Strategic Directions in Constraint Programming

PASCAL VAN HENTENRYCK

Brown University, Providence, RI ⟨pvh@cs.brown.edu⟩

VIJAY SARASWAT ET AL.[1]

AT&T Research, 600 Mountain Avenue 2A-430, Murray Hill, NJ 07974

1. INTRODUCTION

A *constraint* can be thought of intuitively as a restriction on a space of possibilities. Mathematical constraints are precisely specifiable relations among several unknowns (or *variables*), each taking a value in a given domain. Constraints restrict the possible values that variables can take, representing some (partial) information about the variables of interest. For instance, "The second side of a sheet of a paper must be imaged 9000 milliseconds after the time at which the first side is imaged," relates two variables without precisely specifying the values they must take. One can think of such a constraint as standing for (a possibly infinite) *set* of values, in this case the set {⟨0, 9000⟩, ⟨1500, 10500⟩, . . .}.

Constraints arise naturally in most areas of human endeavor. They are the natural medium of expression for formalizing regularities that underlie the computational and (natural or designed) physical worlds and their mathematical abstractions, with a rich tradition going back to the days of Euclidean geometry, if not earlier. For instance, the three angles of a triangle sum to 180 degrees; the four bases that make up DNA strands can only combine in particular orders; the sum of the currents flowing into a node must equal zero; the trusses supporting a bridge can only carry a certain static and dynamic load; the pressure, volume, and temperature of an enclosed gas must obey the "gas law"; Mary, John, and Susan must have different offices; the relative position of the scroller in the window scroll-bar must reflect the position of the current text in the underlying document; the derivative of a function is positive at zero; the function is monotone in its first argument, and so on. Indeed, whole subfields of mathematics (e.g., theory of Diophantine equations, group theory) and many celebrated conjectures of mathematics (e.g., Fermat's Last Theorem) deal with whether certain constraints are satisfiable.

Constraints naturally enjoy several interesting properties. First, as previously remarked, constraints may specify partial information—a constraint need not uniquely specify the value of its variables. Second, they are additive: given a constraint c_1, say, $X + Y \geq Z$, another constraint c_2 can be added, say, $X + Y \leq Z$. The order of imposition of constraints does not matter; all that matters at the end is that the conjunction of constraints is in effect. Third,

[1] Contributors to this article include Alan Borning, Alex Brodsky, Philippe Codognet, Rina Dechter, Mehmet Dincbas, Eugene Freuder, Manuel Hermenegildo, Joxan Jaffar, Simon Kasif, Jean-Louis Lassez, David McAllester, Ken McAloon, Alan Mackworth, Ugo Montanari, William Older, Jean-Francois Puget, Raghu Ramakrishnan, Francesca Rossi, Gert Smolka, and Ralph Wachter.

constraints are rarely independent; for instance, once c_1 and c_2 are imposed it is the case that the constraint $X + Y = Z$ is entailed. Fourth, they are nondirectional: typically a constraint on (say) three variables X, Y, Z can be used to infer a constraint on X given constraints on Y and Z, or a constraint on Y given constraints on X and Z, and so on. Fifth, they are declarative: typically they specify what relationship must hold without specifying a computational procedure to enforce that relationship. Any computational system dealing with constraints must fundamentally take these properties into account.

Constraint programming (CP) is the study of computational systems based on constraints. It represents a harnessing of the centuries-old notions of analysis and inference in mathematical structures with several modern concerns: general languages for computational representation, efficiency of analysis and implementation, and tolerance for useful (albeit incomplete) algorithms (tied perhaps to "weak" methods such as search), all in the service of design and implementation of systems for programming, modeling, and problem-solving in different domains. As discussed in the next section, work in this area can be traced back to research in artificial intelligence and computer graphics in the 1960s and 1970s that focused on explicitly representing and manipulating constraints in computational systems. Only in the last decade, however, has there emerged a growing realization that these ideas provide the basis for a powerful approach to programming, modeling, and problem solving, and that different efforts to exploit these ideas can be unified under a common conceptual and practical framework.

The basic essence of this framework is the separation of concerns into levels. The first level is that of very generally defined *constraint systems*—systems of inference with pieces of partial information based on such fundamental operations as constraint propagation, entailment, satisfaction, normalization, and optimization. In addition to the traditional constraint systems that have already been investigated over centuries (such as over the real numbers, integers modulo p), CP focuses on a wide variety of systems (arising often from application concerns) ranging from "unstructured" finite domains to equations over trees (term-unification) to temporal intervals. Increasing attention is being paid to discovering efficient techniques for performing these constraint operations across wide classes of such constraint systems and to discovering common exploitable structures across constraint systems.

Operating around this level is the second *programming-language* level, which allows the user to specify more information about which constraints should be generated, how they should be combined and processed, and so on. Perhaps unique to CP are modeling languages that exploit logic-based control constructs [e.g., constraint logic programming (CLP) or concurrent constraint programming (CCP)]. These languages interact with the first level purely via the basic constraint operations. This provides the user with a very expressive framework (parametric in the underlying constraint system) for generating, manipulating, and testing constraints, while (in the case of the logic-based languages) preserving their declarative character. This realization of unified frameworks has simultaneously been accompanied by the implementation of several general systems that are finding widespread use in applications as diverse as modeling physical systems and controlling robots to scheduling container ships in harbors.

This central organizational idea has many ramifications. What emerges is a general declarative framework potentially more promising than either full first-order logic (which is expressive, but undecidable in theory and usually inefficient in practice) or restricted versions such as the Horn clause subset that underlie logic programming (which are usually efficient in practice, but not

expressive enough for many applications). For what is fundamentally acknowledged is that different computational techniques (constraint-solving algorithms) will be useful in different computational contexts—and a uniform scheme is provided for integrating these techniques into a powerful computational framework. For the theoretician, metatheorems can be proved (and analysis techniques invented) once and for all that apply to an infinite family of systems; for the implementer, different constructs (backward chaining, backtracking, suspension) can be implemented once and for all; for the user, only one set of ideas needs to be understood, although with rich (albeit disciplined) variation (via constraint systems).[2]

Today CP is contributing exciting new research directions in a number of distinct areas, such as artificial intelligence (natural language understanding, scheduling, planning, configuration, etc.), concurrent computing, database systems, graphical interfaces, hardware verification, operations research and combinatorial optimization, programming language design and implementation, reactive systems, and symbolic computing algorithms and systems. The field is being driven both by a need for internal organization and structure and by the demands of the increasingly sophisticated real-world applications to which it is being applied.

The state of the art in CP is reported in international conferences on Principles and Practice of Constraint Programming (PPCP) [Montanari and Rossi 1995b; Freuder 1986] and Practical Applications of Constraint Technology (PACT), and in the recently established journal, *Constraints*. Work continues to be reported in the conferences and journals of related areas such as artificial intelligence, logic programming, databases, and operations research. Interested readers may find related surveys in Van Hentenryck [1991], Frühwirth et al. [1992], and Jaffar and Maher [1994].

The rest of this paper is organized as follows. First we develop some background on the origin of constraint programming. The state of the art in the application of constraint ideas in various fields is then discussed. Finally we identify some key strategic directions for further development.

2. THE ORIGINS OF CONSTRAINT PROGRAMMING

Some of the earliest ideas leading to CP may be found in the artificial intelligence (AI) area of constraint satisfaction, dating back to the 1960s and 1970s. The pioneering works on networks of constraints were motivated mainly by problems arising in the field of picture processing [Montanari 1970; Waltz 1975]. In these works, constraints were explicitly represented as binary compatibility matrices and the goal was to develop efficient polynomial algorithms that could discover incompatibilities by looking at just a few constraints. This can greatly speed up the subsequent phase in which one or all solutions are to be found via backtracking. In picture processing, these algorithms sometimes eliminated most infeasible picture interpretations, for example, those allowed by each constraint alone but not by a conjunction of a small subset. In some cases this phase results in just one (the only one) alternative being left, thus eliminating backtracking completely [Waltz 1975]. The main algorithms developed in those years were related to achieving (variations of) arc- or path-consistency

[2] From a methodological point of view, it is important to realize that not all researchers in CP work across both of these levels. Some prefer to exploit the unifying framework of constraints while working purely within the first level of constraint systems, considering issues around programming to be orthogonal to their concerns. Others exploit the unifying framework of constraints to develop programming language notions, while not paying attention to the properties of particular constraint systems. Some focus on fruitfully exploiting the synergy across the boundary between the two levels.

[Montanari 1970; Mackworth 1977; Mackworth and Freuder 1985] (see Section 3.1). The former finds (and eliminates) values from variables' domains that are incompatible with some constraint concerning that variable, whereas the latter eliminates pairs of values that are allowed according to a given constraint c but not if one looks at a chain (a path) of constraints starting and ending at the same points as c. In other words, one can say that these algorithms propagate the information given by one constraint to other constraints.

In these systems, there was still no notion of constraint programming; rather, the problem was modeled directly via sets of constraints that were solved using an algorithm. (Mention must also be made of the remarkably prescient systems REF-ARF [Fikes 1968] and ALICE [Lauriere 1978]. Both provided simple but very useful constraint languages for specifying search problems, and solved them using customized constraint solvers with embedded propagation and search techniques.) However, we see later that many constraint-based computational frameworks counted on these algorithms and results to achieve simple and efficient implementations.

Early application areas for constraints were interactive graphics and circuit modeling and diagnosis. The first of these systems was Ivan Sutherland's [1963] Sketchpad, developed in the early 1960s. Sketchpad was an interactive graphics application that allowed the user to draw and manipulate constrained geometric figures on the computer's display. It included the concepts of a constraint as a declarative relation enforced by the computer, of local propagation constraint solvers, and of multiple cooperating solvers. A subsequent (similar) system, ThingLab [Borning 1981], included a facility for compiling constraint satisfaction plans, allowing constraints to be re-satisfied rapidly for changing inputs. EL [Stallman and Sussman 1977] was an early constraint-based circuit analysis program. The concepts developed here led to a variety of other systems and languages, including Steele's [1980] constraint language, perhaps the first explicit effort at designing a programming language based on constraints.

