Exploiting Interchangeabilities in Constraint Satisfaction Problems*

Alois Haselbock
Institut fur Informationssysterne
Technische Universitat Wien
Paniglgasse 16, A-1040 Vienna, Austria

Abstract

Constraint satisfaction - a method for representing and solving many AI problems in a very elegant manner - is a well-studied research area of recent years. Freuder observed that some constraint satisfaction problems are fashioned so that certain domain values of constraint variables are interchangeable. The use of such knowledge can increase search efficiency drastically by reducing the problem. In this paper we carry on these considerations and give a formal foundation of interchangeabilities by the notion of domain partitions induced by equivalence relations. We show how these domain partitions can be used in a very accurate manner by the majority of existing constraint propagation algorithms and introduce a novel backtrack procedure exploiting such interchangeabilities of domain values. Both theoretical analysis and experiments indicate that our proposed approach is an improvement of Freuder's use of neighborhood interchangeability and has very good behavior for certain problem types.

1 Introduction and Motivation

Constraint Satisfaction is a well-studied research area of recent years. Using constraint satisfaction methods, many problems - especially in AI - can be represented in a very declarative way by identifying the variables of interest for the problem, laying down the domains for the variables and restricting the variable assignments by constraints. More formally, a constraint network R consists of a finite set $V = \{v_1, \ldots, v_n\}$ of variables and a set $C = \{c_1, \ldots, c_e\}$ of constraints. Associated with each variable v; is a finite, discrete domain D_v. A constraint c on the variables $V_k \subseteq V$ is in its extensional form a subset of the Cartesian product of the domains of the afflicted variables. The expression var(c) denotes the tuple of variables the constraint is defined on, and $rel(c) \subseteq \mathbf{X}_{v \in var(c)} D_v$ is the relational information of the constraint c.[1] The assignment of a value $d \in D_v$ to a variable v is denoted by v<—d. A tuple t of assignments of variables $V_k \subseteq V$ satisfies a constraint c, if and only if $t[var(c)] \in rel(c)$.[2]

A constraint satisfaction problem (CSP) is the task of finding one or all variable assignments for a constraint network R such that all the constraints of R are satisfied. There are various techniques (like network consistency techniques or backtrack search procedures) for the handling and solving of constraint networks (good descriptions can be found in [Mackworth, 1977; Mackworth, 1987; Haralick and Elliott, 1980; Dechter and Pearl, 1988; Dechter, 1992]). Though, it is well-known that the use of special methods for certain problem types can reduce search effort. In this paper we focus on problems where the domain values are structured objects rather than atomic data, and the constraints refer to attributes of the objects and not to the objects as a whole. Examples of application areas of these types of problems are design, configuration, or diagnosis. Mittal and Frayman [Mittal and Frayman, 1987] have proposed an approach of partial choices, where at each search step a decision is made only for a part (i.e., an attribute) of a constraint variable. The methods partial commitment and partial guess are essentially least commitment techniques where some underconstrained decisions are delayed until more information is derived. Another interesting work in this field is [Mackworth et al., 1985], where an hierarchical arc-consistency procedure (HAC) is introduced. This algorithm proceeds on the assumption that the variable domains are organized hierarchically in subsets, where the leaf nodes of each domain hierarchy are the intrinsic domain elements, whilst the "abstract" nodes represent groups of domain values with common properties. If the constraints treat groups of domain values equally, HAC avoids repetitive checks by filtering on abstract domain levels.

Thus, for many problem areas it is a matter of fact that some domain objects of a constraint variable behave in the same manner and it is therefore a waste of search time to handle them as totally different objects. Freuder introduced in [Freuder, 1991] the notion of interchangeability, where two domain values are interchan-

*This work was supported by Siemens Austria AG under project grant CSS (GR 21/96106/4).

[1] Our notation is similar to the one used in [Dechter, 1992].

[2] t[A] stands for the projection in the sense of relational algebra and extracts for our purposes exactly those values of the tuple t which are relevant to the constraint.

geable in some local or global environment, if they can be substituted for each other without any effects to the environment. Let us summarize the two main definitions.

