish state railway DSB [33, 34]. Another current research project is investigating the use of ABL to control the Sydkraft power system of Malmo, Sweden [44, p. 1].

In what follows we first present an overview of Franksen's approach to ABL. We then describe how the work of Møller has generalized ABL for other variable domains. The operation of ABL for finite domains is then demonstrated using the cryptogram example from Section 1.2.

A	B	C	D	$A \wedge ((B \Rightarrow C) \vee D)$
T	T	T	T	T
T	T	T	F	T
T	T	F	T	T
T	T	F	F	F
T	F	T	T	T
T	F	T	F	T
T	F	F	T	T
T	F	F	F	T
F	T	T	T	F
F	T	T	F	F
F	T	F	T	F
F	T	F	F	F
F	F	T	T	F
F	F	T	F	F
F	F	F	T	F
F	F	F	F	F

Figure 4: A Truth Table Representation of $A \wedge ((B \Rightarrow C) \vee D)$

3.1 Franksen's Approach to Array Based Logic

In this section, we explain the most important concepts of Franksen's approach, namely those needed to understand the description of Møller's work which follows. In particular, we describe the basic concepts and operations of ABL as well as the model incorporated in the technology. For a more detailed overview of Franksen's work in boolean ABL, see Chapter 1 of [45].[11]

3.1.1 Basic Concepts

Franksen takes a systemic approach to logical reasoning on the boolean constraint domain $\mathcal{BD}$. The fundamental idea behind his approach is to treat the set of possible truth values of a conjunction of propositional logic formulae as a system which can be manipulated and analyzed. Traditionally, the set of 2^N possible truth values of a propositional logic formula with N variables has been stored in truth table format. For example, the truth table for the formula $A \wedge ((B \Rightarrow C) \vee D)$ for boolean variables A, B, C and D is given in Figure 4.

In ABL, the truth values of a logic formula with N variables (or those of a conjunction of logic formulae) are stored in an N dimensional array with each dimension or axis in the array being labelled with the possible values (false and true) of one of the N variables.[12] A truth value is stored in the array for each element of the

[11] Michael Jenkins has implemented heuristics [30] for building array based boolean logic systems based on Franksen's work in the NIAL [31] programming language.

[12] In general, there may not be a one to one correspondence between variables and array axes. In

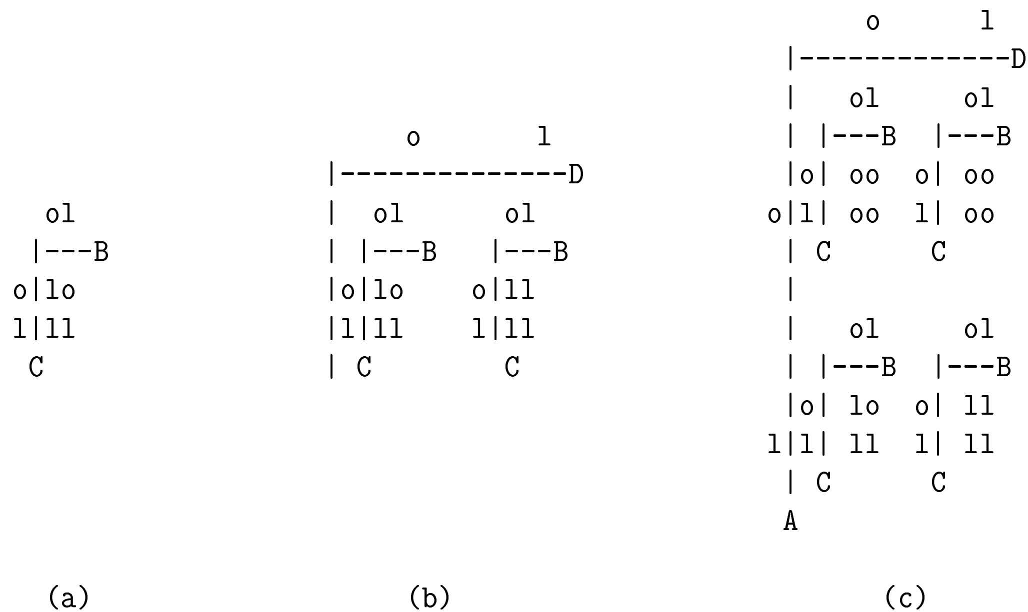

Figure 5: Truth Value Array Representations

Cartesian product of the domains of the variables involved in the set of formulae. If the element satisfies the formulae, then true is stored. Otherwise false is stored. The resulting array contains 2^N truth values.

The concept of the truth value array representation is illustrated in Figure 5. A 2×2 truth value array for the expression $B \Rightarrow C$ is given in (a). The truth value array for $(B \Rightarrow C) \vee D$ is shown in (b). The $2 \times 2 \times 2 \times 2$ truth value array for the expression $A \wedge ((B \Rightarrow C) \vee D)$ is shown in (c). In the figure, each true value is represented by an l, each false one by an o. The axes are ordered A, D, C, B and are laid out alternately horizontally and vertically starting with the last axis.

3.1.2 Operations

Truth value arrays must be created, combined and analyzed. The three operations that do so are *outer product*, *colligation* and *projection*. Outer product [31, p. 82] is an array transformer [31, pp. 143-150] which takes two lists of elements and a binary operation and applies the operation to each element of the Cartesian product of the lists. Outer product is used to build up the truth value arrays of propositional logic formulae.

Colligation (or *generalized transposition* as it sometimes called) fuses axes together that are labelled with the same variable. Geometrically, this corresponds to taking out diagonal hyperplanes from the truth value array, and is done to ensure that axes corresponding to a repeated variable assign only equal values to the variable.