The main step towards modern constraint programming was achieved when it was noted that logic programming was just a particular kind of constraint programming. Logic programming is based on a declarative computational paradigm in which a program is a logic theory and each computation step solves a system of term equations via the unification algorithm. Its declarative nature made it already close to the idea of constraints, which indeed state *what* has to be satisfied but not *how*. Moreover, the use of a backtracking search to find the answer to a given query is also very similar to the standard backtracking procedures usually used for solving constraint problems. However, what really counted was the observation that term equations are just constraints of a special type and that thus the unification algorithm is just a special kind of constraint-solving algorithm [Lassez et al. 1988]. This has led to the definition of a general framework called constraint logic programming (CLP) [Jaffar and Lassez 1987] that has all the features of logic programming but is parametric with respect to the kind of constraints used within the language. Moreover, it has also brought fundamental changes in areas that were extensively based on equational term rewriting, like computational logic, since researchers in that area realized that they could switch to a more powerful and expressive paradigm by moving from term equalities to constraints [Jouannaud 1994].

Although the CLP scheme immediately gave rise to languages such as CLP(R) [Jaffar et al. 1992] and Prolog III [Colmerauer 1990], it took the practical experience of application-oriented research to link CLP to the propagation algorithms developed earlier in AI. The

language CHIP [Van Hentenryck 1989; Dincbas et al. 1988] realized that extensive use of early ideas on propagation was necessary at both the language and the implementation levels to make CLP languages useful for solving large combinatorial problems (which is usually the task in constraint solving). Thus the language was equipped with the possibility of defining a domain for each variable, and propagation algorithms (mainly achieving arc-consistency) were used to reduce the search for a solution. Facilities for controlling the generation of constraints (forward rules, conditionals, annotations) were provided, although without a clear declarative foundation. This is even more so in recently developed languages such as cc(fd) [Van Hentenryck et al. 1995], where constraint propagation methods can be specified in the language. In this way, the underlying constraint solver can be tailored to the users' needs, achieving the so-called *glass-box approach* (Section 3.7.1).

But constraints in CLP-like languages showed their power not only to model and solve combinatorial problems, but also to prune the search during the computation and thus speed up the execution of a program. This also was a fundamental point, since until then constraints were seen mostly as a knowledge-representation tool rather than as a way to guide computations and prune uninteresting branches.

Another step towards a more general notion of constraint programming came from the area of concurrent logic programming. Concurrent logic programming had already shown that it provided a beautiful, elegant, and powerful notation for concurrent programming, based on the so-called "process" reading of definite clause programs [Shapiro et al. 1989].[3] However, the field was hampered in part by the lack of a clear logical analysis of the synchronization mechanisms introduced into such languages primarily via operational notions. Maher [1987] provided a breakthrough with his analysis that entailment lay at the heart of the synchronization mechanisms. On this basis, Saraswat [1993] developed the simple but general concurrent constraint (CC) programming framework, which views computation as arising from the activities of agents that communicate via a shared set of variables on which they can either impose ("tell") or test ("ask") for the presence of some constraint. The decoupling of this notion of constraint-based computation from definite clause programming made possible the introduction of techniques of process algebra for the further conceptual development of the framework (including the introduction of indeterminacy, etc.). On the one hand, CC programs without asks (and with "angelic" nondeterminism) can be viewed as CLP programs, and CC programs with constraints restricted to term equations are just concurrent logic programs. However, CC provides a general declarative framework for concurrency encompassing and extending data-flow languages, languages based on "residuation," [Aït-Kaci and Podelski 1993] and concurrent functional languages. For the CC paradigm was based on another fundamentally novel observation: that constraints can be used not only to state and solve combinatorial problems, but also to specify process communication and synchronization in a general way. The definition of the CC framework also gave an important impetus to the development of new semantics for such languages that exploit the coexistence of constraints and concurrency in order to be more informative and prove more interesting properties. Examples are the semantics based on traces and closure operators [De Boer and Palamidessi 1991; Saraswat et al. 1991] and those based on truly concurrent models such as Petri nets [Montanari and Rossi 1995a; Gupta et al. 1996].

Languages based directly upon the CC idea are Oz [Smolka 1995], AKL

[3] Another important thread woven into the work on concurrency was the study of "delay primitives" in languages such as Prolog-II and Mu/Nu-Prolog.

[Haridi and Janson 1990], and partly, CIAO [Hermenegildo et al. 1994]. However, the CC framework has to be seen more as a theoretical environment in which new ideas and computational models are defined formally and their theoretical power understood, rather than as a real language. For example, the languages cc(fd) previously discussed are based on the idea of (partial) arc-consistency as closure operators, which arose from the study of the CC semantics.

The two-level architecture of constraint programming is also suited for embedding constraints in more conventional languages, as demonstrated by the 2LP system (which embeds a simplex-based solver into a C-like language) and ILOG Solver, a successful commercial system that embeds many of the ideas and flavor of CLP but as a C++ class library for finite domain constraints.

Among all the constraint languages that have been implemented, it is safe to say that those most widely used today are those based on the CLP framework (but not necessarily using a CLP-like syntax). In fact, these have proven to be successful in many application areas such as resource management and resource allocation. In particular, on benchmark operations research (OR) problems such as job-shop scheduling, these techniques have led to great performance improvements.

3. CONSTRAINT PROGRAMMING TODAY

This section contains an overview of the developments in constraint programming in various subfields. For each subfield, we discuss the main contributions, the applications, and the open issues and directions. The overlap of interests in various subfields will thereby be apparent; we also attempt to emphasize the particular foci of interest that each subfield brings to the table.

3.1 Constraint Programming in Artificial Intelligence

AI research has contributed to considerable progress in constraint-based reasoning. Powerful algorithms perform orders of magnitude better than more naive approaches on difficult combinatorial problems. Considerable attention has been paid to tractability issues: identifying easy classes of problems and generating distributions of problem instances that are hard. Insights into problem structure have supported and connected these research avenues.

Growing interest in applications has motivated increasing interest in representation issues. For example, attention is being paid to overconstrained systems [Jampel et al. 1996], where preferences must be expressed. Modeling is emerging as a major challenge: automating the formulation of real problems in a suitable form for efficient algorithmic processing.

The classic AI constraint paradigm is the constraint satisfaction problem (CSP). It consists of a set of problem variables, each associated with a domain of values, and a set of constraints. Each of the constraints is expressed as a relation, defined on some subset of variables, denoting the consistent value assignments that satisfy the constraint. Often a problem is posed as a constraint network, with variables corresponding to nodes and constraints corresponding to arcs connecting variables occurring in the same constraint.

A solution is an assignment of a value to each variable such that all the constraints are satisfied. Typical tasks are to determine whether a solution exists, to find one or all solutions, to find whether a partial instantiation can be extended to a full solution, and to find an optimal solution relative to a given cost function. Constraints can be described by explicitly presenting the consistent or inconsistent value combinations, or by mathematical expressions or computable procedures that specify these combinations. Often, restrictions

are placed on the paradigm, for example, finite discrete domains or binary constraints (involving two variables), but increasingly, real-world problems are pushing towards extensions.

Algorithms. In general, the tasks posed in the constraint satisfaction problem paradigm are computationally intractable (NP-hard). Over the last two decades, a great deal of theoretical and experimental research has been focused on developing algorithms for solving constraint satisfaction problems and on identifying restricted subclasses that are tractable [Dechter 1992; Mackworth 1992; Tsang 1993].

Techniques for processing constraints can be classified roughly as inference or search, and these approaches interact. Inference methods (such as the path and arc-consistency techniques described in the following) enforce various forms of local consistency that add inferred problem constraints, which can prune away inconsistent values and build up partial solutions. These methods are perhaps the distinguishing contribution of AI to constraint reasoning. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments) and those that explore the space of complete value assignments (to all the variables) stochastically.

Consistency inference. Consistency-enforcing or constraint propagation algorithms[4] transform a given constraint network into an equivalent, yet more explicit network by deducing new constraints to be added onto the network. Intuitively, a consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. For example, an arc-consistency algorithm (Section 2) ensures that any legal value in the domain of a single variable has a legal match in the domain of any other single variable. Path consistency ensures that any consistent solution to a two-variable subnetwork is extensible to any third variable, and, in general, i-consistency algorithms guarantee that any locally consistent instantiation of $i - 1$ variables is extensible to any ith variable. When a network of n variables is n-consistent it is said to be globally consistent, meaning that a solution can be assembled in a backtrack-free manner in any variable ordering. Consistency-enforcing algorithms can be used to preprocess a problem to prune subsequent search, or they can be applied during search. By themselves, these algorithms are, in essence, approximation algorithms that frequently can decide inconsistency.

Systematic search. The most common algorithm for performing systematic search is backtracking. Backtracking incrementally attempts to extend a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. When extension is impossible the algorithm "backs up" to make alternative choices. Improvements of backtracking algorithms have focused on the two phases of the algorithm: moving forward (lookahead schemes) and backtracking (lookback schemes) [Dechter 1990; Kondrak and van Beek 1995].

When moving forward to extend a partial solution, some consistency inference can be carried out to prune the remaining problem space and help decide which variable and value to choose next [Haralick and Elliot 1980]. These methods, which vary in the strength of constraint inference (propagation), try to find a cost-effective balance between pruning and overhead.

Lookback schemes are invoked when the algorithm encounters a dead end. These schemes perform two functions: decide how far to backtrack by analyzing the reasons for the dead end, a

[4] Montanari [1970], Mackworth [1977], Freuder [1978], Mackworth and Freuder [1985], Dechter and Pearl [1987].

process often referred to as backjumping, [Gaschnig 1979]; and record the reasons for the dead end in the form of new constraints so that the same conflicts will not arise again. Terms used to describe this idea are constraint recording and no-good learning [Dechter 1990; Stallman and Sussman 1977].

The order in which variables are instantiated (search order) can have an enormous effect on the cost of finding a solution. An algorithm must choose in which order to process variables, values, and constraints. Often some form of the "fail-first principle" (which chooses the most constrained variable first) is employed in an attempt to prune large portions of the search space by failing high up in the backtrack search tree (e.g., Haralick and Elliot [1980]).

