Constraint Programming

The Constraint Programming Working Group*
ACM-MIT SDCR Workshop
published in ACM Computing Surveys, Dec. 1996

Oct 14, 1996

1 Introduction

A *constraint* can intuitively be thought of 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 $\{\langle 0, 9000\rangle, \langle 1500, 10500\rangle, \ldots\}$.

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, etc. Indeed, whole subfields of

*Editors: Pascal Van Hentenryck and Vijay Saraswat; Contributing Authors: 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, Ralph Wachter.

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 remarked above, 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, 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 non-directional: 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, 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 sixties and seventies which 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 two 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 brings a focus 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 system, to discovering common exploitable structure across constraint systems.

Operating around this level is the second level of *programming* language which allows the user to specify more information about which constraints should be generated, how they should be combined and processed etc. 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, which are finding wide-spread 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 underly 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 meta-theorems 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, though with rich (albeit disciplined) variation (via constraint systems).[1]

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

[1] ¿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.

The state of the art in CP is reported in international conference on Principles and Practice of Constraint Programming (PPCP) [99,38] and Practical Applications of Constraint Technology (PACT), and in the recently established CONSTRAINTS journal. 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 [61,40,69].

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 sixties and seventies. The pioneering works on *networks of constraints* were motivated mainly by problems arising in the field of *picture processing* [97,127]. 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 that were 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 [127]. The main algorithms developed in those years were related to achieving (variations of) *arc-* or *path-consistency* [97,89,87] (see Section 3.1). The former finds (and eliminates) values from variables' domains which are incompatible with some constraint concerning that variable, while the latter eliminates pairs of values which are allowed according to a given constraint c but not if one looks at a chain (a path) of constraints which start and end 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 which were solved using an algorithm. (Mention must also be made of the remarkably prescient systems REF-ARF [34] and ALICE [82]. Both provided simple but very useful constraint languages for specifying search problems, and solved them used customized constraint solvers with embedded propagation and search techniques.) However, we will 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 Sketchpad [123], 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 [8], included a facility for compiling constraint satisfaction plans, allowing constraints to be re-satisfied rapidly for changing inputs. EL [121] was an early constraint-based circuit analysis program. The concepts developed here led to a variety of other systems and languages, including Steele's constraint language [122], 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 where 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 [81]. This has led to the definition of a general framework, called Constraint Logic Programming (CLP) [68], which 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 [74].

Although the CLP scheme immediately gave rise to languages like CLP(R) [71] and Prolog III [25], it took the practical experience of application-oriented research to link CLP to the propagation algorithms developed earlier in AI. The language CHIP [59,31] realized that extensive use of early ideas on propagation was necessary at both the language and the implementation level 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, though without a clear declarative foundation. This is even more so in recently developed languages such as cc(fd)

[62], 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 [116].[2] 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 provided a breakthrough with his analysis that entailment lay at the heart of the synchronization mechanisms [90]. On this basis, Saraswat 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 constraints[112]. 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. On the other hand, CC provides a general declarative framework for concurrency encompassing and extending data-flow languages, languages based on "residuation" [2], 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 impulse to the development of new semantics for such languages, which 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 [7,114], and those based on truly concurrent models like Petri nets [98,52].

Languages based directly upon the CC idea are Oz [117], AKL [57], and, partly, CIAO [64]. However, the CC framework has to be seen more as a the-

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

oretical environment where 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) discussed above 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 today the ones that are most widely used 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 improvements in performance.

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 shall also attempt to emphasize the particular focii 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 [72], 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, e.g. 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 [27,86,125].

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 below) 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 [97,89,35,87,29] 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 will make 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 i^{th} 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, toward 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 (look-ahead schemes) and backtracking (look-back schemes) [26,79].

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 [56]. These methods, which vary in the strength of constraint inference (propagation), try to find a cost effective balance between pruning and overhead.

Look-back schemes are invoked when the algorithm encounters a dead end. These schemes perform two functions: One, decide how far to backtrack by analyzing the reasons for the dead end, a process often referred to as *backjumping*, [45]. Two, 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* [26,121].

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., [56]).