---

particular, there may be more than one axis labelled with the same variable. If so, the *colligation* operation may be used to eliminate the extra axes. See Section 3.1.2.

There are two kinds of projection, *abductive* projection and *deductive* projection. They are mutually derivable. Deductive projection involves taking the logical *or* over a set of truth values while abductive projection involves finding the logical *and* of the values. Deductive projection can be used to eliminate axes in the system which correspond to variables which are irrelevant in an analysis of the system. This process is called *variable elimination*.

3.1.3 The Model

The goal of ABL is to build a model of a physical system which, once built, can be repeatedly used to deduce the values of output variables based on values of input variables. The model is based on the Walrasian economic model [56]. U, the *universe of discourse*, consists of the global domain (also called the environment) and the system and their interaction. The *system* S is a set of relations. The *global domain* is a set of N variables V = $\{v_i \mid 1 \leq i \leq N\}$. Associated with each variable is a domain of the values which the variable can attain. The set of all such variable domains is D = $\{d_i \mid i = 1..N\}$. Formally, the global domain GD is the ordered set consisting of all ordered pairs $(v_\mathrm{i}, d_\mathrm{i})$, i = 1 .. N such that $v_i \in V$ and $d_i \in D$.[13]

Any variable that has been instantiated with a non-empty subset of its domain is said to be an *external influence* on the system. The variables are related to each other by means of a set of *constraints* C which defines the behaviour of the physical system. Conceptually speaking, a *relation*[14] or *isolated element* R_k is a triple (C_k, V_k, TV_k) consisting of a set of constraints $C_k \subseteq C$, and the set of variables $V_k \subseteq V$ related by the constraints and a truth value mapping TV_k which maps a truth value to each element of the Cartesian product of D_k, the variable domain subset of D corresponding to V_k, under the constraints C_k.[15]

Initially, the constraints are usually distributed among relations such that each constraint is the only element of the constraint set of exactly one relation. The resulting set of relations S_{prim} is known as the *primitive system*. The connections between common variables among relations define additional constraints. These are called *connectivity constraints*. The connectivity constraints of a system define *the topology of interconnected relations*, also known as the *system structure*. The system structure may be represented by an undirected multigraph $G_S = (S_{prim}, E)$ such that an arc (i, j) with label v is in E if the isolated relations R_i and R_j share a common variable v. This multigraph is known as the *colligation graph* of the system. The process of removing connectivity constraints is known as *constraint elimination*. Creating the system is also known as *system modelling*. The process of deducing the values of output variables based on external influences on the system is called *system simulation*.

---

[13]In general, an environment variable may be an input or an output variable.

[14]Note that in ABL the use of the term *relation* differs from its traditional mathematical use. In ABL, we use the term to refer to the data structure used to represent a mathematical relation as a set of tuples.

[15]Note that while conceptually relations are triples, they are represented in ABL as pairs. See Section 3.2.1.

3.2 Møller's Extension of Array Based Logic

Møller's generalization of ABL to arbitrary variable and constraint domains is the topic of this section. (The reader should note that although some research has been done on the representation and manipulation of other variable and constraint domains in ABL, in this paper we restrict our attention to ABL for finite domains. For an introduction to array based constraint reasoning systems over other domains, see Chapter 3 of [44].) Møller's approach is to create relations using the tuples that satisfy the constraints of the search problem. (Note that the tuples are called constraints in the terminology of Section 1.3.) The resulting relations constitute a Finite CSP. The relations are then combined to create the system model.

We begin this section with a discussion of how these relations are represented in Møller's work. Next comes a description of how relations can be joined together. An explanation of the method for creating the model of the system is presented next. The simulation of the system is then discussed. Next, a detailed example is presented. Lastly, the complexity of the ABL finite domain operations is discussed.[16]

3.2.1 Representation of Relations

It is desirable to have a compact representation of relations. Clearly the truth values of its set of constraints must be stored. The list of variables related by its constraints will also be maintained. However, the constraints themselves will only be implicitly represented. The representation of the set of variables of the relation is straightforward: the variables are stored in a list in their global domain order. It is more complicated, however, to store the set of truth value mappings of the relation.

There are five isomorphic representations for the set of truth values of a relation, two of which are commonly used in practice. The simplest is the *binary array representation*. This is the representation discussed for boolean variables in Section 3.1, above. For finite domain relations, each axis of the array is labelled with the domain of one of the variables present in the set of constraints. Since the domains of the variables can have an arbitrary number of elements, the axes of the array can be of any size.[17] The array contains the truth value of the conjunction of constraints for each element of the Cartesian product of the variables. Hence, for a set of constraints over N variables each with domain size M, an N dimensional array with M^N values is required. For example, consider the global domain

$$GD = \{(\mathrm{X}, \{0, 1, 2\}), (\mathrm{Y}, \{4, 5, 6\}), (\mathrm{Z}, \{4, 5, 9\})\}$$

and the constraint $\mathrm{X} + \mathrm{Y} \leq \mathrm{Z}$. The truth table representation of this expression is given in Figure 6.

The corresponding $3 \times 3 \times 3$ binary array representation is shown in Figure 7.

In practice, a more compact method of representation is required since for large values of M and N the manipulation and storage of such a table is not feasible. This

[16] Jenkins and the author have created NIAL implementations [30] of the compression and join techniques for finite domain ABL relations described by Møller in Chapter 2 of [44].

[17] This is in contrast to boolean ABL binary arrays, whose axes all are of size 2.