Stochastic search. In the last few years, greedy local search strategies have been reintroduced into the satisfiability and constraint satisfaction literature. These algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill-climbing" metaphor to move towards more and more complete solutions [Minton et al. 1992]. To avoid getting stuck at "local maxima" they are equipped with various heuristics for randomizing the search or for dynamically changing the guiding criterion function by constraint weighting. Although these methods can often be spectacularly successful, their stochastic nature generally voids the guarantee of completeness provided by the systematic methods and thus, in particular, prevents a proof of unsatisfiability or optimality. Analyzing the power of these methods and understanding how to integrate them into a general CP framework are challenging research topics.

Structure-driven algorithms. Problem structure can be characterized and exploited at the micro level (the structure of the constraints) and the macro level (the structure of the constraint network) [Dechter 1992; Freuder 1994].

Many structure-driven techniques emerged from the topological characterization of tractable problems described in the next section. Various graph-based techniques whose complexities are tied to graph parameters were identified. Even when the macro structure of the original problem does not have a characterized tractable structure (e.g. a tree structure), we may still take advantage of tractability results. For example, tree-clustering transforms a problem into a tree-structured metaproblem whose variables are subproblems of the original problem, and the cycle-cutset method extracts a tree-structured subproblem from the original problem [Dechter 1992]. The micro structure can be exploited by, for example, developing specific consistency-enforcing algorithms for specific classes of constraints, or removing values that are redundant because they participate in the same solutions (e.g., see Section 3.7.1).

Structure-driven algorithms such as variable elimination, clustering, and conditioning can be applied across many areas of reasoning such as satisfiability, solution of linear inequalities, belief assessment and belief maximization in Bayes networks, combinatorial optimization, and planning under uncertainty [Dechter and van Beek 1995].

Tractability. The identification of polynomially recognizable restrictions that are sufficient to ensure tractability is important from both the theoretical and the practical points of view and has been extensively studied over the last two decades. Most tractable classes were recognized by realizing that enforcing low-level consistency (in polynomial time) guarantees global consistency or backtrack-free search (e.g., Freuder [1982] and Dechter and Pearl [1987]).

The basic network structure that supports tractability is a generalized tree structure. This has been observed repeatedly from different perspectives in constraint theory [Mackworth and Freuder 1993; Dechter 1992], complex-

ity theory, and database theory. In particular, enforcing arc consistency in a network having a tree structure ensures global consistency along some ordering.

Tractable classes characterized at the micro level have exploited ideas such as tight domains and tight constraints, row-convex networks, implicational constraints, and max-ordered constraints. These classes justify the intuition that problems having large domains and higher arity constraints are generally harder. The investigation of classes of constraints that ensure tractability in whichever way they are combined has related tractability to algebraic closure properties of the constraints [Jeavons et al. 1996].

Finally, special classes of constraints associated with temporal reasoning have received much attention in the last decade. Tractable classes include subsets of Allen's [1983] (qualitative) interval algebra, as well as quantitative binary linear inequalities over the reals, of the form $X - Y \leq a$ [Dechter et al. 1991]. The focus in the AI community (in contrast to OR) is on handling new types of queries and on combining such constraints with qualitative constraints.

Generating hard instances. Another theme that has received great interest recently is locating the "really hard" problems [Cheeseman et al. 1991]. It turns out that when problems are generated randomly, most of them are very easy. Consequently, special care is needed in selecting the random generator if nontrivial problems are to be produced. It has recently been demonstrated that most random generators have a phase transition from easy to hard, where hard distributions are located wherever only few solutions exist.

Applications. The previously described algorithms serve as general-purpose inference engines for accomplishing tasks modeled as constraint satisfaction problems. Many tasks are naturally so modeled:

—reasoning tasks including default reasoning, abduction, causal reasoning, diagnostic reasoning, temporal reasoning, and spatial reasoning;
—cognitive tasks including machine vision, natural language processing, and planning; and
—task domains including scheduling, resource allocation, configuration, and design.

3.2 Constraint Programming in Databases

The importance of constraints in the context of databases has been recognized for a long time. For instance, in SQL/92, the current standard for SQL, simple arithmetic constraints can be used in defining queries and assertions (which are a form of "integrity constraint," i.e., conditions that must be satisfied by a database instance). The use of arithmetic constraints for semantic query optimization and optimization of SQL queries involving constraints has been extensively investigated.

The area of constraint databases (CDBs), in which constraints are integrated as a basic data type, has emerged recently, prompted by the seminal work of Kanellakis et al. [1995]. Constraint databases naturally extend relational, deductive, or object-oriented databases by making feasible the use of constraints to represent possibly infinite but finitely representable complex data. This has turned out to be natural for many application domains, since constraints possess great modeling power. Constraints serve as a highly uniform data type for conceptual representation of heterogeneous data, including spatial and temporal behavior, complex design requirements, and partial and incomplete information.

For example, arithmetic constraints over real variables within a subset of first order logic can describe a wide variety of data, including 2D or 3D geographic maps; geometric modeling objects for CAD/CAM; fields of vision of sensors; 4D (3 + 1 for time) trajectories of objects moving in 3D space, based on

the movements equations; translation of different systems of coordinates; operations research type models such as manufacturing patterns describing interconnections among quantities of manufactured products and resource materials.

The notion of constraint data relies on a simple and fundamental duality: a constraint (formula) ϕ in free variables $x_1, \ldots, x_n$ is interpreted as a set of tuples $(a_1, \ldots, a_n)$ over the scheme $x_1, \ldots, x_n$ that satisfy ϕ. Conversely, a finitely representable relation over the scheme $(x_1, \ldots, x_n)$ can be viewed as a constraint. For example, a constraint $(-4 \leq w \leq 4) \wedge (-1 \leq z \leq 2)$ with variables ranging over reals is interpreted as the set $\{(w, z) | (-4 \leq w \leq 4) \wedge (-1 \leq z \leq 2)\}$ and describes, say, the rectangle shape of a desk given in its local system of coordinates (w, z). Users can intuitively think of a constraint as an object in space (i.e., space of points) or as a symbolic expression, interchangeably, depending on the application and context of its use. We use a generic name *constraint object* in the context of databases.

A constraint object is usually represented by a collection of atomic constraints, such as real polynomial, linear, or dense order, and their logical combinations. Constraint objects are manipulated by means of a constraint calculus/algebra involving logical operations such as quantification, conjunction, disjunction, negation, and implication. If we only use linear constraint over reals within first-order logic, we can express any linear transformation such as rotation, translation, and stretch; check convexity, discreteness, and boundedness; compute convex hull, augment objects, change coordinate systems, and so on.

Thus constraint objects can be manipulated by a very expressive and general-purpose language, as opposed to using separate custom operators for each specific type of transformations (as done typically in extensible or spatial database systems). For many useful constraint domains, query languages manipulating constraint objects are highly optimizable, in terms of indexing and filtering (e.g., Brodsky et al. [1995], Kanellakis et al. [1993], and Srivastava [1992]), and constraint algebra algorithms and global optimization (e.g., Brodsky et al. [1993] and Goldin and Kanellakis [1996]). Examples of implemented constraint databases are Gross-Brunschwiler [1996] and Byon and Revesz [1995].

Although the use of constraints as data is a central feature in constraint databases, an important contribution of the field is the technology that has been developed with regard to the use of constraints for optimizing evaluation of database queries. The idea of storing constraints as tuples in the database (so-called magic template tuples) and using this information to prune the search during database query evaluation was first proposed in Ramakrishnan [1988]. The idea was refined in Balbin et al. [1989] and Mumick et al. [1990] to allow constraint propagation without actually storing constraints in the database, for the case of nonrecursive SQL queries, by careful repositioning of the constraints in a query. This prompted a series of work on the repositioning of constraints in (recursive and nonrecursive) database queries for the purpose of optimization, such as pushing constraint selections in Srivastava and Ramakrishnan [1992] and Levy et al. [1994] or finding redundant parts of evaluation trees using query constraints in Levy and Sagiv [1992].

The promise of the emerging constraint database work is that it will provide a uniform framework for the declarative and efficient querying of symbolically represented data. Developing custom tools for specific applications usually requires considerable programming effort, and yields products that are not easy to change and may not perform overall optimizations that interleave database, mathematical programming, and computational geometry manipulation techniques. Existing DBMS do not

handle constraints as stored data, and CLP implementation techniques need to be developed to deal with large amounts of persistent data.

The work of Hansen et al. [1989] considered polynomial equality constraints as rules, taking advantage of their adirectionality. Kanellakis et al. [1995] proposed a framework for integrating abstract constraints into database query languages by providing a number of design principles, and studied, mostly in terms of expressiveness and complexity, a number of specific instances. A restricted form of linear constraints, called linear repeating points, was used to model infinite sequences of time points (e.g., Kabanza et al. [1990]). More recent works on deductive databases (e.g., Mumick et al. [1990]) considered manipulation and repositioning of constraints for optimizing recursion. Algorithms for constraint algebra operators such as constraint joins, and generic global optimization were studied in Brodsky et al. [1993]. The work of Kanellakis et al. [1993] proposed an efficient data structure for secondary storage suitable for indexing constraints that achieves not only the optimal space and time complexity as priority search trees, but also full clustering. The work of Brodsky et al. [1995] proposed an approach to achieve the optimal quality of constraint and spatial filtering. A number of works consider special constraint domains: integer-order constraints [Revesz 1993]; set constraints [Revesz 1995]; dense-order constraints [Grumbach and Su 1995]. Linear constraints over reals have drawn special attention.[5] The use of constraints in spatial database queries was addressed in Paredaens et al. [1994]. The work of Srivastava et al. [1994] used constraints to describe incomplete information. Constraint aggregation was studied in Kuper [1993].

[5] See Afrati et al. [1994], Brodsky et al. [1993], Grumbach et al. [1995], and Vandeurzen et al. [1995].

3.3 Constraint Programming in User Interfaces

Constraint programming has a long history of use in graphics and user interfaces, beginning with the Sketchpad system [Sutherland 1963]. Common applications of constraints in user interface construction include layout and other kinds of geometric constraints, maintaining consistency between application data and a view on those data, keeping multiple views consistent, animation, and providing semantic feedback.

