On the applicability of *extended Logic Programming* for representation and solution of *Constraint Satisfaction Problems*
DRAFT VERSION

Steven Woods
89800462

Abstract

Logic programming languages such as Prolog provide a convenient way in which to represent relations such as variable constraints. Additionally, the nondeterministic nature of the unifiction method underlying Prolog supports implicitly the search required to solve Constraint Satisfaction Problems (CSPs). The standard backtracking control structure of Prolog and other logic programming languages results in very large search spaces which makes it inadequate for interesting problems. However, an extension of the Prolog control structure which includes consistency methods has been shown to embody search procedures that utilize forward or active application of constraints which reduce the search space *a priori* rather than *a posteriori* during search.

Standard "AI" approaches to solving CSPs have been rooted in a melding of search strategies with constraint propagation algorithms to form hybridized search strategies that make use of domain independent and domain dependent heuristics during search. Additionally, stronger, richer representation languages are useful in terms of both controlling search refinement and providing more accurate problem descriptions.

In this paper we show that while extended logic programming with consistency methods is far superior to standard logic programming for addressing CSPs, it is still too rigid and inflexible in both representation and control issues to be considered as a *sufficient* toolset for the CSP application developer. Essentially, the tools provided are tied too tightly together to allow sufficient flexibility to take advantage of specific problem characteristics to best effect, and what is really needed is a toolset consisting of a strong representation language, search and consistency tools that may be merged to any degree, and support for dynamic changing of control strategy throughout the CSP search space.

1 Introduction

A broad class of problems may be represented as sets of constraints, and may be solved via methods described in the literature as constraint satisfaction techniques. While in general the Constraint Satisfaction Problem (CSP) is NP-complete[1], certain domains lend themselves well enough to heuristic approaches to problem simplification so that solutions or adequate partial solutions may be located in a reasonable amount of time and effort (Nadel 1989, Yang & Fong 1992, Minton, Johnston, Philips & Laird 1992, Prosser 1993, Woods 1993*b*, Woods 1993*a*). Recent work such as Nadel (1990) and Tolba, Charpillet & Haton (1991) has emphasized how important problem representation, with respect to selection of constraints and domain "objects", can be in terms of efficiency of solution. Other work dating from the early work of Waltz (1975) and Mackworth (1977), and including Mackworth (1981), Mackworth (1992), Dechter & Dechter (1987), Dechter (1990), Dechter (1992), Mackworth & Freuder (1985), Freuder (1991), Freuder & Wallace (1992), Mackworth & Freuder (1993) and others has focused on generalizing local reduction or consistency achievement in constraint problems, and on typifying the effectiveness of such approaches.

The "AI" community has primarily addressed constraint satisfaction in terms of techniques for hybridizing consistency methods with search, representation and complexity issues, and developing new approaches to achieving varying consistency in networks of constraints. However, no universal "paradigm" or "model" has really emerged as a generic catch-all for the discussion and modelling of constraint satisfaction problems, other than the typical constraint-graph, perhaps. Individual AI implementation-based work has been based on local implementations of published algorithms, perhaps novellized or merged in new ways with other techniques such as particular search strategies. Several works (Prosser 1993, Woods 1993*b*, van

[1] CSP NP-Completeness is demonstratable by restriction to Graph 3-Coloring for instance.

Run 1994) have indicated that they are part of larger works which hope to provide means of constructing a generic toolset for stating and solving CSP in many domains.

In the Logic Programming (LP) community, much interest in the past 8-10 years has been focused on providing support in logic programming for formulating and solving problems based on constraints. Logic programming languages have been shown to be capable of providing one way of representing both constraints and nondeterminism and of "freeing" an application developer from the overhead of specifying specific search routines. Unfortunately this lack of flexibility in creating variant search strategies or in taking advantage of local constraint propagation in advance of search failure is potentially extremely limiting in terms of performance. Consequently, logic programming languages have emerged, such as Logic Programming extended with constraint consistency (CCLP) (Van Hentenryck 1989), and Constraint Logic Programming (CLP), which function as constraint solvers by providing implicit representation possibilities and non-determinism while also providing "built-in" facilities for local consistency application or constraint propagation and search control.

In the course of this paper we will be examining the constraint logic programming paradigm from the perspective of the new languages and determining whether they have become general and powerful enough to capture the representation and solving control requirements of a "typical" real-world problem known as *Spatial Template Recognition* (STR) formulated as a CSP (Woods 1993*b*, Woods 1993*a*).

We will attempt to summarize the aspects of "Constraint Satisfaction" that would appear to be sufficient for this domain in the new extended LP world, and which are in apparent need of extension. Reference for the purposes of this paper will be made specifically to the recent work of Van Hentenryck (1989) and Van Hentenryck, Saraswat & Deville (unknown) in outlining the original extended (CC) LP framework. Other extensions of this work in LP with parallelism (Montanari & Rossi n.d., Kasif & Delcher n.d.), hierarchies (Menezes, Barahona & Codognet n.d.), and spatial domains (Dube & Yap n.d.), are also of interest in specification and solution of the STR problem, however, fall outside the direct scope of this paper.

2 Spatial Template Recognition (STR)

Generally speaking, STR is the problem of finding an instance of some predefined configuration of objects that may be deployed amongst a given situation, or set of object instances. An example of a spatial template in a two-dimensional molecular domain would be a "map" of a typical molecule of some type that describes the molecule in terms of the relative positions, types and orientations of the various atoms of the molecule. An example of a situation might be some two-dimensional sample containing many instances of different types of molecules, and the question formed by STR would be: "Is there an instance of this molecule in this situation?".

3 Constraint Satisfaction: an AI clearbox approach

A constraint satisfaction problem representation consists of three major components: A set of variables, a range of domain values for each variable, and a set of constraints amongst the variables restraining individual assignments of domain values to a particular variable depending on domain assignments to other variables. A solution to a CSP is simply an assignment to each variable of a domain value from that variable's domain such that all constraints restricting assignments are satisfied. In some cases we may be interested in only a single satisfying assignment, while in other problem instances all possible satisfying assignments may be of interest.