X	Y	Z	$X + Y \leq Z$
0	4	4	T
0	4	5	T
0	4	9	T
0	5	4	F
0	5	5	T
0	5	9	T
0	6	4	F
0	6	5	F
0	6	9	T
1	4	4	F
1	4	5	T
1	4	9	T
1	5	4	F
1	5	5	F
1	5	9	T
1	6	4	F
1	6	5	F
1	6	9	T
2	4	4	F
2	4	5	F
2	4	9	T
2	5	4	F
2	5	5	F
2	5	9	T
2	6	4	F
2	6	5	F
2	6	9	T

Figure 6: A Truth Table Representation of $X + Y \leq Z$

```
     4      5      9
|---------------------Z
| | 012    012    012
| |------X|-----X|----X
|4| loo  4|llo  4|lll
|5| ooo  5|loo  5|lll
|6| ooo  6|ooo  6|lll
  Y       Y      Y
```

Figure 7: A Binary Array Representation of $X + Y \leq Z$

```
0 4 4
0 4 5
0 5 5
1 4 5
0 4 9
0 5 9
0 6 9
1 4 9
1 5 9
1 6 9
2 4 9
2 5 9
2 6 9
```

Figure 8: List of True Tuples in Array Form

0 0 0 1 0 0 0 1 1 1 2 2 2	4 4 5 4 4 5 6 4 5 6 4 5 6	4 4 5 5 5 9 9 9 9 9 9 9 9
X	Y	Z

Figure 9: The Affirmative Form

can be achieved by only storing the array indices of the true values in the binary array for each variable. Known as the *affirmative form*, it is effective for many problems since for a sufficiently large set of constraints, the number of possible satisfying true values is usually relatively small. For the binary representation given in Figure 7, the subset of X × Y × Z of true tuples is

$$\{(0,4,4),(0,4,5),(0,5,5),(1,4,5),(0,4,9),(0,5,9),\\(0,6,9),(1,4,9),(1,5,9),(1,6,9),(2,4,9),(2,5,9),(2,6,9)\}.$$

This list is given in array form in Figure 8. The corresponding affirmative form is shown in Figure 9. The array indices for each variable are listed in the order of the tuples above. Note that a corresponding representation exists for storing the false values of a binary array representation. It is called the *negative form* [44, p.32]. Any representation which makes use of tuples to store the truth value mapping is said to be in *developed form* [44, p.52]. Note that the affirmative form is best for performing deductive reasoning, whereas the negative form is better for abductive reasoning. In this paper we concentrate on the use of ABL to perform the former. For a description of how abductive reasoning is done in ABL, see [44, pp. 80-84].

A still more compact representation is possible for the developed form in some cases. The idea is to store lists of Cartesian arguments corresponding to subspaces in the binary array all of whose elements are true. The elements of the Cartesian product

```
+---------------------------------------+-----+
|+-------------+-----------+---------+|X Y Z|
||+-----+---+-+-+|+-----+-+-+-+|+-+-+-+-+||     |
|||0 1 2|0 1|0|0|||4 5 6|4|4|5|||9|5|4|5|||     |
||+-----+---+-+-+|+-----+-+-+-+|+-+-+-+-+||     |
|+-------------+-----------+---------+|     |
+---------------------------------------+-----+
```

Figure 10: Standard Array Form of the Relation $X + Y \leq Z$

of a list of Cartesian arguments are tuples that have true values in the original array. For example, in the binary array shown in Figure 7, the subspace

$$\{(x, y, z) \mid x \in \{0, 1, 2\}, y \in \{4, 5, 6\}, z \in \{9\}\}$$

consists completely of true values. The list of Cartesian arguments corresponding to that subspace is [[0, 1, 2], [4, 5, 6], [9]]. A representation of truth values incorporating lists of Cartesian arguments is said to be in *undeveloped form*. The process of converting a relation from developed to undeveloped form is called *compression*. The conversion of relations from undeveloped form to developed form is known as *expansion*.

An heuristic compression algorithm has been devised [44, p. 62] which converts a developed set of tuples to undeveloped form in polynomial time. An alternative compression heuristic exists [44, p. 70] which attempts to find the undeveloped representation for a list of tuples with the simplest (as opposed to the fewest) lists of Cartesian arguments. (For a full description of compression strategies, see [44, pp. 62-70].)

In ABL for finite domains, a relation is represented as a pair (a 1-dimensional array with two items) composed of one of the last four truth value representations of the five described above (affirmative undeveloped and developed, and negative undeveloped and developed) and the list of the variables involved in the constraints of the relation. Since deductive reasoning is most commonly performed in ABL, a relation is usually represented by an affirmative form of its truth values and its variable list. This representation is called the *standard array form* of the relation. The standard array undeveloped representation of the relation described in this section is

$$[[[[0,1,2],[0,1],[0],[0]],[[4,5,6],[4],[4],[5]],[[9],[5],[4],[5]]],[X,Y,Z]]$$

This is given in array form in Figure 10. A clearer representation of the same relation in standard form is given in Figure 11. In the figure, the NIAL operations `mix` and `pack` have been applied to the first element of the array.

3.2.2 Joining Relations

The colligation and outer product operations discussed in Section 3.1 are unified in the operation of joining relations. The idea of joining two relations $R_k = (C_k, V_k, TV_k)$

```
+--------------+-----+
|+-----+-----+-+|X Y Z|
||0 1 2|4 5 6|9||     |
|+-----+-----+-+|     |
||0 1  |4    |5||     |
|+-----+-----+-+|     |
||0    |4    |4||     |
|+-----+-----+-+|     |
||0    |5    |5||     |
|+-----+-----+-+|     |
+--------------+-----+
```

Figure 11: Alternative Format of the Standard Array Form of the Relation X+Y $\leq$ Z

and $R_l = (C_l, V_l, TV_l)$ is to produce a relation $R_{kl} = (C_k \cup C_l, V_k \cup V_l, TV_{kl})$ where TV_{kl} is the largest subset of the Cartesian product of the domains of the variable set $V_k \cup V_l$ satisfying the constraint set $C_k \cup C_l$. Note that ideally any algorithm that does so should be incremental in the second sense discussed in Section 2.4. An obvious strategy is therefore to use TV_k and TV_l to construct TV_{kl} rather than building it from scratch. Møller has created a procedure for joining two relations in affirmative developed or affirmative undeveloped form that uses this strategy [44, pp. 71-78].