Supporting interactive user interfaces places a number of demands on constraint satisfaction algorithms that may not arise in other application areas. The algorithms must be fast—in a typical interactive application, the constraints must be re-satisfied each time the screen is refreshed while moving some part. State and state change are also fundamental in these applications, as geometric objects are moved on the screen, windows are reshaped, and so forth. We typically also require the algorithm to provide specific values for variables rather than symbolic solutions, since the graphical elements must be shown in some location.

Two classes of algorithms in common use for user interface (UI) applications are one-way constraint algorithms and multi-way local propagation algorithms. In a one-way algorithm, each constraint has a distinguished output variable that the solver can set to satisfy that constraint; the other variables are only referenced by the constraint. For example, if c is the output variable in the constraint $a + b = c$, the solver can update c to satisfy the constraint if a or b changes. A multi-way local propagation constraint includes a collection of methods for satisfying that constraint. For example, the $a + b = c$ constraint would have three methods: $a \leftarrow c - b$, $b \leftarrow c - a$, and $c \leftarrow a + b$, which can be used to find a value for a, b, or c that satisfies the constraint. Examples of user interface toolkits using one-way

constraints include Amulet [Myers 1996] and its predecessor Garnet; examples of multi-way local propagation algorithms include DeltaBlue [Sannella et al. 1993], SkyBlue [Sannella 1995], and QuickPlan [Vander Zanden 1996]. (These multi-way algorithms all also support constraint hierarchies [Borning et al. 1992; Jampel et al. 1996], which allow both required and preferential constraints. Constraint hierarchies are useful in such common UI tasks as specifying which parts of a figure we prefer to leave fixed while moving some other part.)

Some algorithms allow for cycles of constraints (e.g., simultaneous equations) and inequalities, neither of which is supported by traditional local propagation algorithms. Examples include QOCA [Helm et al. 1992], which solves simultaneous linear equations and inequality constraints while optimizing a quadratic expression, Bramble [Gleicher 1995] and Juno-2 [Heydon and Nelson 1994] which use numerical solvers, Indigo [Borning et al. 1996], an interval propagation algorithm for inequality constraints, and DETAIL [Hosobe et al. 1996] and Ultraviolet [Borning and Freeman-Benson 1995], both of which are hybrid algorithms supporting both local propagation and cycle solvers.

3.4 Constraint Programming in Operations Research

Operations research is a vast field represented by departments in major universities and industrial settings around the world. The field of OR has significant overlap with AI, branch-and-bound search being a classic example, tabu search and simulated annealing being somewhat more recent examples. CP is a much smaller but emergent discipline that is situated at the confluence of computer science (CS), AI, and OR.

A principal area of intersection of CP with OR is the field of NP-hard combinatorial problems. What most distinguishes OR approaches to these problems is the consistent use of continuous methods based on linear programming. With this (very successful) method, known as mixed integer programming, an application is modeled as a system of linear constraints on real and integer variables. To assist in the solution process, the model is enhanced with constraints known as cuts that tighten the linear relaxation of the model [Nemhauser and Wolsey 1988]. This is often critical in limiting the amount of search that is required to find a solution. Generating the right cuts for a given application is a demanding craft that exploits the mathematical structure of the problem. The problem-solving process also requires a linear programming and/or mixed integer programming library.

On the other hand, in CP the emphasis has been less on the mathematical structure of the particular application and more on higher-level modeling and solution methods and tools, and on the integration of ideas from many different constraint systems. This has led to languages based on finite domain solvers and linear programming solvers, phase transition analysis of problem difficulty, algorithmic advances, and the like. It has also led to the expansion of the OR arsenal with constraint solving libraries other than linear and mixed integer programming libraries.

A classic shared interest of CP and OR is declarative programming. In fact, in terms of languages, the interaction between CP and OR goes back at least to Lauriere [1978]. The formulation of a mixed integer program is quintessentially declarative. Moreover, the algebraic modeling languages of OR (such as GAMS, AMPL, AIMMS) provide an example of a very pure form of declarative programming system. This programming paradigm is in evolution and may well be converging with developments in the CP world, as declarative programming systems become more open to integrating other paradigms. A case in point is the 2LP language (linear programming and logic programming), which is designed to encapsulate a part of the practice of OR, namely, mixed

integer programming and extensions [McAloon and Tretkoff 1997].

Work in OR on discrete optimization has also contributed to developments in CP. Indeed, some of the recent success in CP on scheduling problems can be traced back to Carlier and Pinson [1989] on the job shop problem. Conversely, the CP work has led to new algorithms for these and related applications and to the creation of software tools to facilitate exploitation of these techniques.

As computational sciences such as OR develop more complex methods to deal with more challenging applications, a role to be played by CP is to furnish software tools and concepts to organize the construction of these systems. To this effort CP brings some new ideas and facility with program and language design that will help bring OR technology to a much larger audience. CP systems are being used commercially in many application areas, where they bring competitive advantage to users over traditional approaches in terms that often include application development ease, quality of solution, and speed at obtaining this solution. Such applications are typically in the areas of scheduling (disjunctive constraints, task intervals), resource control (cumulative, bottlenecks), transportation (cycle constraints, labeling heuristics), personnel rostering (sequence constraints), workforce scheduling (constraint cooperation), circuit verification (Boolean constraints), electromechanical systems (constraints and finite-state machines, safety and fairness properties). Some of these applications are described in the proceedings of the conferences on "Practical Applications of Constraint Technology—PACT."

3.5 Constraint Programming in Concurrency

As noted in Section 2, the use of constraints as a convenient mechanism for process communication and synchronization in a concurrent environment led to the development of the CC paradigm, where processes interact by posting and asking constraints over a shared set of variables. This very general and elegant computational paradigm has received a lot of theoretical and implementation attention since its conception in 1989. In fact, the literature shows many semantics efforts that try to adapt either the interleaving models of process description algebras to CC [Saraswat 1993; De Boer and Palamidessi 1991] or the truly concurrent ones of Petri nets and event structures [Montanari and Rossi 1995a; Rossi and Montanari 1994]. Other theoretical efforts focus on the possibility of analyzing CC-like programs at compile time, thus deriving properties to be used at run time. This holds, for example, for the works on abstract interpretation [Zaffanella 1995; Codognet et al. 1990], which execute CC programs on an abstract constraint domain with the hope of deriving some useful knowledge for program simplification, for those on suspension analysis [Codish et al. 1994], whose aim is to understand the conditions under which CC programs deadlock, and for those on relating CC and CLP languages [Bueno et al. 1994], which try to parallelize CLP programs using CC-based techniques or to sequentialize CC programs via an analysis of their inherent concurrency.

Languages such as AKL [Haridi and Janson 1990], Oz [Smolka 1995], and CIAO [Hermenegildo et al. 1994] are essentially based on the CC ideas, although they add many features mainly because of application needs and efficiency. For example, AKL employs a model of computation based on the so-called Andorra principles, which basically leads to executing all deterministic steps first. Oz is a lexically scoped language with first-class procedures, state, and encapsulated search. CIAO is an extensible constraint language supporting CC-style concurrency and synchronization primitives in combination with standard CLP programming, as well as several control rules.

3.6 Constraint Programming in Robotics and Control Theory

A major challenge facing the constraint research community is to develop useful theoretical and practical tools for the constraint-based design of embedded intelligent systems. An archetypal example of an application in this class is the design of controllers for sensory-based robots.

Many of the tools developed to date in the CSP and CP paradigms are not adequate for the task, despite the superficial attraction of the constraint-based approach. The fundamental difficulty is that, for the most part, the CSP and CP paradigms presume an offline model of computation. But intelligent systems embedded as controllers in real physical systems must be designed in an online model. Moreover, the online model must be based on various time structures: continuous, discrete, and event-based. The requisite online computations, or transductions, are to be performed over various type structures including continuous and discrete domains. These hybrid systems require new models of computation, constraint satisfaction, and constraint programming. For example, Zhang and Mackworth [1994] defined constraint satisfaction as a dynamic system process that approaches asymptotically the solution set of the given, possibly time-varying, constraints. Under this view, constraint programming is the creation of a dynamic system with the required property. Many robots can be designed as online constraint-satisfying devices [Pai 1991; Zhang and Mackworth 1995a]. A robot in this restricted scheme can be verified more easily. Moreover, given a constraint-based specification and a model of the plant and the environment, automatic synthesis of a correct constraint-satisfying controller becomes feasible, as shown for a simple ball-chasing robot in Zhang and Mackworth [1995b].

Another approach has been developed recently in Saraswat et al. [1995] and Gupta et al. [1997] for modeling timed reactive systems. Reactive systems are those that react continuously with their environment at a rate controlled by the environment. Execution in a reactive system proceeds in bursts of activity. In each phase, the environment stimulates the system with an input, obtains a response in bounded time, and may then be inactive (with respect to the system) for an arbitrary period of time before initiating the next burst. Examples of reactive systems are controllers and signal-processing systems. The timed concurrent constraint programming (TCC) framework extends CCP by adopting the synchrony hypothesis of languages such as ESTEREL: program control constructs are determinate primitives that respond instantaneously to input signals. At any instant the presence and the absence of signals can be detected. This is accomplished by augmenting CCP with two constructs: first, **hence** A requires that the program A be executed at every time instant from the next time onwards. Next, a construct **if** c **else** A is added requiring A to be triggered if the constraint c is *not* enforced now or through quiescence. This "nonmonotonic" control construct is motivated by Reiter's Default Logic and provides a very powerful and simple way to formalize the elaborate synchrony constructs of languages such as ESTEREL and LUSTRE. The same ideas have been used to extend CCP to continuous time, by introducing the notion of autonomous activity [constraints of the form $(d/dt)(X) = k$ which allow a variable to vary continuously with real time, independent of stimulus from the environment] and changing the underlying model of time from the integers to the reals. The resulting framework is quite simple mathematically and a very powerful basis for compositional modeling [Gupta et al. 1995].

The modeling and design of robotics systems and embedded control systems presents a serious challenge and opportunity for constraint-based theories of computation.