3.1 STR as CSP

A relatively obvious formulation of STR as a **Constraint Satisfaction Problem** (CSP) was proposed in Woods (1993*a*). We initially define STR as follows: A situation consists of a set of elements each with a set of characteristics (such as color, type, etc.). A template consists of a set of variables, each possibly limited by a set of range or specific characteristic values. Additionally, each variable is possibly connected to any other variable (or possibly set of variables) by a constraint-arc or set of arcs which impose relative characteristic contraints on aspects such as position or orientation, or which possibly insist upon equality of a pair of variable characteristic values. A template instance is "found" in a situation if a 1:1 mapping exists from situation elements to variables such that no situation element maps to more than one variable, and based on this assignment, all constraints represented by constraint-arcs are satisfied. A decision problem based on this definition could be expressed as "Given a Template T = (set of Variables V + set of Variable-constraints VC + Arc-constraints AC), and a Situation S = (set of Situation elements), is there an instance of T in S?". Problems specified in this fashion, where a solution is composed of a set of consistent element-to-slot assignments are well described in Kumar (1992). An example of a template constraint graph is shown in Figure 1. The graph has five node slots where each slot is constrained by several attribute ranges, and 11 inter-slot spatial constraints on distance, position, and orientation.

Of course in a "real" situation, this 1:1 mapping

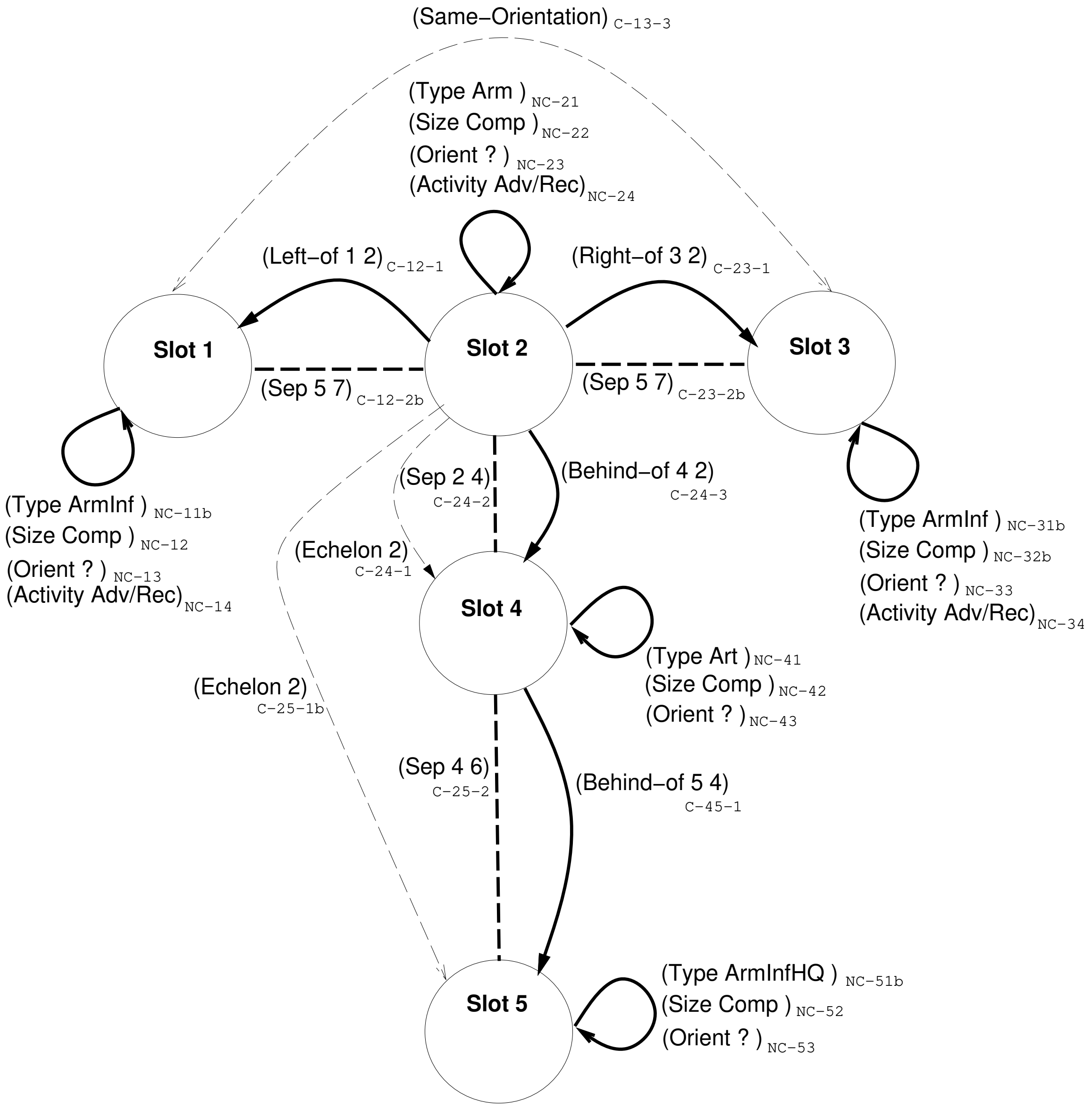

Figure 1: Whole CSP 5 variables, 11 arc constraints.

may be difficult or impossible to ascertain due a number of circumstances such as uncertainty or error in the perceived view of the world, or due to an inaccurate conception of of a template that contains either less or more elements or contraints than an actual template. If we still wish to solve the problem we must somehow fall back upon either some measurement of the "degree of fit/match" and estimate our confidence in this approximate solution, or perhaps abstract out the "most important" features of our template and attempt to locate this in a situation. In the approach defined in Woods (1993*b*) and Woods (1993*a*), abstraction through carefully defined constraint relaxation "levels" or template descriptions is combined with domain dependent and domain independent heuristics in a search strategy to offer a way of handling this uncertainty[2].

3.2 What generic CSP formulation provides

Formulation of a problem in a generic CSP format brings us immediately into the world of search, heuristics and approximate solutions, rather than minimalist or best solutions. A problem such as STR is formulated in the CSP framework described previously, and then some strategy must be utilized to locate a mapping of variables to domain values that satisfies some utility measure. For instance, in many problem formulations distinction is made between *hard* constraints that must not be violated in any candidate solution or mapping, and *soft* constraints that we would prefer not to violate but which we may relax if necessary. Of course this type of distinction can also be extended in some domains such as that of course scheduling discussed in Yang & Fong (1992), where one is faced with a wide range of constraint or preference rigidity on various characteristics of a solution, and where many of these are possibly contradictory. The acceptance of violations of some constraints in a solution is known as *constraint relaxation*. Complex strategies for relaxation have been examined, including the work of Freuder & Wallace (1992), and later of Yang & Fong (1992) and Woods (1993*b*) where abstraction is used as a model for the relaxation of constraints.