This brute force technique proceeds as follows. For each variable common to the two relations, the outer intersection of all Cartesian arguments (or tuples) is taken. The elements of the outer intersections that are non-empty over all common variables are the legal subspaces of the joined relation. Using the legal subspaces, the arguments of the joined relation are extracted from the outer intersections of the common variables and the Cartesian product of uncommon variables. An interesting question is whether a search technique is more efficient to perform the join. See Section 4.3.)

3.2.3 Modelling the System

Recall that the objective of ABL is to create a system to perform real-time simulation of external influences on output variables. It is therefore necessary to produce a system such that the values of any output variable can be deduced based on a set of input values quickly. As stated in Section 3.1, intersections exist among the sets of variables of relations, namely the connectivity constraints. In general, the more connectivity constraints there are in the system, the more time consuming the simulation process will be. Also, as the number of relations increases, so too does the complexity of simulation. Since joining relations reduces the number of relations and performs connectivity constraint elimination, it would seem that it would be a good strategy to join relations in the system as much as possible. However, in many cases the size of relations grows dramatically as they are joined. There is therefore a tradeoff between the space complexity of a system and the complexity of performing simulation using

it. (A third issue of importance, especially for problems that grow dynamically is the amount of time needed to build the system.) The process of creating a system that effectively balances the first two factors is known as *building the system.* Møller has developed an heuristic procedure [44, pp. 118-141] for this task.

This operation proceeds as follows. Starting with the primitive system, pairs of relations are repeatedly joined. When, because of spatial considerations, it is infeasible to join any more relations, constraint elimination is executed by performing deductive projection searches on the disjoint variables of the relations. In particular, initially the system is set to be the primitive system, *i.e.*, $S = S_{prim}$.[18] New systems are repeatedly constructed in the following manner. For each pair of relations in the current system, the *connection factor* is determined. The connection factor CF_{ij} for two relations R_i and R_j is defined as

$$CF_{ij} = NC_{ij} * NC_{ji} * S_i * S_j$$

NC_{ij} is the number of noncommon or disjoint variables between relations R_i and R_j, *i.e.*, NC_{ij} is the number of variables in the variable set of R_i that are not in the variable set of R_j. S_i is the number of tuples in relation R_i.[19] For example, given the relations R_u and R_v such that

$R_u = (\{c_u\}, \{\mathrm{A, B, C, D, E}\}, \{t_0, t_1, \ldots, t_{22}\})$ and
$R_v = (\{c_v\}, \{\mathrm{D, E, F, G}\}, \{t_0, t_1, \ldots, t_{39}\})$

the connection factor $CF_{uv} = CF_{vu} = 3 * 2 * 23 * 40 = 5520$.

Let the smallest connection factor among pairs of relations in S be CF_{kl}. Relations R_k and R_l are than joined using a technique such as that described in Section 3.2.2. Let the resulting relation be R_{kl}. If the number of tuples in R_{kl} is less than a constant tolerance T, then a new system $S = S - R_k - R_l \bigcup R_{kl}$ is created.

The process is repeated until either S is a singleton or the number of tuples in the joined relation R_{kl} is greater than T. In the latter case, the values of the global domain variables not in each remaining relation are determined using deductive projection searches of the other remaining relations. The expanded representation which results from deductive projection searches of other relations is called the *complete array form* of the relation. If at any point in the above process the system contains a relation without tuples, then the process also halts. Under those circumstances, the system is said to be *inconsistent* and the corresponding problem has no solutions.

Note that constraint elimination in Møller's extension to ABL is in effect performed during the building of the system either by joining relations together or by extending them to the complete array form. The reader should also note that by building the system in the way described in this section, the entire set of solutions to a constraint based problem is generated at once.[20] The process of determining the

[18]See Section 3.1.3 for a definition of the primitive system.

[19]Observe that the connection factor is symmetric *i.e.*, $CF_{ij} = CF_{ji}$ for any two relations R_i and R_j.

[20]This is in sharp contrast to the CLP paradigm, under which solutions are found one at a time.

states (or values) of variables within the system under external influences is described next.

3.2.4 System Simulation

The interaction between the system and environment is captured by system simulation. As stated in Section 3.1, the determination of the values within the system of output variables based on external influences or instantiations of input variables is the basic operation of system simulation. This is accomplished by means of a process known as the *state vector transformation*. Before this can be described, some preliminary definitions are necessary. The *state* of a variable in a relation is the set of all values that it takes on within the tuples of the relation. The *state* of a variable in the system is the set of all values that it takes on in the system. The *system state* is a vector composed of the states of all the global domain variables in the system.

The idea of the state vector transformation is to find the subset of the tuples of a relation that is constrained by the external influences. The latter can be viewed as constraints on the domains of variables. The external influences to the system is an ordered set of sets that contains either the domain or a subset of the domain for every variable in the global domain. During the state vector transformation, a new relation is found that contains the subset of the tuples of the original relation such that every element in the tuples is contained in the set of the external influences corresponding to its variable. This is done by using outer intersection techniques similar to those used to join relations described in Section 3.2.2. The state of the new relation is now found using deductive projection techniques like those described in Section 3.1. The process of system simulation involves finding the system state vector based on an external influence vector. If the system is a single relation, the state vector transformation is applied to it as described above. If it is a collection of relations in complete array form, the transformation is applied to one of them.