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 [96]. 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. While 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) [27,37].

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 meta-problem whose variables are subproblems of the original problem, and the cycle cutset method extracts a tree-structured subproblem from the original problem [27]. 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 uncertainity [30].

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., [36,29]).

The basic network structure that supports tractability is a generalized tree structure. This has been observed repeatedly from different perspectives, in constraint theory [88,27], complexity 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 [73].

Finally, special classes of constraints associated with temporal reasoning have received much attention in the last decade. Tractable classses include subsets of Allen's (qualitative) interval algebra [3], as well as quantitative binary linear inequalities over the reals, of the form $X - Y \leq a$ [28]. 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 [19]. 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 non-trivial 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 algorithms described above 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, spatial reasoning.
- Cognitive tasks including machine vision, natural language processing, planning.
- Task domains including scheduling, resource allocation, configuration, 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", that is, 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 [76]. 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 2- or 3-D geographic maps; geometric modeling objects for CAD/CAM; fields of vision of sensors; 4-D

(3 + 1 for time) trajectories of objects moving in 3-D space, based on the movements equations; translation of different systems of coordinates; operations research type models such as manufacturing patterns describing interconnections between 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 will 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; etc.

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., [13,77,118], and constraint algebra algorithms and global optimization (e.g., [12,47]). Examples of implemented constraint databases are [49,16].

While 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 [106]. The idea was refined in [4,100] to allow constraint propagation without actually storing constraints in the database, for the case of non-recursive 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 non-recursive) database queries for the purpose of optimization, such as pushing constraint selections in [119,83] or finding redundant parts of evaluation trees using query constraints in [84].

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 [55] considered polynomial equality constraints as rules, taking advantage of their adirectionality. [76] 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., [75]). More recent works on deductive databases (e.g., [100]) 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 [12]. The work [77] 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 [13] proposed an approach to achieve the optimal quality of constraint and spatial filtering. A number of works consider special constraint domains: integer order constraints [107]; set constraints [108]; dense-order constraints [50]. Linear constraints over reals have drawn special attention [1,12,51,126]. The use of constraints in spatial database queries was addressed in [105]. The work [120] used constraints to describe incomplete information. Constraint aggregation was studied in [80].

3.3 Constraint Programming in User Interfaces

Constraint programming has a long history of use in graphics and user interfaces, beginning with Sketchpad system [123]. 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 that 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, which 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 will satisfy the constraint. Examples of user interface toolkits using one-way constraints include Amulet [101] and its predecessor Garnet. Examples of multi-way local propagation algorithms include DeltaBlue [111], SkyBlue [110], and QuickPlan [129]. (These multi-way algorithms all also support constraint hierarchies [11,72], which allow for both required and preferential constraints. Constraint hierarchies are useful in such common User Interface (UI) tasks as specifying which parts of a figure we would 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 [58], which solves simultaneous linear equation and inequality constraints while optimizing a quadratic expression, Bramble [46] and Juno-2 [65] which use numerical solvers, Indigo [9], an interval propagation algorithm for inequality constraints, and DETAIL [67] and Ultraviolet [10], 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 which 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 [102]. 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 which 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 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, etc. 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 [82]. 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 might 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 [93].

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 trace back to [17] 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 which 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, labelling heuristics), personnel rostering (sequence constraints), workforce scheduling (constraint cooperation), circuit verification (Boolean constraints), electro-mechanical 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 mechanisms 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 received a lot of both theoretical and implementational 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 [112,7] or the *truly concurrent* ones of Petri nets and event structures [98,109]. Other theoretical efforts focus on the possibility of analyzing CC-like programs at compile-time and thus derive properties to be used at run-time. This holds, for example, for the works on *abstract interpretation* [128,22] which execute CC programs on an abstract constraint domain with the hope to derive some useful knowledge for program simplification, for those on *suspension analysis* [20], whose aim is to understand the conditions under which CC program deadlock, and for those on relating CC and CLP languages [15], which try to parallelize CLP programs using CC-based techniques or to sequentialize CC programs via an analysis of their inherent concurrency.