DEFINITION 1.1 (FULL INTERCHANGEABILITY [FREUDER, 1991]) Two values d_1 and d_2 of a domain D_v of a constraint network R are fully interchangeable, if and only if (1) every solution to R which contains d\ as an assignment for v remains a solution when d_2 is substituted for d_1, and (2) every solution to R which contains d_2 as an assignment for v remains a solution when d_1 is substituted for d_2.

DEFINITION 1.2 (NEIGHBORHOOD INTERCHANGEABILITY [FREUDER, 1991]) Two values d_1 and d_2 for a variable v in a constraint network are said to be neighborhood interchangeable, if and only tf the following condition holds (C is the set of all constraints in the network):

$$\forall c \in C : v \in var(c) \rightarrow \{t' \mid t \in rel(c), t[v] = d_1, t' = t[var(c) - v]\} = \{t' \mid t \in rel(c), t[v] = d_2, t' = t[var(c) - v]\}.$$

Neighborhood interchangeability ts a sufficient, but not a necessary condition for full interchangeability.

It is shown in [Freuder, 1991] that all neighborhood interchangeabilities can be computed in a pre-phase of search in $0(n^2.a^2)$[3]. Then, all interchangeable values can be replaced by one representative, which is essentially a form of problem reduction, because a subsequent search has to handle a network of eventually smaller size. It is proven that the usage of local interchangeabilities at preprocessing time is guaranteed to be cost effective for some CSPs.

But it is often the fact that different domain values are interchangeable only w.r.t. certain constraints, whilst there are constraints which distinguish between the intrinsic values. In that cases, neighborhood interchangeability makes no use of that nuances of equivalence. We extend Freuder's ideas in a strict manner: we try to find groups of domain values which are essentially not distinguishable w.r.t. a single constraint. If a domain D_v is filtered by a constraint c, the values of D_v must be enumerated and checked against other value constellations. We propose the replacement of D_v by classes of values (a domain partition w.r.t. c), where the size of the new domain and therefore the number of checks are minimal subject to c, without loss of any information.

The paper is organized in the following way: Section 2 gives a formal definition of domain partitions and shows a simple way of computing them by constructing discrimination trees. In Section 3 we show how various algorithms can be modified in term to use these domain classifications, and in Section 4 we evaluate the proposed techniques by experimental results. Section 5 concludes the work.

[3] n is the number of variables and a is the maximum domain size. The network is assumed to be binary.

2 Domain Partitions

On each domain D_v of a constraint network and each constraint c where $v \in var(c)$, an equivalence relation E^v_c can be defined, where two domain values of D_v are equivalent in regard of E^v_c, if and only if they behave in exactly the same manner w.r.t. the constraint c. The key item is that this equivalence relation induces a partition of a variable domain into groups of locally (i.e., subject to a single constraint) interchangeable values.

DEFINITION 2.1 (THE RELATION E^v_c) Let c be a constraint of a constraint network R and v $\in$ var(c). Two values d\ and d_2 of the domain D_v are in the relation E^v_c (we write $d_1 E^v_c d_2$), if and only if the following condition holds:

$$\{t' \mid t \in rel(c), t[v] = d_1, t' = t[var(c) - v]\} = \{t' \mid t \in rel(c), t[v] = d_2, t' = t[var(c) - v]\}.$$

(Note the difference to neighborhood interchangeability: the relation E^v_c depends on the single constraint c.)

THEOREM 2.1 The relation E^v_c is an equivalence relation.

PROOF SKETCH: Two domain values $d_1, d_2 \in D_v$ are in the relation E^v_c, if the set of all assignment tuples where d\ is assigned to v and the constraint c is satisfied is equivalent to the set where d_2 is assigned to v. Therefore, E^v_c is in fact an equivalence relation, because it is defined by the equality-relatoon on sets, which is - of course - an equivalence relation. □

COROLLARY 2.2 It is well-known in set theory that each equivalence relation on a set S induces a partition of S, which is a set of non-vacuous subsets of S where the elements of the partition are mutual exclusive and the decomposition is exhaustive. Thus, according to each constraint c of a constraint network every domain D_v $(v \in var(c))$ can be partitioned into equivalence classes induced by E^v_c. We write $\pi_{E^v_c}$, or π^v_c for short.