3.2.5 The Example Done in ABL Finite Domain

We now illustrate the operation of ABL for finite domains using the sample cryptogram problem from Section 1.2. In our presentation of this example, we have followed Møller's approach to the example as described in his thesis [44, pp. 132-136].[21] As we shall see, in the example Møller creates a primitive system slightly different from that prescribed by the ABL model described in Section 3.1.3 where usually each constraint of the problem becomes the sole constraint of the constraint set of one relation of the primitive system.

In Møller's approach to solving the cryptogram in ABL, the primitive system consists of four relations. The constraint set of each relation contains the constraints which apply to one column of the problem. For example, the constraint set of the first relation contains the three constraints for the rightmost column of the problem: the arithmetic constraint for the non-carry digits E, D and Y, the arithmetic constraint

[21] The author has implemented the example in NIAL, whereas the original example is illustrated with APL [7] code.

for the carry digit R4, and the distinctness constraint among the three non-carry digits. Møller's approach leads to a formulation of the cryptogram problem with 14 constraints.

The arithmetic constraints for the non-carry digits for each column are given below in right to left order.

$$\mathrm{Y} = (\mathrm{D} + \mathrm{E}) \bmod 10 \quad (\mathrm{c}_0)$$

$$\mathrm{E} = (\mathrm{R4} + \mathrm{N} + \mathrm{R}) \bmod 10 \quad (\mathrm{c}_1)$$
$$\mathrm{N} = (\mathrm{R3} + \mathrm{E} + \mathrm{O}) \bmod 10 \quad (\mathrm{c}_2)$$
$$\mathrm{O} = (\mathrm{R2} + \mathrm{S} + \mathrm{M}) \bmod 10 \quad (\mathrm{c}_3)$$
$$\mathrm{M} = \mathrm{R1} \quad (\mathrm{c}_4)$$

Next we list the four constraints for the carry digits:

$$\mathrm{R4} = (\mathrm{D} + \mathrm{E}) \operatorname{div} 10 \quad (\mathrm{c}_5)$$

$$\mathrm{R3} = (\mathrm{R4} + \mathrm{N} + \mathrm{R}) \operatorname{div} 10 \quad (\mathrm{c}_6)$$
$$\mathrm{R2} = (\mathrm{R3} + \mathrm{E} + \mathrm{O}) \operatorname{div} 10 \quad (\mathrm{c}_7)$$
$$\mathrm{R1} = (\mathrm{R2} + \mathrm{S} + \mathrm{M}) \operatorname{div} 10 \quad (\mathrm{c}_8)$$

Although it is not included in the constraint set of one of the relations of the primitive system, the constraint which requires distinctness among all the non-carry variables still applies:

$$\mathrm{alldistinct}(\mathrm{S}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{M}, \mathrm{O}, \mathrm{R}, \mathrm{Y}) \quad (\mathrm{c}_9)$$

In addition, there are constraints to ensure that the non-carry digits are distinct within the four rightmost columns:[22]

$$\mathrm{alldistinct}(\mathrm{E}, \mathrm{D}, \mathrm{Y}) \quad (\mathrm{c}_{10})$$

$$\mathrm{alldistinct}(\mathrm{E}, \mathrm{N}, \mathrm{R}) \quad (\mathrm{c}_{11})$$
$$\mathrm{alldistinct}(\mathrm{E}, \mathrm{N}, \mathrm{O}) \quad (\mathrm{c}_{12})$$
$$\mathrm{alldistinct}(\mathrm{S}, \mathrm{M}, \mathrm{O}) \quad (\mathrm{c}_{13})$$

The global domain GD is

$$GD = \{(\mathrm{S}, D_{dig}), (\mathrm{E}, D_{dig}), (\mathrm{N}, D_{dig}), (\mathrm{D}, D_{dig}), (\mathrm{M}, D_{car}), (\mathrm{O}, D_{dig}), (\mathrm{R}, D_{dig}), (\mathrm{Y}, D_{dig}), (\mathrm{R1}, D_{car}), (\mathrm{R2}, D_{car}), (\mathrm{R3}, D_{car}), \mathrm{R4}, D_{car})\}.$$

[22] Given the presence of c_9, these constraints appear to be redundant. However, they are required to reduce the size of the relations of the primitive system. The alldistinct constraint is implemented by array operations since this is easily done.

where $D_{dig} = \{i \in \mathcal{N} \mid 0 \leq i \leq 9\}$ and $D_{car} = \{0, 1\}$.

The set C of constraints is

$$C = \{c_i \mid 0 \leq i \leq 13\}$$

as defined above.

The primitive system S_{prim} is $\{R_0, R_1, R_2, R_3\}$ where the initial relations are:

$$R_0 = (\{c_0, c_5, c_{10}\}, \{\mathrm{E, D, Y, R4}\}, \{t_0, t_1, \ldots, t_{71}\})$$

$$R_1 = (\{c_1, c_6, c_{11}\}, \{\mathrm{E, N, R, R3, R4}\}, \{t_0, t_1, \ldots, t_{143}\})$$
$$R_2 = (\{c_2, c_7, c_{12}\}, \{\mathrm{E, N, O, R2, R3}\}, \{t_0, t_1, \ldots, t_{143}\})$$
$$R_3 = (\{c_3, c_4, c_8, c_{13}\}, \{\mathrm{S, M, O, R1, R2}\}, \{t_0, t_1, \ldots, t_9\})$$

The sets of tuples for each relation can be generated all at once fairly easily using compositions of NIAL operations. For example, the 72 tuples of R_0 are derived by the NIAL code shown in Figure 12. First, the Cartesian product EDpairs of E and D is obtained by `cart D E`. The set Yvalues of possible sums that the variable Y can attain is the remainder of all possible sums of D and E *i.e.*, `EACH sum (cart D E) mod 10`. The set of possible values R4 that can be carried to the next column is the quotient of these sums, namely `EACH sum (cart E D) quotient 10`.