CSPs can be solved in several ways. One possibility is to "propagate" the domain restraints implied by finite variable domains and explicitly inter-variable constraint relations so as to iteratively reduce the size of variable domains until all remaining variable domain values are "consistent" with one another. These types of global constraint propagation algorithms have been explored at some length in the works of both Mackworth and Dechter. Another possible solution strategy (generate and test) is to generate all possible permutations of variable domain value combinations and test a complete assignment to see if the problem constraints are met. This is grossly inappropriate in even fairly simple problems for reasons of combinatorics. CSPs are typically solved with some kind of search. Basically a domain value is selected for each variable in some kind of turn such that the constraints are satisfied (or possibly satisfied to some degree), and then another variable is selected. If some variable cannot be instantiated so as to satisfy the constraints, then backtracking is invoked, and a previous assignment is undone. A great deal of literature has surrounded search both in AI and other fields, and CSP provides an excellent environment for different search strategies and approaches or philosophies to flourish. Recent work has included that of: Woods (1993*c*) where application of constraints throughout search is shown to be formulable in such a way as to reduce the size of the search space subsequent to backtracking; Woods (1993*a*) in which STR is shown feasible across a variety of simplistic search algorithms and large domain ranges when making use of locality-based spatial heuristics; Prosser (1993), where the entire range of algorithms commonly applied to CSPs is evaluated and contrasted in varying circumstances; and van Run (1994), where the ordering and re-ordering of variables during search is discussed in detail.

In very many CSP domains, we are interested in a solution that is locally minimal with respect to some evaluation function. To this end, recent work (Minton et al. 1992, Yang & Fong 1992) has shown how effectively "good" solutions may be found for domains of substantial size, where finding the "best" solutions is exponential. These works have utilized a type of iterative-repair heuristic during search that minimizes the numbers of conflicts among constraints based upon an initial (possibly random) assignement of variables to domain values.

CSP solutions may be characterized by a variety of search algorithms, domain dependent heuristics that are used to control a particular search strategy, domain independent heuristics that essentially change the nature of one search strategy to another, possibly more CSP-specific strategy, and localized strategies that tend to focus on repairing a rough solution iteratively until it is "good-enough". Each of these areas are currently active research areas, and little agreement has been forthcoming to date, even about which CSP problem characteristics favour which approach. Essentially, solutions are created uniquely for a particular problem domain from this toolset, and generalizations are made where possible. One obvious problem with this type of approach is the intense repetition of implementation amongst researchers and application-builders. It is true that the algorithms and methods

[2] In other work such as that of Lapointe & Proulx (1994), a different approach is used in which the fuzzy *distance* between a perceived instance and a template is measured and is used to determine a "match" between situation and template.

are frequently published in their entirety, and are discussed at some length in the literature, but algorithms must still be translated into code in the language desired, debugged, and application front ends need to be built for each use. Even in those rare cases where researchers make their code and documentation available, these provisions are rarely of a quality that could be said to be robust or standardized in any sense.

So then, CSP applications suffer from uniqueness. Each application is built upon a new set of algorithm implementations and this work is expended over and over in both the research and the development community. It would appear that a widely distributed and standardized tool that provided both a language capable of expressing a problem in terms of variables, domain ranges, and constraints, and the ability to generate solutions to these problems would be a great benefit. Each of these problems presents something of a different challenge. CSP has basically evolved as those who have written about it have extended it, and there is little agreement about what a "standard" problem might look like, and what types of constraints are typical. Additionally, CSP search covers a very broad range, and different strategies apply best to unique situations. Little if any work has attempted to categorize problem solutions in terms of the problem characteristics. Consequently, the provision of a generic "system" for solution of CSPs is problematic at best.

In summary, we see the following issues of prime importance for a tool that purports to provide a platform for the expression and solution of problems based in constraints:

- *Expressiveness/Representation* : can our problems be expressed concisely and "naturally" but also effectively in the language? For example, when some problem domains have been represented in terms of conceptual abstractions they have been observed to naturally fit certain search methodologies (such as downward refinement of solutions from abstract to specific) and demonstrate substantial practical performance and solution-quality improvements over single-level represented problems. In an abstraction hierarchy, a problem is defined in terms of several levels of components, initially containing only the problem *key* or *integral* features and subsequent changed representations containing additional detail. While some abstraction approaches may be seen as simple heuristic control strategies (for example, variable ordering), richer abstraction representations include mapping information between the levels that can be utilized in more detail control, for example in terms of defining a possible solution space where all "base level" solutions can be found through selecting upper-level abstract solutions. Figure 2 details such a mapping feature in a CSP problem. The usefulness of this property in assisting search control has been examined in Woods (1993*b*) for the STR domain, and in other works including Holte (1993) and Bacchus & Yang (1991) and Bacchus & Yang (1992). A system that could not express these abstractions and mappings is by definition not capable of "naturally" representing important characteristics of the problem.
- *Search control*: can the strategy selected be modified dependent on the problem represented, or even possibly based on a given problem instance or dynamically based on problem characteristics?
- *Consistency propagation* : as a precursor to search or in conjunction with search, it often appears to be sensible and beneficial to "propagate" constraints and limit one domain size based on a previous variable domain value selection. Typically, these propagations are done based on problem-specific characteristics (heuristics), or CSP-specific heuristics (such as domain size).
- *Hybridization* : While both use and specification of search strategy and consistency propagation are important, many problems are best expressed through an interleaving or "hybrid" of the two methods. In fact, it has been claimed (Nadel 1989) that all search strategies can be expressed as an interleaving of some amount of in-search constraint propagation and standard backtracking. From this viewpoint, Foward Checking (FC), LookAhead (LA), Partial LookAhead (PLA), and "intelligent" BT such as BackJumping (BJ) and BackMarking (BM) are all essentially primitive BT with a certain amount of in-node constraint propagation occuring before expansion of the search space to another level. Consequently, construction of interesting or novel search strategies depends upon being able to control (possibly dynamically) the amount of effort spent on consistency propagation at the expense of backtracking throughout the search space.
- *Domain dependent heuristics* are basically specifications about a problem aspect that makes the problem "easier to solve" in some sense. It is imperative that any specification language and solution strategy be powerful enough to support heuristics.
- *Domain independent heuristics* are specifications about CSPs or alternative ways of viewing CSP characteristics, such that this view makes CSPs in general "easier to solve". Using these types of heuristics essentially allows the user to adjust a particular generic strategy into a new one not initially provided in the language. While a "naive"

developer may not wish to make use of this type of power, it is essential for the CSP researcher and application developer.