EXAMPLE: Let the variables v_1 v_2 and v_3 represent three ports of a board where modules must be mounted on. The available modules have two main characteristics: their mode ("analog" or "digital", abbreviated by a resp. d) and their version number (1 or 2). Thus, the domains of the variables can be specified by

$$D_{v_1} = D_{v_2} = D_{v_3} = \{m^1_a, m^2_a, m^1_d, m^2_d\}.$$

The following constraints restrict the possible constellations: (c_{12}) the modules mounted on v_1 and v_2 must be of different mode; (c_{13}) the modules mounted on v_1 and v_3 must have different version numbers. From the perspective of port v_1, the domain D_{v_1} can be partitioned in the following way:

$$\pi^{v_1}_{c_{12}} = \{\{m^1_a, m^2_a\}, \{m^1_d, m^2_d\}\},$$
$$\pi^{v_1}_{c_{13}} = \{\{m^1_a, m^1_d\}, \{m^2_a, m^2_d\}\}.$$

Note that in this simple CSP there is no pair of neighborhood or fully interchangeable domain values in D_{v_1} in the sense of Definitions 1.1 and 1.2. □

In that way, every element of a domain partition π^v_c is a set of domain values which are interchangeable w.r.t. the constraint c. Similar to [Freuder, 1991],

these domain partitions can be computed by generating *discrimination trees*. A discrimination tree is a tuple (N, E, ϕ, ψ), where N is a set of nodes, E is a set of edges (each edge is a pair $(n1, n2)$, $n1, n2 \in N$), ϕ is a labeling function which assigns to each node of N a (possible empty) set of domain values, and ψ is a labeling function which assigns to each edge from E a tuple of domain values.

We postulate a procedure $DT(c, v)$ for computing a discrimination tree for the domain of a variable v w.r.t. the constraint c ($v \in var(c)$). Basically, the following is done by $DT(c, v)$: for each value d of the domain D_v starting at the root a branch is generated, where the labels ψ of the edges of this branch are the corresponding consistent tuples t', where $t \in rel(c)$ such that $t[v] = d$ and $t' = t[var(c) - v]$. If existing suitable parts of the branch are constructed already, they will be used. If all consistent tuples of the variable v are on the branch, the actual domain value is added to the node label ϕ of the current node. A canonical ordering of the domain values is assumed. Therefore, the set of non-vacuous node labels of a discrimination tree produced by a procedure call $DT(c, v)$ is a domain partition π_c^v as described previously.[4]

An upper bound of the complexity of procedure $DT(c, v)$ is $O(a^k)$ where a is the domain size of v and k is the arity of c. This can be proven by the following considerations: the outer loop (expand a branch for each domain value) is performed a times, and the maximal number of suitable tuples t' is the number of different $(k-1)$-tuples of domain elements, which is at most a^{k-1}. Thus, a worst-case upper bound of the procedure $DT(c, v)$ is $O(a.a^{k-1}) = O(a^k)$. If for every constraint every domain of the corresponding variables is decomposed by building a discrimination tree, the upper bound complexity is $O(e.k.a^k)$, where e is the number of constraints and all constraints are at most of arity k. Therefore, an effort of this magnitude must be spent for the computation of all domain partitions in a preprocessing phase of search.

3 Adaptation of Various Constraint Propagation Algorithms

Now we are in the position to show how these domain partitions can be used to increase efficiency of various existing algorithms. We give a few modifications of the key procedures and show the advantages of the use of such interchangeability-techniques for certain problem types. We focus on binary CSPs.

3.1 Constraint Filtering

Constraints are most commonly used in a destructive manner. The critical and most time consuming task in network consistency procedures is to check if all values of a particular variable domain can potentially be member of a solution. These checks are done repetitively for singular variables w.r.t. singular constraints. In the case of binary constraints, usually the procedure revise(v_i, v_j) is used, which removes all values of D_V, for which no value of the domain of Vj can be found such that the binary constraint $c_{ij;}$ between v_i and V_j is satisfied. Thus, the worst-case complexity of revise is $0(a^2)$ where a is the maximum domain size.