The next line of code groups the values of E, D and Y into a list of triples (allEDYtriples). Those triples having no duplicate values are next extracted. This is accomplished by applying the NIAL `diverse` operation to each triple. A boolean list, EDYmask, is the result. Each element of the list will be true if and only if the corresponding triple contains three different values. The `sublist` operation, which takes a boolean mask and a list as arguments and returns the elements of the list which correspond to true values in the mask, is then used to create the list of distinct triples, distinctEDYtriples. The corresponding values of R4 are then appended to create the set alltuples of tuples for the relation R_0.

The primitive system topology of the cryptogram is shown in Figure 13. The connection factors for the relation pairs are shown in Figure 14. Clearly, the best pair of relations to join is R_2 and R_3. The resulting relation R_{23} is

$$R_{23} = (\{c_2, c_3, c_4, c_7, c_8, c_{12}, c_{13}\}, \{\mathrm{S, E, N, M, O, R1, R2, R3}\}, \{t_0, t_1, \ldots, t_{77}\})$$

The system S is now altered to be $\{R_0, R_1, R_{23}\}$. The connection factors for S are shown in Figure 15. This time, it is best to join relations R_0 and R_1. The resulting relation R_{01} is $(\{c_0, c_1, c_5, c_6, c_{10}, c_{11}\}, \{\mathrm{E, N, D, R, Y, R3, R4}\}, \{t_0, t_1, \ldots, t_{519}\})$ The system becomes $S = \{R_{01}, R_{23}\}$. These final two relations are joined with the result being

$$R_{0123} = (\{c_0, c_1, c_2, c_3, c_5, c_6, c_7, c_8, c_{10}, c_{11}, c_{12}, c_{13}\},$$
$$\{\mathrm{S, E, N, D, M, O, R, Y, R1, R2, R3, R4}\}, \{t_0, t_1, \ldots, t_{297}\}).$$

```
E := tell 10;
D := tell 10;
EDpairs := cart E D;
Yvalues := EACH sum EDpairs mod 10;
R4values := EACH sum EDpairs quotient 10;
allEDYtriples := EDpairs EACHBOTH link Yvalues;
EDYmask := EACH diverse allEDYtriples;
distinctEDYtriples:= EDYmask sublist allEDYtriples;
distinctR4values := EDYmask sublist R4values;
alltuples := distinctEDYtriples EACHBOTH link distinctR4values;
```

Figure 12: NIAL Code for Generating R_0 Tuples

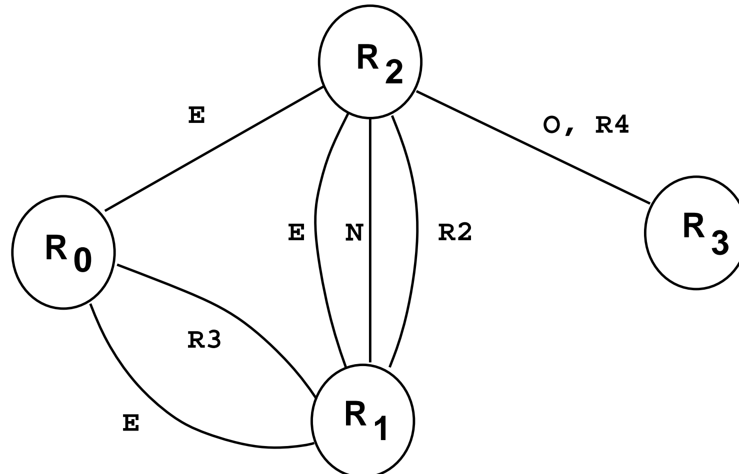

Figure 13: The Primitive System Structure of the Cryptogram Problem

	R_0	R_1	R_2	R_3
R_0	0	62208	124416	11520
R_1	62208	0	82944	28800
R_2	124416	82944	0	8640
R_3	11520	28800	8640	0

Figure 14: Connection Factors for the Primitive System

	R_0	R_1	R_{23}
R_0	0	62208	101088
R_1	62208	0	89856
R_{23}	101088	89856	0

Figure 15: Connection Factors for the Altered System

Unfortunately, the join operations have produced some tuples with identical values in the non-carry digits. One last application of the `EACH diverse` operation gives us the system $S = \{R_{fin}\}$ where

$$R_{fin} = (\{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\},$$
$$\{\mathrm{S, E, N, D, M, O, R, Y, R1, R2, R3, R4}\}, \{t_0, t_1, \ldots, t_{24}\}).$$

This contains the final solution of 25 tuples.

Consider the external influence vector sv_1 shown below. (The variable domains are listed in their global domain order.)

$\{\{7, 8, 9\}, D_{dig}, D_{dig}, \{0, 1, 2, 3, 4, 5\}, D_{dig}, D_{dig}, \{1\}, \{7\}, D_{car}, D_{car}, D_{car}, D_{car}\}$

When sv_1 is applied to the final system R_{final}, the resulting subrelation contains the two tuples (8, 3, 2, 4, 0, 9, 1, 7, 0, 1, 0, 0) and (8, 5, 4, 2, 0, 9, 1, 7, 0, 1, 0, 0).