Many languages strive to provide solutions that function as black boxes and allow the language users to "get by" without "seeing" or being forced to learn all of the details the language supports. In fact, for a language to be well-accepted by a wide application-intensive audience, this is likely a prerequisite. However, this goal is often directly in opposition to those who wish to make full use of the features of a language.

Typically, real applications of CSPs have been implemented on a case-by-case basis with all of the software development problems of medium sized software constructed from scratch for a single use. The applications have been overly specific and difficult to modify. Identical work has been repeated ad nauseum in the construction of search and consistency algorithms. Specific domain constraint algorithms have had to be constructed from scratch and integrated into a single framework. These factors are consistent with the world that has been software development. Generally when an application area seems to be widespread and generalizable enough to justify (perhaps financially) the existence of a distributed toolset, one or more emerge and become accepted (take for example the various window manager software for X). The questions for the case of CSP solvers are many. What should go in the toolset? What model should be the basis for arranging the various tools in the toolset? How can constraints be "best" expressed in such a toolset? What does "best" mean - is it "ease of representation", or perhaps "flexibility of representation"? So, for imperative types of languages, a toolset could conceivably provide the "genericness" that will stave off repeated implementations of the same algorithms and also provide a "sensible" framework for constraint solving applications.

But what other possibilities might provide a similar solution framework? Logic Programming "languages" with extensions (CCLP) have been suggested, and in the next section of this paper, we will examine CCLP specifics, and attempt to determine what level of service CCLP provides, and whether this is adequate or desirable for either the researcher or application developer in the domain of combinatorial constraint based problems.

4 Extended Logic Programming: a blackbox alternative

So far we have described the "traditional" approach to solving constraint satisfaction problems and have outlined a "typical" real domain with real world application. The "traditional" approach provides a set of tools for the research or application developer that basically serve as a starting point for the construction of efficient problem solvers in one domain or possibly in a broader class of domains. Certainly many of the applications studied have suggested that certain strategies perform better than others, however, little headway has been made to date in explicitly categorizing these domains in terms of which strategy suits which domain and why. Identification of critical domain characteristics that suggest particular solution strategies is an active and interesting research area today.

4.1 Logic Programming

Van Hentenryck (1989) states "Logic Programming is based on the idea that *computation can be seen as controlled deduction ...*". In logic programming, a subset of first-order logic known as Horn claus logic was interpreted procedurally in a nondeterministic (i.e. backtracking based) recursive language that we know today as Prolog. Prolog corresponds to an efficient implementation of a particular sound and complete proof procedure for definite clauses.

Prolog, as representative of logic programming languages, has a relational form. Consequently, it provides a convenient way to state constraints which basically describe relationships. Since nondeterminism is embedded in Prolog, the Prolog user can be naive of the details of how this is carried out, and can essentially obtain complete solutions based on simple problem statements while remaining free from the tedium of programming and maintaining this functionality. Thus, logic programming languages such as Prolog provide:

1. A relational method of stating constraints between the arguments of a predicate that is at the root of the language.
2. Nondeterminism implicit in the language.

At first glance then, Prolog appears a natural for representing and implementing constraint problems, and a good conceptual tool for discrete combinatorial problem expression and solution. The obvious explanation of why this is not the case is the complexity of even a moderately sized constraint problem. The sheer combinatoric nature of the possible number of solutions given moderate domain sizes over a variable set results in a large search space. Horrid efficiency problems result when a generate-and-test type of approach is used, which simply generates combinations of variables and domain values and evaluated them as candidate problem solutions. A "perfect" or total solution to a CSP is an assignment from the domain values for each variable to each of the variables template slots. In one simplified problem that corresponds to the STR problem, each of n variables are to be assigned from a pool of m possible domain values, with the ordering of the selection corresponding to an assignment to the variables. There are m choose n candidate solutions for our set, that is, the number of n-combinations of