In Figure 1, a modified procedure called $revise^{dp}$ is depicted. We use the expression $\overline{d}_c^v$ (d is a domain value) to denote the equivalence class of value d subject to the equivalence relation E_c^v: $\overline{d}_c^v = \{d' \in D_v \mid dE_c^v d'\}$.

```
1   procedure revise^dp(v_i, v_j):
2       Δ_i := {};
3       do until D_{v_i} becomes empty:
4           x := an element of D_{v_i};
5           Δ_j := D_{v_j};
6           do until Δ_j becomes empty:
7               y := an element of Δ_j;
8               if (x, y) ∈ rel(c_ij)
9                   then Δ_i := Δ_i ∪ (x̄^{v_i}_{c_ij} ∩ D_{v_i});
10                      Δ_j := {};
11                  else Δ_j := Δ_j − ȳ^{v_j}_{c_ij};
12          D_{v_i} := D_{v_i} − x̄^{v_i}_{c_ij};
13      D_{v_i} := Δ_i.
```

Figure 1: A revise procedure using domain partitions.

The main difference between the "classical" revise and revise^{dp} is that the former checks in the worst case all tuples from $D_{v_i} \times D_{v_j}$ and the latter treats groups of interchangeable values equally and therefore possibly saves checks utilizing information included in the domain partitions. If we assume that the domain partitions (i.e., the sets π_c^v for all constraints c and all variables $v \in var(c)$) are of size $a1$ $(1 \leq a1 \leq a)$, a worst case bound of algorithm $revise^{dp}$ is $O(a1^2)$. We see that if the structure of a constraint network gets no use of domain partitions (all domain partitions are of same size as their corresponding domains), a\ would be equal to a and the worst case behavior is not worse than that of the standard revise algorithm. Of course, the smaller a\ is in comparison to a, the better is the improvement.

It is easy to see that revise and revise^{dp} produce exactly the same outcome. In that way, algorithms which use revise (such as arc- and path-consistency algorithms) simply have to change each call to revise by a call to revise^{dp} and get effective use of reduced domain sizes. Also many backtrack procedures use revise and therefore can benefit from revise^{dp}. The palette ranges from classical chronological backtracking, where at each search step the domain of the current variable is made consistent to all past assignments, to various forms of look-ahead schemes, where the future search space is brought to certain degrees of (arc-)consistency. Particularly at the latter approaches, revise is used excessively.

3.2 Backtrack Search

In the following, we want to evolve a slightly modified tree search scheme where interchangeable search branches are recognized by the use of domain partition infor-

[4] A detailed description of the procedure DT and a proof that it actually computes the right thing is given in an extended report version of this paper.

mation. The structure of the algorithm is basically the same as classical backtrack tree search as described, for instance, in [Fox and Nadel, 1989].

But first we have to give some notations we need for the development of the search procedure. Each output of a traditional backtrack procedure is an assignment tuple representing a solution for the given CSP. Because we want to handle groups of interchangeable values, we have to modify the form of the output. Instead of single assignment values, sets are used. In that way, assignment tuples are shifted to assignment bundles.

DEFINITION 3.1 (ASSIGNMENT BUNDLE) Let V be the set of n variables of a constraint network R. An n-tuple Δ where the ith element of Δ $(1 \leq i \leq n)$ ts a non-vacuous subset of the domain D_{V_i} is called an assignment bundle.

DEFINITION 3.2 (SOLUTION BUNDLE) Let T be the set of all solutions to a given constraint network R. An assignment bundle $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ on the variables V of R is said to be a solution bundle, if and only tf

$$\Delta_1 \times \ldots \times \Delta_n \subseteq T.$$

Therefore, solution bundles represent groups of path' through the search tree, where each path stands for a valid variable assignment of the constraint network. The terms of local and global consistency (see, for instance, [Dechter, 1992)) can be extended to assignment bundles.

DEFINITION 3.3 (CONSISTENCY OF ASSIGNMENT BUNDLES)

- *An assignment bundle Δ^p on the variables $V_p \subseteq V$ is said to be* locally consistent, *if every assignment tuple extractable from Δ^p is locally[5] consistent.*
- *An assignment bundle Δ^p on the variables $V_p \subseteq V$ is said to be* globally consistent, *if there exists an extension assignment bundle Δ^J on the variables $(V - V_p)$ such that $\Delta^p \cup \Delta^J$ is a solution bundle.*
- *An assignment bundle Δ^p is said to be* inconsistent, *if every assignment tuple extractable from Δ^p is inconsistent (i.e., no tuple can be extended to a solution).*

Now we want to modify the classical backtrack search shell such that for each pass a bundle of assignments is computed. The following theorem gives us the fundamental basis for the utilization of domain partitions for that purposes.