We note at this point that although he presents no formal algorithm for creating the primitive system and building the system for CSPs in general, Møller's technique for doing the cryptogram example appears to differ from that described in the text of the thesis and used in other ABL examples found in it. In particular, there is usually a one to one correspondence between the constraints of the problem and the relations of the primitive system whose tuples satisfy them. The system is then built by combining those initial relations into ever larger ones using the join operation. Clearly this is not the case in the example since the constraint sets of the relations of the primitive system contain multiple relations and one constraint (c_9) is not a part of the primitive system and is enforced not by a join but rather by an APL operation at the end of the build process.

In presenting the example in this manner, Møller appears to have struck a balance between dedication to the techniques of ABL and pragmatism. On the one hand, he has followed to a large extent the general procedures outlined in [44] for the creation of the primitive system and building the system. On the other hand, he has done the example in such a way that *practical* implementations of it are possible. For example, he has avoided the creation of relations with a prohibitive number of tuples.[23] This is desirable since in addition to requiring a large amount of memory, large relations can be computationally expensive to join.

A last observation of this section is that many other procedures exist for doing the example in ABL. For example, the distinct tuples might be determined after each *intermediate* join that is executed. Alternatively, the alldistinct tuples for each column might be determined and subsequently joined with the tuples satisfying the arithmetic constraints for that column.

[23]In particular, the alldistinct relation for the 8 non-carry variables would have 1814400 tuples.

3.3 Complexity Issues

In general, the problem of finding the undeveloped representation for a binary array with the fewest lists of Cartesian arguments is NP-complete. (The problem is equivalent to that of partitioning a bipartite graph into a minimal set of complete bipartite subgraphs. The problem of partitioning a graph into a minimal number of cliques (shown to be NP-complete in [20, p. 193]), can be reduced to the latter problem.)

It must be emphasized that the compression operation is not guaranteed to find a smaller representation of the truth value mapping of all relations. It is merely an heuristic which in many cases will reduce the size of the relation. In the worst case scenario, no compression will be achieved. This entails that the space complexity of relations in ABL finite domain is still $O(M^N)$ for a relation with N variables with domain size M.

4 Conclusion

In this paper we have described two different paradigms for solving finite domain constraint satisfaction problems arising from different disciplines. In this concluding section we summarize the two approaches, discuss some interesting issues related to the two approaches and present some potential areas of research.

4.1 Summary

The two approaches to Constraint Based Reasoning discussed in this paper have been undertaken from distinctly different points of view. Constraint Logic Programming developed within Computer Science as a natural extension to Logic Programming for general domains. Finite Domain Constraint Logic Programming Languages have primitives which allow users to express combinatorial search problems and pose queries regarding them. These queries are solved using search and consistency techniques borrowed from CSP research which have been incorporated into the resolution scheme of Logic Programming. Array Based Logic, on the other hand, was created by Engineers with the objective of modelling large scale physical systems as Constraint Satisfaction Problems. A system of all the tuples which are solutions to the problem must be built in a bottom-up fashion by joining sets of tuples or relations satisfying subsets of the problem constraints. Real-time user queries can then be answered using search techniques on the resulting system.

4.2 Interesting Issues

As we begin this research, several interesting issues and questions occur to us. Among these are "What is the exact correspondence between the two approaches?", "How do the efficiencies of the two approaches compare?", "Can CLP heuristics be used to accelerate ABL algorithms?", "Can array computations be used to speed up CLP computations?", "Can ABL be implemented efficiently in parallel?", "What are the advantages of the array based representation?" and "Are there any algorithms or

techniques for Constraint Satisfaction which are facilitated by the array based approach to CBR?" These questions have in turn suggested some potential avenues of research in the two areas we have discussed in this paper. We now discuss some of these possible research directions.

4.3 Potential Research Ideas

As stated in Section 3.2.2, the join procedure included in Møller's thesis may be inefficient, since it generates the full Cartesian product of the tuples of the common variables of the relations being joined. We believe that a better way to proceed may be to use a search procedure that looks for tuples in one relation that have matching values in common variables in the tuples of the other. This technique might define a relationship between the system simulation and the join procedure. It appears that performing a join is equivalent to doing a query of all identical values of common variables from one relation to the other. It also might reduce the time required for system simulation if the tuples are in some sort of predefined order. If that were the case, then searches could be cut off once certain values were reached. Of course, this would increase the complexity of the relational join operation. Another possibility is to use relational database techniques to build the system of relations. A recent publication in the area is [23]. In the paper, the authors establish a sufficient condition for the constraint hypergraph of a CSP which enables the corresponding problem to be decomposed into subproblems. They show that the use of existing algorithms in conjunction with the decomposition strategy can result in improvements in efficiency.

It would be interesting to examine the correspondences between the two approaches for finite domain constraint problems. In addition, we see the potential for parallel implementations of both system modelling and system simulation. Finding effective parallel algorithms for these operations might make ABL techniques feasible for large scale problems. We feel that there are certainly other possible heuristics for compression of relations and building the system and feel that these merit some investigation. An interesting open question is whether it is more important to minimize the time complexity of building the system (in particular for dynamic problems whose system must be repeatedly built) or the space complexity of the final system. We note that the two goals are not necessarily contradictory. Lastly, we note that since it is of practical importance to make the system as small as possible, it would be prudent to remove variables from the system wherever possible. In particular, if the state of a variable is a single value then it is a constant for any positive tuple. That variable might be removed from the system.

References

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley and Sons, 1989.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] H. Aït-Kaci and A. Podelski. Entailment and disentailment of order-sorted feature constraints. manuscript, 1993.

[4] Bang & Olufsen. *Beologic*, 1990.

[5] A. Barr, E. Feigenbaum, and P. Cohen. *The Handbook of Artifical Intelligence, Volumes 1 & 2*. Morgan Kaufmann, 1981.

[6] A. Bockmayr. Logic programming with pseudo-boolean constraints. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 327–350. MIT Press, 1993.