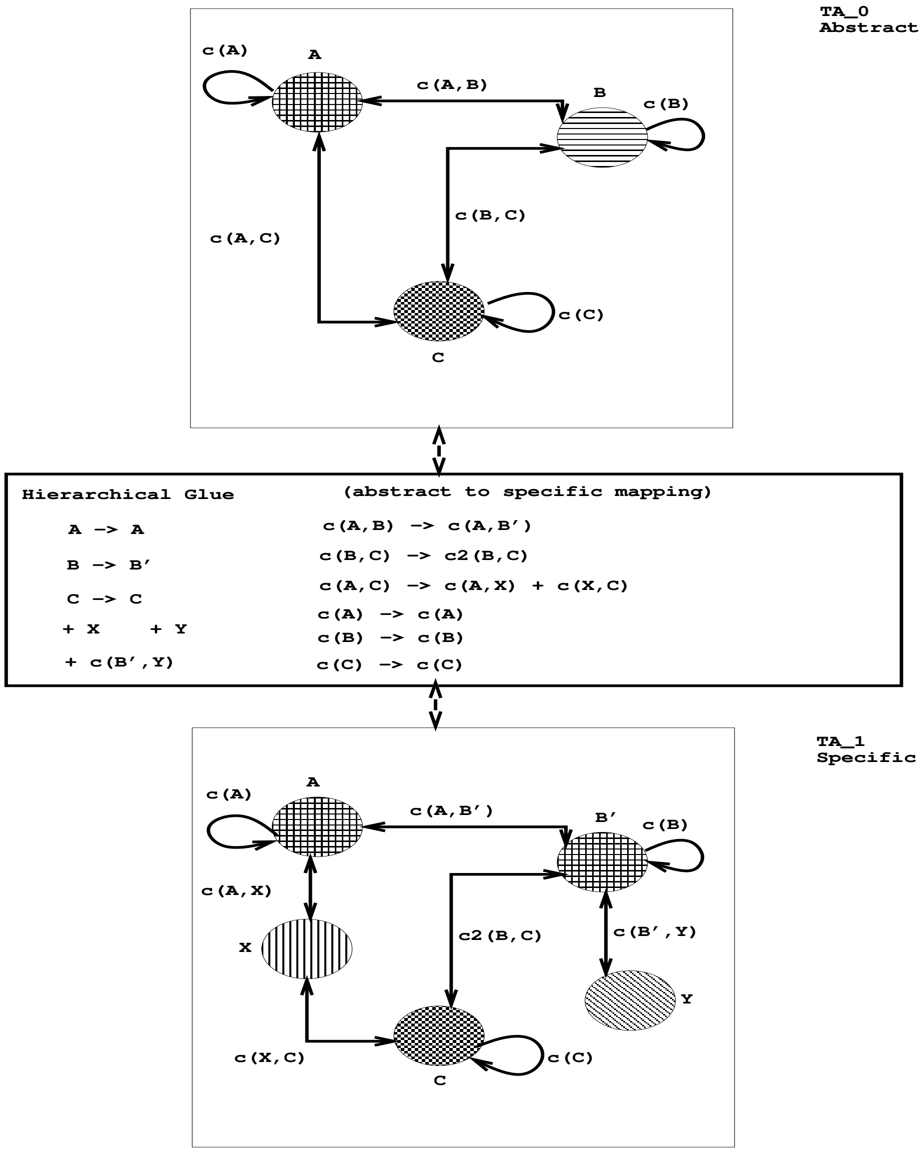

Figure 2: Explicit expression of representation change.

an m-set:

$$NumberCandidateSolutions = m!/n!(n-m)! \quad (1)$$

Thus there are some number of solutions exponential in the size of the number of possible domain values, m. Note that in many problems, including STR, the number of domain values typically far dominates the number of variables. Naively generating and testing these possible solutions is not going to be a good idea, even if we attempt it through a marginally better approach such as backtracking. As pointed out in Van Hentenryck (1989), this type of approach is dedicated to "recovering" from a failed assignment of value to variable via backtracking, rather than "avoiding" such an occurrence without blindly wandering into it.

Van Hentenryck (1989) outlines the design of Prolog from a position where an algorithm is defined as $algorithm = logic + control$, where *logic* refers to the facts and rules specifying what the algorithm does, and control refers to how the algorithm can be implemented by applying the rules in a particular order[3]. Extensions or improvements to logic programming languages such as Prolog are typically based on the preservation of the *logic* aspect, and focus on refinement of *control*. Essentially this approach results in a situation where the generate and test "problem conception" can be maintained, with the supposed simplicity and ease of use this suggests, and yet the efficiency of improved backtracking can be enjoyed.

Assuming an underlying Generate and Test scheme, control in Prolog is based on the *Prolog Computation Rule (PCR)* which determines two things:

- The order in which goals will be attempted to be satisfied, where the goals are the elements of a query or resolvent.
- The order in which rules will be applied, where the rules are the facts in the database of facts and rules.

If the PCR were changed somewhat so that:

1. if constraints of the form of disequations[4] were taken as goals only if they are ground,
2. as soon in the resolution process as the goal set or resolvent contained some ground disequations, one is selected as the current goal

then the way in which Prolog searched would be based on Generate and Test, but would behave as Backtracking since selection of a *constrained to fail* variable domain value would immediately signal a dead end and a backtrack point.

In fact, Prolog implementations frequently make use of more complicated search strategies known as "Intelligent Backtracking". These strategies basically attempt to reason in a limited way about the "failure cause". On this basis, they backtrack to a previous variable instantiation that is conflicting, via some constraint, with the current instantiating variable. They do not blindly backtrack to some intermediate variable that may not be involved with the constraint, as is the case with standard Backtracking. Van Hentenryck (1989) points out that while this technique can be an improvement, there are theoretical limitations that imply the necessity of using heuristics for finding the correct backtracking points, and practical limitations as there may be significant overhead in locating the backtrack point. Further, it is observed that it is better to try to avoid these failures rather than to react to them in some manner after the fact. Towards a more systematic approach, Lookahead search schemes are suggested as a better way of reducing the search space *a priori*. In fact, a flexible form of this systematic reduction of the space is precisely what has been discussed in the context of our previous requirement of *Hybridization* for effective CSP formulation. In Section 4.3 we shall examine more precisely this extension of logic programming to see to what extend the hybridization requirement is met.

In our previous discussion of Section 3.2, consistency techniques were introduced which reduce the search space prior to the explicit discovery of a failure in a backtracking search approach. Spending work during search in removing variable domain values on the basis of certain local constraints is essentially preventative work - preventing a dead end before encountering it during standard backtracking. Van Hentenryck (1989) points out quite correctly that the value of this preventative work has been shown in many application domains[5] to produce substantial improvement over standard backtracking. Further, it is stated that it should be considered a fundamental primitive in CSP solution.

It would seem a natural extension to a logic programming language to provide consistency propagation tools as a basic part of a new language. Basically, that is the purpose of the work described in Van Hentenryck (1989). In this work, the intent is to provide consistency techniques *inside* logic programming so as not to obscure the descriptiveness that supports constraint expression. It is observed that logic programs can be written that support these techniques explicitly but that this approach "complicates" programs (surely they can be no less complicated than a standard procedural implementation and quite likely more difficult to understand), and are very inefficient.

4.2 From Logic Programming to what ?

Surely the construction of embedded consistency techniques in a search-based language with formal de-

[3]Based on Kowalski's division of labor described briefly in Sethi (1990).

[4]Constraints that state that certain variables cannot take on the same value.

[5]Experiments typical of those described in Nadel (1989) support this statement.

scriptive qualites is an important goal, but what form should this embedding take? Much research is being undertaken *currently* (Prosser 1993, van Run 1994, Woods 1993*a*, Woods 1993*c*, Yang & Fong 1992) on search strategies that are appropriate for CSPs in various domains, according to specific properties that have been observed in CSPs. New consistency propagation techniques are being developed and old ones improved on constantly as the nature of CSP problems in specific and general become better understood. Interesting work (Nadel 1989, Prosser 1993, van Run 1994) has shown that re-combination of search and consistency propagation directly results in unique and often more efficient search strategies, particularly in specific domains. Certain domains seem to benefit with more in-search propagation while others from less. Other work has attempted to view constraint propagation and search from a single perspective. In this view, the search space may be reduced at any time through constraint propagation techniques, and that portion explored via search is viewed as part of a single problem space that is navigated with a dynamically changing search procedure (Nadel 1989). Work such as Nadel (1989) and more recently Prosser (1993) have attempted to make generalizations about certain problem properties and relation on search space and search performance, however, there does not current exist a theory of knowledge structure and search that suggests definite causal links between problem structure and search performance.