THEOREM 3.1 *Let Δ^p be an assignment bundle on the variables $V_p \subset V$ which is either globally consistent or inconsistent. Let v be a variable of $V - V_p$ and δ_v be a subset of (or equal to) the domain D_v such that the following two conditions hold:*

1. *$\Delta^p + \delta_v$ is locally consistent;*
2. *(C^J are all binary constraints from v to variables of $V - V_p$)*
   $$\forall d_1, d_2 \in \delta_v : \forall c \in C^J : d_1 E_c^v d_2.$$

Then $\Delta^p + \delta_v$ is either globally consistent or inconsistent.

[5] I.e., all the constraints of the subnetwork defined by the variables V_p are satisfied.

PROOF SKETCH: (I) If Δ^p is inconsistent, then there is no way to extend it to a solution and therefore $\Delta^p + \delta_v$ is also inconsistent. (II) Suppose Δ^p is globally consistent and $\Delta^p + \delta_v$ is neither globally consistent nor inconsistent. Then two assignment tuples t_1 and t_2 have to exist, where $t_1, t_2 \in \times_{\delta \in \Delta^p + \delta_v} \delta$ and one of them (say t_1) is globally consistent (an extension to a solution is guaranteed) and the other (t_2) is not globally consistent. This can not be the case, because the assignment values of v in t_1 and t_2 are interchangeable both w.r.t. the *past* (by the maintenance of local consistency) and w.r.t. all constraints to *future* variables. □

```
1   procedure backtracking^dp (k,D):
2       revise^dp (k,p) for 1 ≤ p < k;
3           % or do some kind of look ahead filtering
4       dk ← D[k];
5       do until dk becomes empty:
6           choose x from dk;
7           C^J ← all constraints on v_k to future vars;
8           D[k] ← (⋂_{c∈C^J} x̄_c^{v_k}) ∩ dk;
9           dk ← dk − D[k];
10          if k = n
11              then output(D);
12              else backtracking^dp (k + 1, D).
```

Figure 2: Using domain partitions at backtrack search.

Now the modification of the backtrack procedure is easy Figure 2 sketches the new algorithm backtracking^{dp}. At each cycle in the search process the set of variables can be partitioned into three groups: the past variables, the current variable, and the future variables. Since all the remaining domain values of the current variable are consistent to the assignments of the past variables (this is guaranteed by the revise at line 2), they are interchangeable w.r.t. that partial solution (bundle). Now the domain of the current variable is going to be partitioned along their constraints to future variables. For each group of such interchangeable values a new search branch is opened. Clearly, each output of a call to that procedure is an assignment bundle. The following theorem states soundness and completeness of the proposed algorithm.

THEOREM 3.2 Let D be the set of all variable domains of a constraint network R. Each output Δ of a procedure call backtracking^{dp}(1, D) ts a solution bundle to R. The set of all outputs cover all solutions.

PROOF SKETCH: The design of the algorithm is guided by Theorem 3.1. Thus, at each cycle of the search process the derived assignment bundle is either globally consistent or inconsistent. Inconsistency leads to a dead-end, each output is globally consistent.

The algorithm passes through the whole search tree (even if interchangeable subtrees are condensed) and therefore computes all solutions. □

The advantageous behavior of the search shell backtracking^{dp} for certain problem types is obvious. Interchangeable search branches are bundled and visited once. If a dead-end occurs, all the partial assignments

represented by the derived assignment bundle are proven to be conflicting. A solution bundle represents a group of valid assignments.

So we conclude, if the described domain partition knowledge is available, it can be used in a wide range of CSP algorithms (both at filtering and search) with minimal change of procedures. Apart from a small amount of additional overhead of computing domain partitions in a preprocessing phase and managing groups of domain values instead of singular elements, the new worst-case complexities are not worse than that of the original algorithms. If the problem structures are adequate for (i.e., if the cardinality of domain decompositions are really smaller than the original domain sizes), effective cost reductions can be achieved.

4 Performance Analysis by Experiments

Now we want to investigate the indicated performance improvements of our augmented search technique by experimental analysis. To identify those areas of CSPs interchangeability makes most capital out of, we are going to use the same test model as proposed in [Benson and Freuder, 1992].