3.7 Constraint Systems and Programming Tools

Despite the youth of the field, a good number of tools for developing constraint programs have become available and a substantial set of techniques has been developed to support the efficient implementation of such programs.

3.7.1 *Constraint Domains and Solving Techniques.* A relatively small number of constraint systems (with their associated solution techniques) have been used as a basis for several concrete implementations. The four most important domains, other than rational trees, are Boolean constraints, finite domains, real intervals, and linear constraints; other examples include lists and finite sets.

Boolean constraints are either treated by a specialized constraint solver, as in CHIP or Prolog III, or seen as a specialized case of finite domain constraints. In the latter, a Boolean is considered as an integer between 0 (false) and 1 (true), as in CLP(BNR), Prolog IV, clp(FD), or ILOG Solver. There has also been work on constraint solving over more general Boolean algebras.

Finite domain constraints are constraints on integer-valued variables. These constraints are useful in many application areas. They are usually solved by combining propagation techniques (such as arc-consistency) with backtracking search. Each variable is associated with a finite set of possible values (possible starting time for an activity, possible component for an assembly, possible coworkers for a team member, and so on). This set is called the *domain* of a variable. Inconsistent values are removed from the domain of variables during propagation, and then search tries to assign a value to each variable.

The propagation phase is built on a very simple idea: remove inconsistent values from the domain of the variables. For instance, assume that x, y, and z are three variables with integer values in the closed interval [1, 10], with the constraint $y < z$. We can see that the value of y is at least 1. Since the constraint states that z must be greater than y, $z = 1$ is no longer possible. For that reason, 1 is removed from the domain of z, which becomes [2, 10]. Similarly, the domain of y becomes [1, 9]. The domain of x remains unchanged since no constraints involve x at this point. Let us assume now that we add another constraint, say, $x = y + z$. Now the minimal possible value for y is 1 and the minimal possible value for z is 2, so x has to be at least 3. The domain of x is then reduced to [3, 10]. Furthermore, as the maximal possible value for x is 10 and the minimal value of y is 1, z, which is equal to $x - y$, must be at most 8. Similarly, y, which is equal to $x - z$, must be smaller than 8.

Real interval constraints are the analogue of finite domains when reals are considered instead of integers. As it is impossible to represent explicitly the set of reals that a variable can take, the domain of a real variable is an interval whose bounds are floating-point numbers. The techniques for removing inconsistent values are either similar to finite-domain techniques (e.g., in CLP(BNR), Prolog IV, and ILOG Solver) or are based on mathematical techniques such as automatic differentiation and Taylor series (as in Newton and Helios). Real interval constraints usually include trigonometric and other nonlinear constraints.

Linear constraints are constraints posted on real variables that have a special form: they only involve weighted sums of variables (no product or more complex expressions). For such constraints, very efficient constraint solvers have been implemented using the simplex algorithm as a starting point. Some linear constraint solvers use infinite precision (rational numbers); others use floating point computations. The former are more accurate, whereas the

latter are more efficient. Interior point methods have been introduced in linear programming libraries but have not had an impact on constraint programming more generally.

"Global" constraints refers to an important line of work that aims to define good propagation algorithms for more complex constraints. The removal of inconsistent values can be tricky for more complex constraints. In this context, scheduling constraints, all-different (a set of variables takes on values that are all different), cardinality constraints (the number of constraints within a set that must be satisfied is required to be within given lower and upper bounds), and spatial constraints have been studied in detail in the literature. The use of global constraints is often the key for a successful application. For instance, in scheduling, some constraints can be used to state that a given resource has a finite capacity, which limits the number of tasks that can require the resource at any time. The propagation of such a constraint requires a sophisticated algorithm adapted from operations research.

User-defined constraints are the result of one of the lessons learned so far from the application of CP tools in practice: that domain-specific constraints are often needed. In other words, the user of these systems often needs to extend the constraint system with some constraints that are specific to the application at hand. Several proposals have been made for enabling the user to add domain-specific constraints to the system and tailor the underlying constraint solver (or program a new, specific solver) to these specific constraints. This is called the glass-box approach, in contrast with the original CLP idea of the constraint solver as a black box.

Building on progress in the area of concurrent constraint programming, some languages provide constructs for defining the propagation of a constraint within the language (examples are cc(fd) [Van Hentenryck et al. 1995] and clp(fd) [Codognet and Diaz 1996]). Others propose viewing a constraint as a Boolean expression. The Boolean variable is true if the constraint is necessarily true (entailed by the other constraints), and false if the negation of the constraint is entailed by the other constraints. This makes possible the combination of constraints with logical operators (or, not, and), as well as some more complex constructs such as cardinality [used, for example, in CLP(BNR), Prolog IV, and ILOG Solver]. A related approach is to define constraints using a rewrite system, as in the constraint handling rules solution [Frühwirth 1995]. The promise of such a special-purpose language for defining constraint systems is that properties of a constraint solver such as termination and confluence can be tackled independently of a particular constraint system.

Yet another approach is to provide hooks in the parameter-passing mechanism of the language (e.g., within unification, for CLP systems) through attributed variables or meta-terms [Neumerkel 1990; Holzbaur 1992]. This approach is used extensively in the implementation of constraint solvers in systems such as ECL^{i}PS^e [European Computer Research Center 1993], SICStus, and CIAO [Hermenegildo et al. 1994]. A final approach is motivated by the need to add support for global constraints. In that case the definition of the constraint is done in an imperative language and linked with the CP system using an object-oriented protocol (used in CHARME, ILOG Solver, Oz, CHIP). This approach, called the "no-box" approach of Puget and Leconte, potentially yields the most efficient implementations, although implying a higher programming load.

3.7.2 *Constraint Programming Tools.* The constraint systems previously discussed have been integrated into different programming languages, ranging from subsets of first-order logic to imperative languages such as C++, or even specialized languages. One of the

most popular approaches is to use Horn clauses as a basis (as in Prolog), and then extend this with one or more constraint systems, in addition to unification over Herbrand terms. This constraint logic programming approach has led to many important tools, including CLP(R) (linear constraints), Prolog III (Booleans, linear constraints, and lists), CHIP (Booleans, linear constraints, finite domains), clp(fd) (finite domains, Booleans), ECL^{i}PS^e (finite domains, linear constraints), CAL, GDCC, and so on.

Another popular approach is to embed CLP techniques in a different host language, leading to another set of tools including the following (for each we indicate both the underlying programming language and the constraint domains supported).

—CHARME: specialized language with C-like syntax and finite domains
—2LP: C-based language with linear constraints
—ILOG Solver: C++ library with Booleans, finite domains, real intervals, and linear constraints
—HELIOS: specialized modeling language with real intervals

Finally, a number of systems offer a concurrent language as the underlying programming component (concurrent constraint languages):

—AKL: nondeterministic concurrent constraint language with finite domains. Supports both CC and CLP programming styles. Supports parallel execution
—Oz: specialized concurrent multiparadigm language (object-oriented, higher-order functional, search) with finite domains. Support for distributed execution
—CIAO: extensible concurrent constraint logic language with linear constraints. Supports CC-style programming within CLP, parallel and distributed execution, several control rules, functions

In addition to these and other relatively general-purpose tools, tools specifically tailored to certain problem classes have been proposed. For example, ILOG Schedule is a tool built using ILOG Solver functionality and is specifically tailored to solving scheduling problems while offering a simple, graphical user interface.

3.7.3 *Debugging and Visualization Tools.* The development of industrial applications using early CP systems has pointed out the need for studying CP-specific debugging techniques beyond those traditionally used for imperative or logic programming systems on which they are based. Applying traditional methods, which include standard program tracing as well as declarative debugging approaches [Shapiro 1982], often suffices for developing correct programs, but understanding the performance of CP programs often requires additional tools. Proposed solutions include both compile-time and run-time techniques. A compile-time technique that has received some attention is the static generation and/or checking of assertions. Such assertions can be seen as a generalization of type systems in which relatively general preconditions and postconditions expressed as constraints can be declared for procedures. Assertions can be provided by the user and/or checked by the compiler (when possible) via global analysis. Alternatively, they can be generated by the compiler and the user can inspect them for errors. In both cases global analysis techniques and systems similar to those used by the compiler for optimization purposes, discussed later in this section, can be used for these purposes (e.g., García de la Banda et al. [1997]), as well as, perhaps, other proof techniques previously used in logic programming (e.g., based on induction assertion). A run-time technique currently receiving much attention is the use of visualization, both of the search space and of the

constraint store at different points of execution [Meier 1996].

3.8 Constraint Programming Language Implementation Techniques

Compilers and abstract machines. The programming component that CP offers as an essential addition to the constraint-solving capabilities is implemented in an efficient way in most current CP programming systems via compilation. In the case of library systems built on top of conventional programming languages (such as, for example, ILOG built on top of C++), the compilation of the control component is provided by the host language compiler. In the case of systems that offer a programming language, the programming component is, as mentioned before, very often offered by a logic-programming-based language. Compilation is then generally based, at least conceptually, on a translation to an abstract machine instruction set.[6] The target abstract machines used are most often generalizations of the Warren Abstract Machine, which has proven extremely successful in the context of logic programming. The WAM approach essentially provides a view of the compilation of these languages as a generalization of the standard techniques used in conventional languages, allowing most of the conventional optimizations.

Global analysis. As a result of the compilation-based approach, the performance of current systems is quite acceptable when running code in which general-purpose constraint solving is performed. On the other hand, this approach alone cannot always provide performance in the control component that is competitive with other languages. In particular, their performance often does not reach that of traditional logic programming systems in symbolic applications and is generally far from that of traditional imperative programming languages in (nonconstraint-related) numerical applications. The most generally accepted solution to this has been to develop advanced compilation technology capable of detecting the cases where limited or no constraint solving is involved and compiling those cases in the most efficient way. Some significant progress has already been made in practical global analysis and optimization of constraint logic programming systems. Results on the possible speedups obtainable with global analysis information have been studied (e.g., Marriott and Stuckey [1992] and García de la Banda et al. [1996]), practical frameworks for global analysis have been developed (e.g., García de la Banda et al. [1997]), and some CP systems have been reported that perform global analysis-based optimization [Kelly et al. 1996; García de la Banda et al. 1996]. Such global analysis has also been applied to concurrent CP systems, where one of the most important objectives is to reduce suspension and resumption of goals and synchronization overhead.[7] Finally, recent progress in incremental global analysis (e.g., Hermenegildo et al. [1995]) has the potential to solve most remaining problems related to supporting large programs and the use of global analysis in the interactive program development environment that is common in constraint programming systems. However, the application of extensive optimization in commercial or widely used public domain systems still remains a goal to be achieved. Also, much research remains to be done in finding accurate abstraction techniques for standard constraint systems.