Languages like AKL [57], Oz [117], and CIAO [64] are essentially based on the CC ideas, although they add many features mainly because of application needs and of efficiency reasons. 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

If we examine this problem we see that 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 off-line model of computation. But intelligent systems embedded as controllers in real physical systems must be designed in an on-line model. Moreover, the on-line model must be based on various time structures: continuous, discrete and event-based. The requisite on-line 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 [130] 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 on-line constraint-satisfying devices [104,131]. 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 [132].

Another approach has been developed recently in [113,53] 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 which requires A to be triggered if the constraint c is *not* enforced now or through quiescence. This "non-monotonic" 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 [54].

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 will be 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's 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 analog of finite domains when reals are considered instead of integers. As it is impossible to explicitly represent 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 they 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 non linear constraints.

Linear constraints are constraints posted on real variables which 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), some other use floating point computations. The former is more accurate, while the latter is more efficient. Interior point methods have been introduced in linear programming libraries but have not impacted constraint programming more generally.

"Global" constraints The removal of inconsistent values can be tricky for more complex constraints. An important line of work aims to define good propagation algorithm for more complex constraints. This is sometime referred to as global constraints. In this context, scheduling constraints, all-different (a set of variables takes on values that are all different), and 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 sophisticated algorithm adapted from Operations Research.

User-Defined Constraints. One of the lessons learned so far from the application of CP tools in practice is 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 in hand. Several proposal have been made for making it possible for the user to add domain specific constraints to the system and to 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) [62] and clp(fd) [23]). Some others propose to view a constraint as a Boolean expression. The Boolean variable is true if the constraint is necessarily true (entailed by the other constraints). The Boolean variable is false if the negation of the constraint is entailed by the other constraints. This enables 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 [39]. 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* [103,66]. This approach is used extensively in the implementation of constraint solvers in systems such as ECL^{i}PS^e[32], SICStus, and CIAO [64]. A last approach is motivated by the need for adding 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 was called the "No Box" approach of Puget and Leconte, and potentially yields the most efficient implementations, although implying a higher programming load.

3.7.2 Constraint Programming Tools

The constraint systems discussed above 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, etc.

Another popular approach is to embed CLP techniques in a different host language, leading to another set of tools which includes the following (for each of them we indicate both the underlying programming language together with 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: non-deterministic 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, also tools which are 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 [115], often suffice 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 which 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., [42]), as well as, perhaps, other proof techniques previously used in logic programming (e.g., based on induction assertion). A run-time technique which is currently receiving much attention is the use of visualization, both of the search space and of the constraint store at different points of execution [94].

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 which 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 (e.g., [70,62,31,23]. The target abstract machines used are most often generalization 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 where 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 (non constraint 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., [92,41]), practical frameworks for global analysis developed (e.g., [42]), and some CP systems have been reported which perform global analysis based optimization [78,41]. 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 [21,33,91,15,44]. Finally, recent progress in incremental global analysis (e.g., [63]) 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 which has shown significant speedups in the context of logic programs is automatic parallelization [18]. Exploitation of parallelism in the search (or-parallelism) is comparatively easy and has been shown to provide speedups in several industrial applications containing extensive search [60,32,85]. 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 [43], notions of independence appropriate for (concurrent) CP have been recently proposed [43,14]. 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 [41].

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 is mind, we may 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 [72], 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 [6,11,48], and/or the language could allow for user-guided constraint retraction [24,5] 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 problem-solving techniques are now available to us, but synthesizing appropriate algorithms for specific tasks should be automated [95]. In addition, robust constraint computation must cope with change in the world and in models, and with noise (e.g., in data), and uncertainity (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 the fact 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 like the ability of describing intra- and inter-process communication and synchronization are more and more important in practical applications which consist of distributed environments where both local problem solving and global synchronization and coordination is needed. This is added to the fact that many CP systems already offer many other characteristics that make them well suited in this context. These include 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 meta-programming. 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 [124]. 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 between objects (e.g. attachments, minimal distances, non-collision, etc) or general integrity rules (such as energy conservation laws) can be considered as constraints, and implemented efficiently as such. This generalizes in an obvious way 2D geometrical constraints. 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. While the performance and computing resource economy of current CP systems has proved to be adequate in significant industrial applications, competing very favorably with other techniques and approaches, it appears that there still remain many avenues for improvement, which would make the technology even more competitive. It is expected that improving execution speed and reducing further resource consumption can 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 starting to be in many cases the default installation platform (for example, 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 (specially if the promise of optical interconnect 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 solving algorithms also offers an interesting research direction. Currently, at least one European project has started 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 as needed evolving those systems to allow for clearer and more easily understood user models.