We have seen the "naturalness"[6] with which logic programming seems to be capable of representing relationships such as constraints. Is this "natural approach" sufficient to meet our previously stated requirement for *expressiveness*? Since it has been understood for sometime (indicatively demonstrated for CSPs in Nadel (1990)), that problem representation can have a dramatic effect on the complexity of problem solution both formally and in practice, do the facts support the formal declarative qualities of logic programming as being "rich" enough to support alternative representations and approaches? For example, the work of Freuder & Wallace (1992), Yang & Fong (1992), and most recently Woods (1993*a*) has indicated that representing constraint problems in a hierarchical structure is potentially valuable in terms of problem solution time. Further, this representation structure, as shown in Figure 2, may provide more than an ordering heuristic for variable binding in that an explicit mapping between subsequent hierarchical levels of the CSP is in effect supplying additional problem information that would be lacking in a simple declaration about variables and variable domains, even taking into account the semantics of constraints (Holte, Zimmer & MacDonald 1992, Holte, Zimmer & MacDonald 1993, Holte 1993). We have seen the logic programming paradigm of algorithms defined as the sum of *separately defined* logic and control where control basically determines the way in which the logic is applied. However, abstracted problem representations may be utilized differently. The "logic" represented by the problem specification has a structure that may be exploited explicitly by the control, depending on problem instance. The "logic" of encoded connections between abstraction levels may actually serve as problem solution control information as well.

The intent in raising the issues of search and consistency technique integration and CSP representation issues is to point out that logic programming languages provide both more and less than standard imperative or functional programming languages in terms of tools for constructing algorithms. Logic programming languages provide an additional "assumed" formal representation style that must be acknowledged. However, while formalism restricts and standardizes problem representation, it can result in oversimpliflication and loss of important problem content. Additionally, since logic programming has important in-built algorithmic underpinnings, traditional declarative and functional programming languages provide little in the way of inherent "high level" algorithmic restrictions. Rather, these languages tend to support or provide these "advanced" algorithmic features through commercially or publicly available software libraries of toolsets. These toolsets must be directly manipulated by the program or application developer, and tool characteristics can be explicitly used or ignored as desired.

It is easy to imagine a situation where a particular language embeds a single multi-faceted sorting procedure, and advertises to the language user this procedure as a generic "all-cases-encompassing" sorting tool. It is conceivable that this tool includes heuristics that are capable in many situations of adequately determining a search strategy based on input dataset characteristics. However, the question of all-case appropriateness would seem to be better left to someone familiar with the type of input or problem semantics than to heuristics that are at best partially effective, at least in the absence of a more encompassing theory about data sortedness and sorting algorithm applicability.

[6] Van Hentenryck (1989) is quite fond of describing the *natural* link between logic programming representation with relations (as in Prolog) and formulating constraints. This claim is apparently based on two things: (1) the author is exceedingly familiar with the realm of logic programming, and therefore the logic programming paradigm is "natural", and (2) relations are a common feature to domain experts who would commonly wish to specify such things for a machine to solve, and consequently fit into some "typical" expert's world-view. It immediately comes to mind that the "natural" way of representation (or of doing anything else, for that matter) is not necessarily the "most beneficial", "most efficient", or "most effective" way.

Standard logic programming languages embed a control strategy based upon one type of backtracking. This search strategy may be one of many "intelligent" strategies that have been shown to perform well in various cases, however, there is no existing theory that will map problem representation accurately to particular search strategy performance. We must be concerned with determining whether logic programming extended with constraint processing and modified control structures still falls into trap of limiting the developer unwarrantedly. Can the backtracking (albeit "intelligent" backtracking) incorporate consistency application, and consequently easily benefit from the wealth of information known about constraint propagation methods?

As we examine this extended version of logic programming we must attempt to keep in mind the primary issue of determining whether or not this extended language can provide the *expressiveness*, *search control*, *consistency propagation*, *hybridization*, and incorporation of *heuristics*, early stated in Section 3.2 as necessary componenets of any set of tools intended to construct solutions to Constraint Satisfaction Problems.

4.3 Logic Programming Extended

In this section we will describe the extensions to logic programming described explicitly in Van Hentenryck (1989) as they relate to the embedding of consistency propagation techniques, and explain how this new language is able to handle many constraint problems including STR with a very simple encoding. Essentially extended logic programming is providing a toolset that is capable of propagation of constraints to reduce domains during search and relies on three detailed search strategies that have been shown to be successful across many practical problems[7].

The primary extensions are:

1. **Predicate domain declarations** or constraints which restrict the range of the arguments of predicate's variables, i.e. domain variables.
2. **General (or weak) methods** for controlling the inference process in the absence of much domain-specific control information. Specifically, these methods result in the use of particular inference rules with specific constraints. Essentially, three types of declarations are potentially introduced, each which entails the creation of a different control rule. These declarations are:
    - **Forward Declarations**, which define *for each predicate* that the predicate is a constraint, and that forward checking inference may be applied for this predicate. Predicates or constraints are defined so that the last variable V_n (or forward variable) in a predicate $P(V_1...V_n)$ has its domain reduced when the first $n-1$ variables in P are instantiated to ground. In this way, the domain of V_n is explicitly reduced during search, and the values discarded cannot be used again as values for V_n. The ground value of V_n is determined only when there is precisely one domain value remaining for V_n. These declarations support the construction of a Forward Checking (FC) search strategy in logic programming.
    - **Lookahead Declarations**, basically generalization of Forward Declarations, where when P is a constraint, at least one of the variables of the constraint is a domain-variable. For these predicates, instantiation of the ground variables results in a reduction of the space of the domain variables, possibly resulting in a reduction of one or more of them. This is described in Van Hentenryck (1989) as a general mechanism for enforcing n-consistency between the lookahead-variables.
    - **Partial Lookahead Declarations**, which are applicable in the same cases as the Lookahead Declarations, but which are basically a cross between Forward and Lookahead, where some of the domain values are reduced based on certain specific partial variable groundings in the P predicate/constraint.