Parallelization. A program optimization that has shown significant speedups in the context of logic programs is automatic parallelization [Chassin and Codognet 1994]. Exploitation of paral-

[6] See, for example, Jaffar et al. [1992], Van Hentenryck et al. [1995], Dincbas et al. [1988], and Codognet and Diaz [1996].

[7] See Codish et al. [1993], Falaschi et al. [1993], Marriott et al. [1994], Bueno et al. [1994], and García de la Banda et al. [1995].

lelism in the search (or-parallelism) is comparatively easy and has been shown to provide speedups in several industrial applications containing extensive search [Van Hentenryck 1989b; European Computer Research Center 1993; Li et al. 1993]. On the other hand, comparatively little work has been devoted so far to exploiting parallelism within a given path of the search (and-parallelism) and in the solver itself. Although traditional concepts of independence used in imperative programming (e.g., the Bernstein conditions) or even those of logic programming, do not apply in the context of CP [García de la Banda et al. 1993], notions of independence appropriate for (concurrent) CP have been recently proposed [García de la Banda et al. 1993; Bueno et al. 1994]. Based on this, parallelizing compilers as well as and-parallel abstract machines for CP languages have recently become available, and initial performance results are encouraging [García de la Banda et al. 1996].

4. PROMISING DIRECTIONS

Constraint programming has by now shown that constraints can be used not only to represent knowledge but also as a way to guide search, prune useless branches, filter queries, and describe process communication and synchronization. With this in mind, we identify several directions for research that are promising for systems, programming environments, models, and application packages.

More realistic constraint systems and languages. We need to develop more automatic and systematic ways to acquire and model domain-specific and problem-specific knowledge, developing a richer paradigm to cope with the properties and uncertainties of real-world information. Of course, representation and reasoning are always two sides of the same coin. As we consider new classes of constraints, we must also consider new methods to compute with them; automating the modeling process will itself require capturing some very sophisticated reasoning skills. Moreover, better theoretical and empirical understanding is needed of the relationship between real-world problem parameters and search methods. An important issue is that of over-constrained constraint problems [Jampel et al. 1996], since most real-life problems are indeed over-constrained. Thus either the constraint domain or the language itself should be flexible enough to be able to deal with such situations and solve them in some satisfactory way. For example, the constraints and constraint-solving algorithms could take into account the presence of preferences of some sort [Bistarelli et al. 1995; Borning et al. 1992; Govindarajan et al. 1996], and/or the language could allow for user-guided constraint retraction [Codognet and Rossi 1995; Best et al. 1995] and intelligible explanations for failure. This of course would bring the constraint satisfaction and programming tasks closer to the issues present in optimization problems, since in the presence of preferences one has to decide the best way to choose and/or retract constraints. Thus special attention has to be paid to the interrelation between AI and OR techniques for such tasks. In particular, we must take advantage of the coexistence, in the constraint satisfaction world, of different methods (e.g., systematic and stochastic search) and different disciplines (e.g., artificial intelligence and operations research).

Efficient modeling. Constraint satisfaction knowledge can be represented very declaratively, without regard to how it is to be used. However, modeling a specific problem is not a trivial task, especially since how it is modeled can dramatically affect how well our algorithms perform. We need to automate the process of moving from problem descriptions natural to the problem domain to problem descriptions designed for efficient solution. A variety of prob-

lem-solving techniques are now available to us, but synthesizing appropriate algorithms for specific tasks should be automated [Minton 1996]. In addition, robust constraint computation must cope with change in the world and in models, and with noise (e.g., in data) and uncertainty (e.g., in parameter values).

Towards constraint-based distributed systems. Another challenge for constraint programming systems is related to the role of such systems in network-wide programming. This type of programming is likely to be of growing importance, given that the recent wider diffusion of the Internet and the popularity of the World Wide Web (WWW) protocols are effectively providing a new platform that is standard and ubiquitous and allows a new class of highly sophisticated distributed applications. Features of constraints such as the ability to describe intra- and inter-process communication and synchronization are more and more important in practical applications that consist of distributed environments where both local problem solving and global synchronization and coordination are needed. This is added to the fact that many CP systems already offer many other characteristics that make them well suited in this context, among them dynamic memory management, well-behaved structure and pointer manipulation, robustness, dynamic compilation to architecture-independent bytecode, dynamic databases, search facilities, grammars, code motion, and sophisticated metaprogramming. A number of distributed concurrent constraint systems are currently being worked on, application development libraries are being offered, and network and WWW applications are being reported [Tarau et al. 1996]. It appears that CP is a promising foundation for most aspects of the next generation of distributed systems, where all the advantages of constraints may coexist, and thus lead to simple, elegant, and practically usable environments.

Another interesting related application domain is 3D graphics and virtual reality. Many interactions among objects (e.g., attachments, minimal distances, noncollision, etc.) or general integrity rules (such as energy conservation laws) can be considered as constraints and implemented efficiently as such. This generalizes 2D geometrical constraints in an obvious way. Basically, constraints can be used to enforce hidden relations between objects and thus make sure that the simulated virtual world does not depart too much from our real one.

Towards faster, more efficient systems. The performance and computing resource economy of current CP systems have proved adequate in significant industrial applications, competing very favorably with other techniques and approaches, however, there still remain many avenues for improvement that would make the technology even more competitive. It is expected that improving execution speed and further reducing resource consumption will improve the acceptance of the approach for general-purpose programming as well as encouraging the inclusion of constraint programming techniques, constructs, and libraries in conventional languages. Interesting techniques to be further explored include advanced compilation based on global analysis and (automatic) program and solver parallelization. In fact, parallelization is becoming more and more interesting since multiprocessing hardware is becoming in many cases the default installation platform (e.g., for departmental servers where multiprocessors using fast, inexpensive, off-the-shelf processors are often replacing mainframes at a fraction of their cost). Also, multiprocessor workstations are not unusual any more. It appears likely that this trend towards increased use of parallelism will continue as multiprocessor architectures are better understood, interconnection network performance increases with new technologies (especially if the promise of optical in-

terconnect is finally delivered), and feature size diminishes, allowing placement of several processors on the same chip.

Constraint databases. Many challenges in constraint databases are yet to be addressed. Specific directions of work include: constraint modeling, canonical forms and algebras; data models and query languages; indexing and approximation-based filtering; constraint algebra algorithms and global optimization; systems and case studies. In addition, robust, widely available implementations of these ideas need to be developed.

User interfaces. In user-interface applications, there is a constant need for new constraint-satisfaction algorithms that can handle a wider range of constraints that arise in such applications, and algorithms and data structures with improved space and time efficiency.

The development of better (performance) debugging techniques and more useful visualization paradigms for several constraint domains and constraint-solving algorithms also offers an interesting research direction. Currently, at least one European project is working on the development of both assertion-based and visualization-based debugging techniques for CLP systems.

Among the issues that should be addressed are ways of describing the desired constraints at a higher level of abstraction (closer to the domain of interest), studying the models users have of constraint systems, and evolving those systems as needed to allow clearer and more easily understood user models.

REFERENCES

AFRATI, F., COSMADAKIS, S., GRUMBACH, S., AND KUPER, G. 1994. Linear versus polynomial constraints in database query languages. In *Proceedings of the International Workshop on Principles and Practice of Constraint Programming,* (Orcas Island, WA), vol. 874 *LNCS*, Springer Verlag, New York, 181–192.

AÏT-KACI, H. AND PODELSKI, A. 1993. Towards a meaning of LIFE. *J. Logic Program. 16,* 3&4, 195–234.

ALLEN, J. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26,* 832–843.

BALBIN, I., KEMP, D. B., MEENAKSHI, K., AND RAMAMOHANARAO, K. 1989. Propagating constraints in recursive deductive databases. In *Proceedings of the North American Conference on Logic Programming.*

BEST, E., DE BOER, F. S., AND PALAMIDESSI, C. 1995. Concurrent constraint programming with information removal. In *First Conference on Concurrent Constraint Programming* (Venice).

BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 1995. Constraint solving over semirings. In *Proceedings of the International Joint Conference on Artificial Intelligence.* Morgan-Kaufman, San Mateo, CA.

DE BOER, F. S. AND PALAMIDESSI, C. 1991. A fully abstract model for concurrent constraint programming. In *Proceedings of the CAAP.* Springer-Verlag.

BORNING, A. 1981. The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Trans. Program. Lang. Syst. 3,* 4, 353–387.

BORNING, A., ANDERSON, R., AND FREEMAN-BENSON, B. 1996. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Seattle, Nov.).

BORNING, A. AND FREEMAN-BENSON, B. 1995. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* (Cassis, France), Vol. 976, U. Montanari and F. Rossi, Eds., Springer-Verlag, 624–628.

BORNING, A., FREEMAN-BENSON, B., AND WILSON, M. 1992. Constraint hierarchies. *Lisp Symbol. Comput. 5,* 3 (Sept.), 223–270.

BRODSKY, A., JAFFAR, J., AND MAHER, M. J. 1993. Toward practical constraint databases. In *Proceedings of the International Conference on Very Large Data Bases* (Dublin).

BRODSKY, A., LASSEZ, C., LASSEZ, J.-L., AND MAHER, M. J. 1995. Separability of polyhedra for

optimal filtering of spatial and constraint data. In *Proceedings of the ACM Symposium on Principles of Database Systems.* ACM Press, New York.