4.1 The Experimental Model

Different types of problems can be characterized by the following four parameters: (1) n, the number of constraint variables. (2) a, the maximum domain size. (3) t, the constraint tightness. The tightness of a constraint is the fraction of the number of forbidden tuples to the number of all possible tuples, and ranges therefore between 0 and 1. The higher t grows, the more value tuples are ruled out by the constraint (constraints with high values of t are said to be tight). (4) d, the constraint density. This is an indicator of how many constraints are defined in the network and therefore, how dense the constraint network is. d is a value between 0 and 1 and is specified as follows: Let n be the number of variables, e the number of constraints; the maximum number of constraints e_{max} is $\frac{n.(n-1)}{2}$, the minimum e_{min} is n — 1 (a connected constraint graph is assumed); then d is the v a $\frac{e-e_{min}}{e_{max}-e_{min}}$ In that way, the higher d is, the more constraints are in the network.

Different algorithms are run on randomly generated CSPs and the results are compared. The tests are restricted to binary CSPs.

4.2 Test Cases and Results

Our CSP generator produces samples of random CSPs, where the four parameters n, a, t and d ranges on adjusted intervals. In [Benson and Freuder, 1992] it was pointed out that interchangeability techniques are most profitable if (1) the problem space grows (n and a grow), (2) the constraint tightness is small, and (3) the constraint-density is small. The combination of the last two points specifies those regions of problems where the CSPs are under-constrained. These are problems with many solutions.

We tested our algorithms in that manner and came to similar results. Furthermore, our analysis shows that the use of domain partitions w.r.t. single constraints beats neighborhood interchangeability in all the test cases. This should be demonstrated by the subsequent results.

In the following, three forward-checking search procedures are compared. The first is classical forward-checking (FC). The second is forward-checking where all neighborhood interchangeable domain values are replaced by one representative value in the preface of search (FC-NI). The third is an instance of the search scheme backtracking^{dp} (see Figure 2) where forward-checking filtering is used. We call it FC-DP. A good indicator of the complexity of the search process is the number of consistency checks. Of course, the checks needed for the computation of neighborhood interchangeability resp. the domain partitions are added to the run-time checks. The sample of each test are 50 randomly generated CSPs.

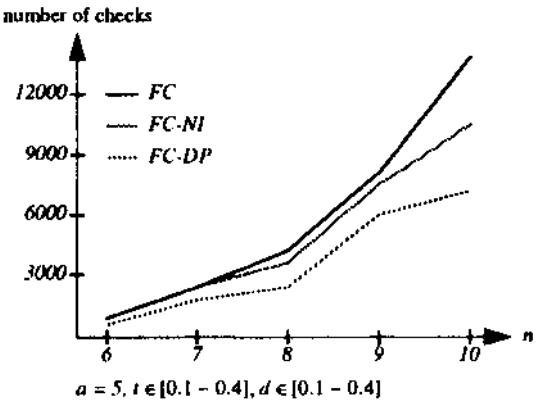

Figure 3: The effects of the use of interchangeability w.r.t. the number of variables.

The first test demonstrates that utilization of interchangeability grows if the problem increases. The variable size n steps from 6 to 10, the maximum domain size a is fixed on 5, and the constraint tightness t and density d are from the interval $[0.1 - 0.4]$ (the profitable ranges for the use of interchangeability!).

Figure 3 shows the results. It can be seen that the positive effect of *FC-NI* and *FC-DP* grows with the size of n. Furthermore, our algorithm *FC-DP* is clearly better than FC-NI, and the distance increases with n.

The second and third test holds n = 10, d = 5, and steps t (resp. d) from 0.1 to 0.9, d (resp. t) is randomly chosen from interval [0.1 — 0.3]. As depicted in Figure 4 and Figure 5, FC-NI and FC-DP are superior to classical FC when t (resp. d) is small. It can also be seen that FC-DP definitely beats FC-NI at these problem types.

These results are convincing. The more tuples are permitted by a constraint (the smaller t is), the better

[6]Forward-checking is a backtrack procedure where at each cycle in the search process all the future variables are filtered against the last-assigned variable. This method is known to behave in a very efficient manner [Haralick and Elliott, 1980].