Of critical importance to this paper is the fact that Lookahead may be interpreted as generalization of AC-3 and AC-4 consistency propagation algorithms as described in Mackworth (1977), Mackworth (1987) and elsewhere. Van Hentenryck (1989) observes that both techniques have advantages and disadvantages *dependent* on the problem in particular, and the choice is best left as a parameter to the user. This observation is *critical* since these two consistency techniques are only but two representatives of a continuous string of newly emerging algorithms (of both global and local applicability). Therefore this selection of two representatives is woefully inadequate, and their combination into search is insufficiently flexible to provide a full range of capability of the hybrid search algorithm. Essentially the search underlying this extended language is hybrid, but only a restricted subset of hybrids can be represented, and worst of all, there is no way to extend this coverage without further exten-

[7] These domains include graph coloring, scheduling with constraints, and several other optimization problems.

sion to the language.

3. **Specific (or strong) methods** applicable when specific information about a constraint is known explicitly in advance. Specific methods achieve the same pruning as is the case for general methods, however, are typically specializations of the general methods for particular constraints and domains. One obvious example of a specific is arithmetic rules in a mathematical domain where the instantiation of two variables might imply a new range on a third that is specified by some arithmetical combination of the first two.
4. **Choice methods** governing the selection of value ordering for a particular variable, and variable selection for instantiation. The following are supported[8]:
   - **Variable Choice**, based on the first-fail principle where we wish to try variables that are likely to fail first, essentially trying the most difficult part of the search space first. Thus, first-fail bases the selection on the variable that is the most constrained in terms of the smallest domain. Since the extended logic language is maintaining explicit domain ranges, this is possible where traditional logic languages had no equivalent explicit representation. Another provided optional heuristic improvement is choosing the variable (in case of domain size ties) that will have to satisfy the most constraints.
   - **Constraints as Choices**, constraints on the search space may take the form not only of assumed variable bindings from a domain, but also assumed constraint results without a constraint evaluation where one is not possible, perhaps due to insufficient binding information. This constraint assumption becomes a backtrack point just as a variable instantiation point[9].
5. **Optimization methods**, including branch and bound search that supports min/max types of problems based on problem splitting coupled with two options for search: best-first and depth-first.

4.4 What extended logic programmin provides

Logic programming languages are well known and wide-spread. Consequently, it is reasonable to assume that many programmers and researchers are familiar with and comfortable with the representation style and behaviour of these languages, and will follow the conceptual jump to improved search behaviour and constraint support in such languages. The question of whether they provide adequate expressiveness appears to have been overlooked in favor of "familiarity" and convenience.

Use of extended logic programming for the representation and solution of constraint satisfaction problems is certainly going to be more efficient than was the case for standard logic programming for the simple reason that the extensions have made use of search procedures and heuristics that have been demonstrated in specific experimental work in the *imperative* world of problem solution, where search algorithms are coded and tested across various domains. Earlier logic programming backtracking and *a posteriori* constraint checking has been abandoned in the rest of the constraint satisfation world for many years in other than toy domains.

It is certainly true that provision of a standardized approach and toolset for the representation and solution of CSPs is beneficial, but extension of logic programming has accomplished this only in part, and from the point of view of encompassing the richness and flexibility of hybirdized search, it is not clear at all what justification can be given for limiting to just three control strategy basics. Standardization must creep in in any tool that hopes to be widely used, however, the overlimitation of these design assumptions appears to be only a product of the language used (logic programming) and motivated by little else.

4.5 What is missing in extended Logic Programming?

1. Representation Issues

   New languages derived from logic programming languages such as Prolog which retain the "relational" based problem description are limited in that richer representation schemes such as abstraction do not appear to fit cleanly into the language approach. Other work has been discussed that makes use of this additional expressivness to model problem characteristics such as explicit expert problem decomposition into parts or into a hierarchy of abstraction based specifications. It seems that the simplicity that lends the attractive intuitive "naturalness" of constraint specification as relations may actually be a strong restriction to problem description.
2. Control Issues

   Logic programming intentionally draws a hard boundary between problem specification or *logic* and problem solution or *control*. Recent work has shown decisively that problem representation can not only be exploited in control or search

[8] Notably missing from this list is domain value ordering, an often discussed search heuristic, that may have easy application in some domains (such as the case where a smaller domain value may be desirable).

[9] This is a very interesting extension that has not to my knowledge been explored very deeply to date in AI/Search research, at least not experimentally with CSPs.

strategies, but that richer problem representation languages that can account for abstration or problem decomposition can achieve great performance improvements specifically by crossing this boundary, and allowing representation to guide control. It is interesting to note that, in a sense, the control specification inherent in these richer schemes is exploited on the basis of other "naturalness" arguments such as "problems are more naturally represented as hierarchies of increasing complexity", or "problems are more naturally represented as a sum or their parts". The effectiveness of divide-solve-combine algorithms in many other areas of computer science is another example of this type of thinking. Decoupling of control and representation may be "clean" but it doesn't appear to be "smart".

Extended logic programming as described in Van Hentenryck (1989) extends traditional logic programming primarily by improving on the control rule underlying logic programming languages to take advantage of three "better" control rules based on Forward Checking search, Partial Lookahead search, and Lookahead search. These strategies attempt to reduce the search space through domain reduction during search *a priori* and consequently may be identified as three discrete combinations of search and consistency propagation selected from a continuous world of possible combinations of search and consistency algorithms. While many domains show dramatic improvement with the use of these search algorithms in place of standard "intelligent" backtracking, this author is unaware of any theory which would suggest that these three are definitively superior to other search strategies, or represent some "best merge" of consistency and propagation with respect to the types of problems they will be applied against. Consequently, it appears that limiting the degree of flexibility of search to these three control rules is unnecessarily limiting in general.

4.6 Conclusions

Logic programming is unnecessarily limited in terms of both representation and division between logic and control, possibly at the expense of both performance and accuracy of problem representation. In conclusion, the extension of logic programming with search strategies that embody a higher degree of in-search consistency propagation will provide a language that may be applied to domains of increased complexity. However, since there is no clear break-point between research and application that suggests these strategies as "superior" in some application sense, it seems arbitrary to limit to these only, when perhaps a more general control structure with a flexible approach to consistency propagation in-search would be more appropriate.