BUENO, F., JOSE GARCÍA DE LA BANDA, M., HERMENEGILDO, M., MONTANARI, U., AND ROSSI, F. 1994. From eventual to atomic and locally atomic CC programs: A concurrent semantics. In *Proceedings of the International Conference on Algebraic and Logic Programming.*

BUENO, F., JOSE GARCÍA DE LA BANDA, M., HERMENEGILDO, M., ROSSI, F., AND MONTANARI, U. 1994. Towards true concurrency semantics based transformation between CLP and CC. In *Proceedings of the International Workshop on Principles and Practice of Constraint Programming,* (Orcas Island, WA) Vol. 874 *LNCS*, A. Borning, Ed., Springer-Verlag.

BYON, J.-H. AND REVESZ, P. 1995. Disco: A constraint database system with sets. In *CONTESSA Workshop on Constraint Databases and Applications* (Sept.).

CARLIER, J. AND PINSON, E. 1989. An algorithm for solving the job shop problem. *Manage. Sci. 35.*

CHASSIN, J. AND CODOGNET, P. 1994. Parallel logic programming systems. *Comput. Surv. 26,* (3) (Sept.), 295–336.

CHEESEMAN, P., KANEFSKY, B., AND TAYLOR, W. 1991. Where the *really* hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence,* 331–337.

CODISH, M., FALASCHI, M., AND MARRIOTT, K. 1994. Suspension analysis for concurrent logic programs. *ACM Trans. Program. Lang. Syst. 16,* (3).

CODISH, M., FALASCHI, M., MARRIOTT, K., AND WINSBOROUGH, W. 1993. Efficient analysis of concurrent constraint logic programs. In *Proceedings of the International Colloquium on Automata, Languages and Programming* (Lund, Sweden, July).

CODOGNET, P. AND DIAZ, D. 1996. Compiling constraints in clp(fd). *J. Logic Program. 27,* 3.

CODOGNET, P. AND ROSSI, F. 1995. NMCC programming: Constraint enforcement and retraction in CC programming. In *Proceedings of the International Conference on Logic Programming.*

CODOGNET, C., CODOGNET, P., AND CORSINI, M. 1990. Abstract interpretation for concurrent logic languages. In *Proceedings of the North American Conference on Logic Programming.* MIT Press, Cambridge, MA.

COLMERAUER, A. 1990. An introduction to Prolog-III. *Commun. ACM 33,* (7).

DECHTER, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artif. Intell. 41,* 273–312.

DECHTER, R. 1992. Constraint networks. *Encyclopedia of Artificial Intelligence,* 276–285.

DECHTER, R. AND PEARL, J. 1987. Network-based heuristics for constraint satisfaction problems. *Artif. Intell. 34,* 1–38.

DECHTER, R. AND VAN BEEK, P. 1995. From local to global relational consistency. In *Proceedings of the International Conference on Constraint Programming,* (Cassis, France) Vol. 976, U. Montanari and F. Rossi, Eds. Springer-Verlag. A full version to appear in *Theor. Comput. Sci.*

DECHTER, R., MEIRI, I., AND PEARL, J. 1991. Temporal constraint networks. *Artif. Intell. 49,* 61–95.

DINCBAS, M., VAN HENTENRYCK, P., SIMONIS, H., AGGOUN, A., GRAF, T., AND BERTHIER, F. 1988. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo).

EUROPEAN COMPUTER RESEARCH CENTER. 1993. *Eclipse User's Guide.*

FALASCHI, M., GABBRIELLI, M., MARRIOTT, K., AND PALAMIDESSI, C. 1993. Compositional analysis for concurrent constraint programming. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science,* IEEE Computer Society Press, Los Alamitos, CA, 210–221.

FIKES, R. 1968. A heuristic program for solving problems stated as nondeterministic procedures. Ph.D. Thesis, Carnegie Mellon University.

FREUDER, E. C. 1978. Synthesizing constraint expressions. *Commun. ACM 21,* 11, 958–966.

FREUDER, E. C. 1982. A sufficient condition for backtrack-free search. *J. ACM 29,* 1, 24–32.

FREUDER, E. C. 1994. Exploiting structure in constraint satisfaction problems. In *Constraint Programming,* NATO ASI series, B. Mayoh, E. Tyugu, and J. Penjam, Eds., Springer-Verlag, 51–74.

FREUDER, E. C. 1996. *Principles and Practice of Constraint Programming—CP96.* No. 1118 in LNCS. Springer-Verlag.

FRÜHWIRTH, T. 1995. Constraint handling rules. In *Constraint Programming: Basics and Trends,* A. Podelski, Ed., No. 910 LNCS, Springer-Verlag, 90–107.

FRÜHWIRTH, T., HEROLD, A., KÜCHENOFF, V., LE PROVOST, T., AND LIM, P. 1992. Constraint logic programming—an informal introduction. In *Logic Programming in Action,* No. 636 LNCS, Springer-Verlag, 3–35.

GARCÍA DE LA BANDA, M., BUENO, F., AND HERMENEGILDO, M. 1996. Towards independent and-parallelism in CLP. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming* (Sept.), LNCS. Springer Verlag.

García de la Banda, M., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. 1997. Global analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst.* (to appear).

García de la Banda, M., Hermenegildo, M., and Marriott, K. 1993. Independence in constraint logic programs. In *Proceedings of the International Logic Programming Symposium* (Oct.) MIT Press, Cambridge, MA, 130–146.

García de la Banda, M., Marriott, K., and Stuckey, P. 1995. Efficient analysis of constraint logic programs with dynamic scheduling. In *Proceedings of the International Logic Programming Symposium* (Portland, OR, Dec.), MIT Press, Cambridge, MA.

Gaschnig, J. 1979. Performance measurement and analysis of search algorithms. Tech. Rep. CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA.

Gleicher, M. 1995. Practical issues in programming constraints. In *Principles and Practice of Constraint Programming: The Newport Papers,* V. A. Saraswat and P. Van Hentenryck, Eds., MIT Press, Cambridge, MA, 407–426.

Goldin, D. Q. and Kanellakis, P. C. 1996. Constraint query algebras. *Constraints.*

Govindarajan, K., Jayaraman, B., and Mantha, S. 1996. Optimization and relaxation in constraint logic languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages.*

Gross-Brunschwiler, R. 1996. Implementation of constraint database system using a compile-time rewrite approach. Ph.D. Thesis, ETH.

Grumbach, S. and Su, J. 1995. Dense-order constraint databases. In *Proceedings of the ACM Symposium on Principles of Database Systems.*

Grumbach, S., Su, J., and Tollu, C. 1995. Linear constraint databases. In *Proceedings of the LCC;* LNCS (to appear) Springer-Verlag.

Gupta, V., Jagadeesan, R., and Saraswat, V. A. 1996. Truly concurrent constraint programming. In *CONCUR96—Concurrency Theory,* U. Montanari and V. Sassone, Eds., Vol. 1119, *LNCS,* Springer-Verlag.

Gupta, V., Jagadeesan, R., and Saraswat, V. A. 1997. Computing with continuous change. *Sci. Comput. Program.* (to appear).

Gupta, V., Jagadeesan, R., Saraswat, V. A., and Bobrow, D. G. 1995. Programming in hybrid constraint languages. In *Hybrid Systems II,* Antsaklis, Kohn, Nerode, and Sastry, Eds., Vol. 999 *LNCS,* Springer Verlag.

Hansen, M. R., Hansen, B. S., Lucas, P., and van Emde Boas, P. 1989. Integrating relational databases and constraint languages. *Comput. Lang. 14,* 2, 63–82.

Haralick, M. and Elliot, J. 1980. Increasing tree-search efficiency for constraint satisfaction problems. *Artif. Intell. 14,* 263–313.

Haridi, S. and Janson, S. 1990. Kernel Andorra Prolog and its computational model. In *Proceedings of the International Conference on Logic Programming,* MIT Press, Cambridge, MA.

Helm, R., Huynh, T., Lassez, C., and Marriott, K. 1992. A linear constraint technology for interactive graphic systems. In *Graphics Interface '92,* 301–309.

Hermenegildo, M., Marriott, K., Puebla, G., and Stuckey, P. 1995. Incremental analysis of logic programs. In *International Conference on Logic Programming,* (June), MIT Press, Cambridge, MA, 797–811.

Hermenegildo, M. and The CLIP Group. 1994. Some methodological issues in the design of CIAO—a generic, parallel concurrent constraint system. In *Principles and Practice of Constraint Programming,* LNCS 874, May, Springer-Verlag, New York, 123–133.

Heydon, A. and Nelson, G. 1994. The Juno-2 constraint-based drawing editor. Tech. Rep. 131a, DEC Systems Research Center, Palo Alto, CA.

Holzbaur, C. 1992. Metastructures vs. attributed variables in the context of extensible unification. In *International Symposium on Programming Language Implementation and Logic Programming* (Aug.), Vol. 631 *LNCS,* Springer Verlag, 260–268.

Hosobe, H., Matsuoka, S., and Yonezawa, A. 1996. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming,* (Boston), E. C. Freuder, Ed., Vol. 1118, Springer-Verlag.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages,* ACM, New York.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *J. Logic Program. 19 & 20,* 503–581.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. An abstract machine for CLP(R). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* (San Francisco, June) 128–139.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. The CLP(R) Language and System. *ACM Trans. Program. Lang. Syst.*

Jampel, M., Freuder, E., and Maher, M. 1996. *Over-Constrained Systems.* LNCS, No. 1106, Springer-Verlag.

JEAVONS, P., COHEN, D., AND GYSSENS, M. 1996. A test for tractability. In *Principles and Practice of Constraint Programming,* E. C. Freuder, Ed., LNCS, No. 1118, Springer-Verlag, Boston, MA, 267–281.

JOUANNAUD, J. P., ED. 1994. *Proceedings of the First Conference on Constraints in Computational Logic (CCL).* LNCS, No. 845, Springer-Verlag.

KABANZA, F., STEVENNE, J.-M., AND WOLPER, P. 1990. Handling infinite temporal data. In *Proceedings of the ACM Symposium on Principles of Database Systems.*

KANELLAKIS, P., KUPER, G., AND REVESZ, P. 1995. Constraint query languages. *J. Comput. Syst. Sci.* 26–52.