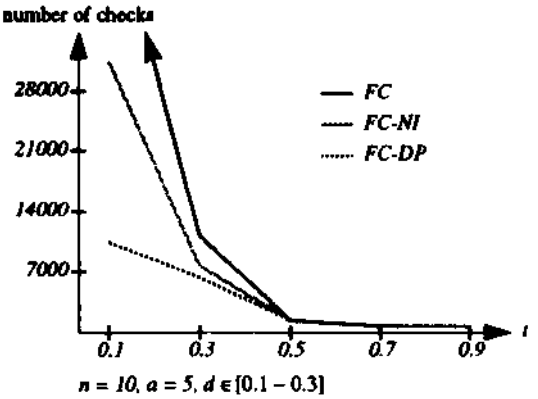

Figure 4: The effects of the use of interchangeability w.r.t. constraint tightness.

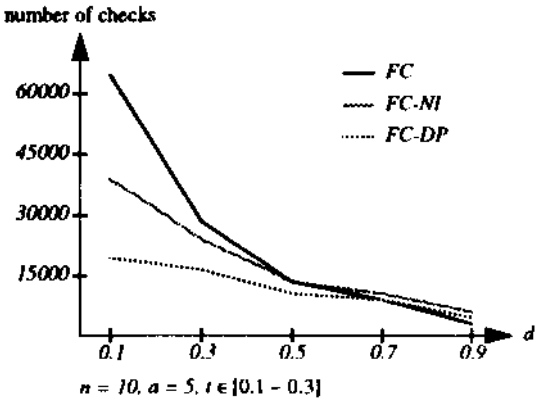

Figure 5: The effects of the use of interchangeability w.r.t. constraint density.

is the chance that different variable values behave in the same manner w.r.t. the constraint and therefore came into the same class of domain values. If the constraint net is not dense (there are few constraints), there are at each choice point for a variable assignment few future constraints and the interchangeable groups of values are going to split less (line 8 of the backtracking^{dp} algorithm, depicted in Figure 2).

The lack of FC-NI is that it uses only the information that domain values are interchangeable w.r.t. all the connected constraints. In that sense, FC-DP is more accurate because of greater degree of granularity. And this can be achieved with the same overhead as the computation of all neighborhood interchangeabilities.

5 Conclusion

We have developed a formal basis for extraction and representation of interchangeable domain values in constraint satisfaction problems. The bulk of existing constraint satisfaction algorithms can be adapted to exploit this information. Application fields arise in many areas of model-based reasoning (such as configuration, simulation or diagnosis), mainly in those cases where component-oriented systems are modelled in terms of constraint problems. Thereby, identifying the variables for a CSP, possible values for the variables are most often representations of complex real-world objects rather than unstructured constants. These objects (consider components) are described by various features. Along these features objects can be grouped into classes where the elements of each class have some set of common properties. In that way, constraints are specifying relations on different aspects of the system and take classes of components rather than singular values into consideration. If that is noticed at reasoning, a much more adequate inference technique is employed.

Acknowledgments

1 am very grateful to Markus Stumptner and Thomas Havelka for valuable discussions and comments on an earlier version of this paper.

References

[Benson and Freuder, 1992] Brent W. Benson and Eugene C. Freuder. Interchangeability preprocessing can improve forward checking search. In Proc. ECAI, pages 28-30, Vienna, August 1992.

[Dechter and Pearl, 1988]
Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, 34:1 38, 1988.

[Dechter, 1992] Rina Dechter. From local to global consistency. Artificial Intelligence, 55:87-107, 1992.

[Fox and Nadel, 1989] Mark Fox and Bernard Nadel. Constraint directed reasoning. Tutorial of the IJCA1-89, 1989.

[Freuder, 1991] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In Proc. AAAI Con}., pages 227233, 1991.

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, 14:263-313, 1980.

[Mackworth et al, 1985] Alan K. Mackworth, Jan A. Mulder, and William S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. Computational Intelligence, 1(3):118-126, 1985.

[Mackworth, 1977] Alan K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8:99-118, 1977.

[Mackworth, 1987] A.K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, Encyclopedia of Artificial Intelligence, pages 205-211 John Wiley & Sons, 1987.

[Mittal and Frayman, 1987] Sanjay Mittal and Felix Fray man. Making partial choices in constraint reasoning problems. In Proceedings AAAJ Conference, pages 631-636, July 1987.