It is true, that extended logic programming as described in Van Hentenryck (1989) is now capable of addressing several real domains. It provides a toolset that includes the primary sources of efficiency demonstrated by earlier *imperative* application testing such as the AC-3 and AC-4 arc-consistency algorithms coupled at least partially flexibly with a search engine capable of accomodating several types of search heuristics.

Additionally, the representation expressiveness appears to be sufficient to achieve reasonable results across many common CSP domains, and with relative ease of programming since the underlying control and algorithmic necessities of such a system are effectively hidden from the user. However, these same features and functionality can be observed in library toolsets that provide AC algorithms and search procedures in parameterized and connected fashion along with a language capable of easily expressing constraints. Since such a toolset is not known to exist at present, extended logic programming languages seem to be the easiest and best available tool, however, their lack of flexibility and ability to include extended algorithms in place of the ones ensconsed in the language will preclude their longevity in such a dynamic and growing area.

References

Bacchus, F. & Yang, Q. (1991), The downward solution property, *in* 'Proceedings of the 12th IJCAI', Sydney, Australia, pp. 286–292.

Bacchus, F. & Yang, Q. (1992), Downward refinement and the efficiency of hierarchical problem solving, personal correspondence.

Dechter, A. & Dechter, R. (1987), Removing redundancies in constraint networks, *in* 'Proceedings of the 6th AAAI', pp. 105–109.

Dechter, R. (1990), From local to global consistency, *in* 'Eighth Canadian Conference on Artificial Intelligence'.

Dechter, R. (1992), 'From local to global consistency', *Artificial Intelligence* **55**, 87–107.

Dube, T. & Yap, C.-K. (n.d.), The geometry in constraint logic programs, c/o Dominic Duggan, cs642.

Freuder, E. & Wallace, J. (1992), 'Partial constraint satisfaction', *Artificial Intelligence* **58**, 21–70.

Freuder, E. C. (1991), Eliminating interchangeable values in constraint satisfaction problems, *in* 'Proceedings of the 9th AAAI', Vol. 2, pp. 227–233.

Holte, R. C. (1993), Automatic change of representation, personal correspondence.

Holte, R., Zimmer, R. & MacDonald, A. (1992), When does changing representation improve problem-solving performance ?, *in* 'Proceedings of the Workshop on Change or Representation and Problem Reformulation'. NASA Ames Technical Report FIA-92-06.

Holte, R., Zimmer, R. & MacDonald, A. (1993), A study of the representation-dependency of abstraction techniques, *in* 'Proceedings of the 1993 Workshop on Knowledge Compilation and Speedup Learning'.

Kasif, S. & Delcher, A. (n.d.), Local consistency in parallel constraint-satisfaction networks, c/o Dominic Duggan, cs642.

Kumar, V. (1992), 'Algorithms for constraint-satisfaction problems', *AI Magazine* pp. 32–44.

Lapointe, S. & Proulx, R. (1994), Statistical identification of templates in message data, Contract Report W7701-3-0847/01-XSK, APG inc. (1994). Combat Intelligence Automation Group, Defence Research Establishment Valcartier.

Mackworth, A. (1977), 'Consistency in networks of relations', *Artificial Intelligence* **8**, 99–118.

Mackworth, A. (1981), Consistency in networks of relations, *in* Webber & Nilsson, eds, 'Readings in Artificial Intelligence', Morgan Kaufmann Publishers Inc., pp. 69–78.

Mackworth, A. (1987), Constraint satisfaction, *in* S. Shaprio, ed., 'Encylopedia of Artificial Intelligence', Vol. 1, John Wiley and Sons, pp. 205–211.

Mackworth, A. (1992), 'The logic of constraint satisfaction', *Artificial Intelligence* **58**, 3–20.

Mackworth, A. & Freuder, E. (1985), 'The complexity of some polynomial network consistency algorithms for constraint satisfaction problems', *Artificial Intelligence* **125**, 65–74.

Mackworth, A. & Freuder, E. (1993), 'The complexity of constraint satisfaction revistited', *Artificial Intelligence* **59**, 57–62.

Menezes, F., Barahona, P. & Codognet, P. (n.d.), An incremental hierarchical constraint solver, c/o Dominic Duggan, cs642.

Minton, S., Johnston, M., Philips, A. & Laird, P. (1992), 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems', *Artificial Intelligence* **58**, 161–205.

Montanari, U. & Rossi, F. (n.d.), Constraint satisfaction, constraint programming, and concurrency, Provided by Dominic Duggan for cs642.

Nadel, B. (1990), 'Representation selection for constraint satisfaction: a case study using n-queens', *IEEE Expert* pp. 16–23.

Nadel, B. A. (1989), 'Constraint satisfaction algorithms', *Computational Intelligence* **5**, 188–224.

Prosser, P. (1993), 'Hybrid algorithms for the constraint satisfaction problem', *Computational Intelligence* **9**(3), 268–299.

Sethi, R. (1990), *Programming Languages*, Addison-Wesley.

Tolba, H., Charpillet, F. & Haton, J.-P. (1991), Representing and propagating constraints in temporal reasoning, *in* 'Proceedings of the International Conference on Tools for Artificial Intelligence', pp. 181–185.

Van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, The MIT Press.

Van Hentenryck, P., Saraswat, V. & Deville, Y. (unknown), 'Constraint processing in cc(fd)', *unknown*.

van Run, P. (1994), Domain independent heuristics in hybrid algorithms for csps, Master's thesis, University of Waterloo.

Waltz, D. (1975), Understanding line drawings of scenes with shadows, *in* P. Winston, ed., 'The Psychology of Computer Vision', McGraw Hill, Cambridge, Massachusets, pp. 19–91.

Woods, S. (1993*a*), Interactive recognition of spatially defined model deployment templates, Research memorandum, Combat Intelligence Automation Group, Defence Research Establishment Valcartier, Courcelette, Quebec CANADA. unclassified.

Woods, S. (1993*b*), A method of interactive recognition of spatially defined model deployment templates using abstraction, *in* 'Proceedings of the 1993 DND Workshop on Advanced Technologies in Knowledge Based Systems and Robotics'.

Woods, S. (1993*c*), Upward exploitation of constraint propagation in hybrid constraint satisfaction search., Defence Research Establishment Valcartier, Dept. of Defence, Courcelette, Quebec, CANADA. unclassified.

Yang, Q. & Fong, P. (1992), Solving partial constraint satisfaction problems using local search and abstraction, Technical Report cs-92-50, University of Waterloo.