KANELLAKIS, P., RAMASWAMY, S., VENGROFF, D. E., AND VITTER, J. S. 1993. Indexing for data models with constraints and classes. In *Symposium on Principles of Database Systems.*

KELLY, A. D., MACDONALD, A., MARRIOTT, K., STUCKEY, P. J., AND YAP, R. H. C. 1996. Effectiveness of optimizing compilation for clp(r). In *Proceedings of the Joint International Conference and Symposium on Logic Programming,* MIT Press, Cambridge, MA, 37–51.

KONDRAK, G. AND VAN BEEK, P. 1995. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence* (Montreal), 541–547.

KUPER, G. M. 1993. Aggregation in constraint databases. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming.*

LASSEZ, J.-L., MAHER, M. J., AND MARRIOTT, K. 1988. *Foundations of Deductive Databases and Logic Programming.* Chapter: Unification Revisited, Morgan-Kaufmann, San Mateo, CA.

LAURIERE, J.-L. 1978. A language and a program for stating and solving combinatorial problems. *Artif. Intell. 10,* 1.

LEVY, A., MUMICK, I. S., AND SAGIV, Y. 1994. Que-ry optimization by predicate movearound. In *Proceedings of the VLDB Conference.*

LEVY, A. AND SAGIV, Y. 1992. Constraints and redundancy in datalog. In *Proceedings of the ACM Symposium on Principles of Database Systems.*

LI, L.-L., REEVE, M., SCHUERMAN, K., VERON, A., BELLONE, J., PRADELLESS, C., PALASKAS, Z., STAMATOPOULOS, T., CLARK, D., DOURSENOT, S., RAWLINGS, C., SHIRAZI, J., AND SARDU, G. 1993. APPLAUSE: Applications using the ElipSys parallel CLP system. In *Proceedings of the International Conference on Logic Programming,* 847–848.

MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artif. Intell. 8,* 1.

MACKWORTH, A. K. 1992. Constraint satisfaction. In *Encyclopedia of Artificial Intelligence,* 285–293.

MACKWORTH, A. K. AND FREUDER, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Intell. 25.*

MACKWORTH, A. K. AND FREUDER, E. C. 1993. The complexity of constraint satisfaction revisited. *Artif. Intell. 25,* 57–62.

MAHER, M. J. 1987. Logic semantics for a class of committed-choice programs. In *Proceedings of the International Conference on Logic Programming,* MIT Press, Cambridge, MA.

MARRIOTT, K. AND STUCKEY, P. 1992. The 3 Rs of optimizing constraint logic programs: Refinement, removal, and reordering. In *Proceedings of the ACM Symposium on Principles of Programming Languages,* ACM, New York.

MARRIOTT, K., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Analyzing logic programs with dynamic scheduling. In *Proceedings of the ACM Symposium on Principles of Programming Languages,* (Jan.), ACM, New York, 240–254.

MCALOON, K. AND TRETKOFF, C. 1997. Logic, modeling and programming. *Ann. Oper. Res.* (to appear).

MEIER, M. 1996. *Grace User Manual.* Available at http://www.ecrc.de/eclipse/html/grace/grace.html.

MINTON, S. 1996. Automatically configuring constraint satisfaction programs: A case study. *Constraints 1/2,* 4–31.

MINTON, S., JOHNSTON, M., PHILIPS, A., AND LAIRD, P. 1992. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell. 58,* 161–206.

MONTANARI, U. 1970. Networks of constraints: Fundamental properties and application to picture processing. *Inf. Sci. 7,* 1974. Also Tech. Rep., Carnegie Mellon University, 1970.

MONTANARI, U. AND ROSSI, F. 1995a. A concurrent semantics for concurrent constraint programming via contextual nets. In *Principles and Practice of Constraint Programming: The Newport Papers.* V. A. Saraswat and P. Van Hentenryck, Eds., MIT Press, Cambridge, MA.

MONTANARI, U. AND ROSSI, F., EDS. 1995b. *Principles and Practice of Constraint Programming,* LNCS, Vol. 976, Springer-Verlag.

MUMICK, I. S., FINKELSTEIN, S. J., PIRAHESH, H., AND RAMAKRISHNAN, R. 1990. Magic conditions. In *Proceedings of the ACM Symposium on Principles of Database Systems,* 314–330.

MYERS, B. 1996. The Amulet user interface development environment. In *CHI'96 Conference*

Companion: Human Factors in Computing Systems, (Vancouver, BC, April), ACM SIGCHI, New York.

NEMHAUSER, G. AND WOLSEY, P. 1988. *Integer and Combinatorial Optimization.* J. Wiley and Sons, New York.

NEUMERKEL, U. 1990. Extensible unification by metastructures. In *Proceedings of the META'90 Workshop.*

PAI, D. K. 1991. Least constraint: A framework for the control of complex mechanical systems. In *Proceedings of the American Control Conference* (Boston, MA, June), 1615–1621.

PAREDAENS, J., VAN DEN BUSSCHE, J., AND VAN GUCHT, D. 1994. Towards a theory of spatial database queries. In *Proceedings of the ACM Symposium on Principles of Database Systems.*

RAMAKRISHNAN, R. 1988. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming.*

REVESZ, P. Z. 1993. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theor. Comput. Sci. 116,* (Aug.).

REVESZ, P. Z. 1995. Datalog queries of set constraint databases. In *Proceedings of the International Conference on Database Theory.*

ROSSI, F. AND MONTANARI, U. 1994. Concurrent semantics for concurrent constraint programming. In *Constraint Programming,* B. Mayoh, E. Tyugu, and J. Penjam, Eds., NATO ASI Series. Springer-Verlag.

SANNELLA, M. 1995. The SkyBlue constraint solver and its applications. In *Principles and Practice of Constraint Programming: The Newport Papers,* V. A. Saraswat and P. Van Hentenryck, Eds., MIT Press, Cambridge, MA, 385–406.

SANNELLA, M., MALONEY, J., FREEMAN-BENSON, B., AND BORNING, A. 1993. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Softw. Pract. Exper. 23,* 5 (May), 529–566.

SARASWAT, V. A. 1993. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Award and Logic Programming Series, MIT Press, Cambridge, MA.

SARASWAT, V. A., JAGADEESAN, R., AND GUPTA, V. 1995. Timed default concurrent constraint programming. *J. Symbol. Comput.* (to appear). Extended abstract appeared in *Proceedings of the ACM Symposium on Principles of Programming Languages* (San Francisco, Jan. 1995).

SARASWAT, V. A., RINARD, M., AND PANANGADEN, P. 1991. Semantic foundations of concurrent constraint programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (Orlando, FL, Jan.), ACM, New York.

SHAPIRO, E. 1982. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, Cambridge, MA.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3.

SMOLKA, G. 1995. The Oz programming model. In *Computer Science Today,* Jan van Leeuwen, Ed., LNCS, No. 1000, Springer-Verlag, Berlin, 324–343.

SRIVASTAVA, D. 1992. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Ann. Math. Artif. Intell.* (to appear).

SRIVASTAVA, D. AND RAMAKRISHNAN, R. 1992. Pushing constraint selections. In *Proceedings of the ACM Symposium on Principles of Database Systems,* 301–315.

SRIVASTAVA, D., RAMAKRISHNAN, R., AND REVESZ, P. 1994. Constraint objects. In *Proceedings of the International Workshop on the Principles and Practice of Constraint Programming* (Orcas Island, WA, May).

STALLMAN, R. M. AND SUSSMAN, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell. 9,* 135–196.

STEELE, G. L. 1980. The definition and implementation of a computer programming language based on constraints. Ph.D. Thesis, MIT.

SUTHERLAND, I. 1963. Sketchpad: A man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference.*

TARAU, P., DAVISON, A., DE BOSSCHERE, K., AND HERMENEGILDO, M., EDS. 1996. *Proceedings of the First Workshop on Logic Programming Tools for INTERNET Applications,* JICSLP '96 (Bonn).

TSANG, E. 1993. *Foundations of Constraint Satisfaction.* Academic Press, London.

VANDEURZEN, L., GYSSENS, M., AND VAN GUCHT, D. 1995. On the desirability and limitations of linear spatial query languages. In *Proceedings of the Symposium on Advances in Spatial Databases,* M. J. Egenhofer and J. R. Herring, Eds., *LNCS,* Vol. 951, Springer-Verlag, 14–28.

VAN HENTENRYCK, P. 1989a. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA.

VAN HENTENRYCK, P. 1989b. Parallel constraint satisfaction in logic programming. In *International Conference on Logic Programming* (Lisbon, Portugal, June), MIT Press, Cambridge, MA, 165–180.

VAN HENTENRYCK, P. 1991. Constraint logic programming. *Knowl. Eng. Rev. 6,* 6, 151–194.

VAN HENTENRYCK, P., SARASWAT, V. A., AND DEVILLE, Y. 1995. Constraint processing in cc(fd). In *Constraint Programming: Basics and Trends,* A. Podelski, Ed., LNCS 910, Springer Verlag.

WALTZ, D. L. 1975. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision,* P. Winston, Ed., McGraw-Hill.

ZAFFANELLA, E. 1995. Domain-independent ask approximations in CCP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* (Cassis, France), U. Montanari and F. Rossi, Eds., LNCS, No. 976, Springer-Verlag.

VANDER ZANDEN, B. 1996. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Trans. Program. Lang. Syst. 18,* 1 (Jan.), 30–72.

ZHANG, Y. AND MACKWORTH, A. K. 1994. Specification and verification of constraint-based dynamic systems. In *Principles and Practice of Constraint Programming,* A. Borning, Ed., LNCS, No. 874, Springer-Verlag, 229–242.

ZHANG, Y. AND MACKWORTH, A. K. 1995a. Constraint programming in constraint nets. In *Principles and Practice of Constraint Programming: The Newport Papers,* V. A. Saraswat and P. Van Hentenryck, Eds., MIT Press, Cambridge, MA, 49–68.

ZHANG, Y. AND MACKWORTH, A. K. 1995b. Synthesis of hybrid constraint-based controllers. In *Hybrid Systems II,* P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, Eds., LNCS, Vol. 999, Springer Verlag, 552–567.