[7] J. Brown, S. Pakin, and R. Polivka. *APL2 at a Glance*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[8] P. Codognet and G. Nardiello. Enhancing the constraint solving power of CLP(FD) by means of path consistency methods. In A. Podelski, editor, *Constraint Programming: Basics and Trends, LNCS 910*, pages 39–61. Springer-Verlag, 1994.

[9] M. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[10] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55(1):87–107, 1992.

[11] Y. Deville and P. van Hentenryck. An efficient arc consistency algorithm for a class of CSPs. In *Proceedings of the International Joint Conference on AI*, pages 325–330, 1991.

[12] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving car sequencing problem in constraint logic programming. In *Proceedings of the European Conference on AI*, pages 290–295, 1988.

[13] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertier. The constraint logic programming language CHIP. In *Proceeedings of the 5th International Conference on 2nd Generation Computer Systems*, pages 249–264, Tokyo, Japan, 1988.

[14] A. Dovier and G. Rossi. Embedding extensional finite sets in CLP. In *Proceedings of the International Logic Programming Symposium*, pages 540–556, 1993.

[15] O. Franksen. Group representations of finite polyvalent logic - a case study using APL notation. In A. Niemi, editor, *A Link Between Science and Applications of Automatic Control*, pages 875–887. Permangon Press, Oxford and New York, 1979.

[16] O. Franksen. Array-based logic - its algebraic inheritence from Graßman and Peirce. In *Graßman-Tagung (150 Jahre 'Lineale Ausdehnungslehre', - Werk und Wirkung Hermann G. Graßmans)*, Rugen, Germany, May 1994.

[17] E. Freuder. Synthesizing constraint expresssions. *Communications of the ACM*, 21:958–966, 1978.

[18] E. Freuder. A sufficient condition for backtrack free search. *Journal of the ACM*, 29(1):24–32, 1982.

[19] E. Freuder and D. Sabin. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming, LNCS 874*. Springer-Verlag, 1994.

[20] M. Garey and D. Johnson. *Computers and Intractiblity: A Guide to the Theory of NP-Completeness.* H. Freeman, San Fransisco, CA, 1978.

[21] H. Geffner and J. Pearl. An improved constraint propagation algorithm for diagnosis. In *Proceedings of the 10th IJCAI*, pages 1105–1111, Milan, Italy, 1987.

[22] H. Guesgen and J. Hertzberg. *A Perspective of Constraint Based Reasoning – An Introductory Tutorial, LNCS 597*. Springer-Verlag, 1992.

[23] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[24] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[25] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:262–313, 1980.

[26] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on the on Principles of Programming Languages*, pages 111–119, Munich, West Germany, 1987.

[27] J. Jaffar, J.-L. Lassez, and M. Maher. A logic programming scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*, pages 441–467. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[28] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[29] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[30] M. Jenkins. ABL heuristics. Personal Communication, 1995.

[31] M. Jenkins and W. Jenkins. *Q'Nial Language.* NIAL Systems Limited, Ottawa, 1987.

[32] R. Kowalski and D. Kuehner. Linear resolution with selective function. *Artificial Intelligence*, 2:227–260, 1971.

[33] B. Kristensen. Arraybaserede sikringsanlaeg til jernbanedrift. Master's thesis, Tehnical University of Denmark, Lyngby, Denmark, 1995.

[34] B. Kristensen. Signal interlocking systems by array-based logic. In *AI'95 Fifteenth International Conference, EC2 & Cie*, June 1995.

[35] J. Lloyd. *Foundations of Logic Programming, Second, Extended Edition.* Springer-Verlag, New York, 1987.

[36] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[37] A. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley & Sons, 1987.

[38] A. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[39] A. Mackworth and E. Freuder. The complexity of constraint satisfaction revisted. *Artificial Intelligence*, 59(1–2):57–62, 1993.

[40] D. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. In *Extensions of Logic Programming: International Workshop, LNCS 475*, pages 253–251. Springer-Verlag, 1991.

[41] S. Minton, M. Johnston, A. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 17–24, 1990.

[42] R. Mohr and T. Henderson. Arc and path consistency revisted. *Artificial Intelligence*, 28:225–233, 1986.

[43] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the European Conference on AI*, pages 651–656, Munich, West Germany, 1988.

[44] G. Møller. *On the Technology of Array-Based Logic.* PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1995.

[45] A. Pedersen. *Digraph Representation in Array-Based Logic.* PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1992.

[46] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[47] J. Robinson. Logic and logic programming. *Comunications of the ACM*, 35(3):40–65, 1992.

[48] N. Sadeh, K. Sycara, and Y. Xiong. Backtracking techniques for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 76:455–480, 1995.

[49] V. Saraswat. The category of constraint systems is cartesian-closed. In *Proceedings of LICS '92, Symposium on Logic in Computer Science*, pages 341–345. IEEE Press, 1992.

[50] D. Scott. Domains for denotational semantics. In *Proceedings of ICALP '82, International Colloquium on Automata, Languages and Programming, LNCS 140*. Springer-Verlag, 1982.

[51] R. Stallman and G. Sussman. Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[52] E. Tsang. *Foundations of Constraint Satisfaction*. Acadmic Press, New York, 1993.

[53] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambidge, Massachusetts, 1989.

[54] P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consitency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.

[55] C. Walinsky. CLP(Σ^*): Constraint logic programming with regular sets. In *Proceedings of the 6th International Conference on Logic Programming*, pages 181–196, Lisbon, Portugal, 1989.

[56] L. Walras. *Elements of Pure Economics*. Allen and Unwin, London, England, 1954.

[57] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, November 1972.

[58] D. Waltz. Understanding line drawings in scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.