Acknowledgments. We gratefully acknowledge very valuable and timely comments from Thomas Frühwirth, John Hooker, Michael Maher, Catuscia Palamidessi, Peter Revesz, Mark Wallace, Peter van Beek and Roland Yap. Particular thanks to Vineet Gupta for invaluable last-minute assistance.

Subject: Final version...tex file.

References

[1] F. Afrati, S. Cosmadakis, S. Grumbach, and G. Kuper. Linear versus polynomial constraints in database query languages. In A. Borning, editor, *Proceedings of the International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 181–192, Orcas Island, WA, 1994. Springer Verlag.

[2] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3&4):195–234, 1993.

[3] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[4] I. Balbin, D.B. Kemp, K. Meenakshi, and K. Ramamohanarao. Propagating constraints in recursive deductive databases. In *Proceedings of the North American Conference on Logic Programming*, 1989.

[5] E. Best, F.S. De Boer, and C. Palamidessi. Concurrent constraint programming with information removal. In *First Conference on Concurrent Constraint Programming*, Venice, 1995.

[6] S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Morgan Kaufman, 1995.

[7] F.S. De Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings of the CAAP*. Springer-Verlag, 1991.

[8] A. Borning. The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.

[9] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Seattle, November 1996.

[10] A. Borning and B. Freeman-Benson. The OTI constraint solver: A constraint library for constructing interactive graphical user interfaces. In U. Montanari and F. Rossi, editors, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 976, pages 624–628, Cassis, France, 1995. Springer-Verlag.

[11] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[12] A. Brodsky, J. Jaffar, and M.J. Maher. Toward practical constraint databases. In *Proceedings of the International Conference on Very Large Data Bases*, Dublin, 1993.

[13] A. Brodsky, C. Lassez, J.-L. Lassez, and M. J. Maher. Separability of polyhedra for optimal filtering of spatial and constraint data. In *Proceedings of the ACM Symposium on Principles of Database Systems*. ACM Press, 1995.

[14] F. Bueno, M. Jose Garcia de la Banda, M. Hermenegildo, U. Montanari, and F. Rossi. From eventual to atomic and locally atomic CC programs: A concurrent semantics. In *Proceedings of the International Conference on Algebraic and Logic Programming*, 1994.

[15] F. Bueno, M. Jose Garcia de la Banda, M. Hermenegildo, F. Rossi, and U. Montanari. Towards true concurrency semantics based transformation between CLP and CC. In A. Borning, editor, *Proceedings of the International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, Orcas Island, WA, 1994. Springer-Verlag.

[16] J.-H. Byon and P. Revesz. Disco: A constraint database system with sets. In *CONTESSA Workshop on Constraint Databases and Applications*, September 1995.

[17] J. Carlier and E. Pinson. An algorithm for solving the job shop problem. *Management Science*, 35, 1989.

[18] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[19] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. In *Proceedings of the International Joint Conference on Artificial Iintelligence*, pages 331–337, 1991.

[20] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[21] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, Lund, Sweden, July 1993.

[22] C. Codognet, P. Codognet, and M. Corsini. Abstract interpretation for concurrent logic languages. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, 1990.

[23] P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3), 1996.

[24] P. Codognet and F. Rossi. NMCC Programming: Constraint Enforcement and Retraction in CC Programming. In *Proceedings of the International Conference on Logic Programming*, 1995.

[25] A. Colmerauer. An introduction to Prolog-III. *Communication of the ACM*, 33(7), 1990.

[26] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[27] R. Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, pages 276–285, 1992.

[28] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[29] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[30] R. Dechter and P. van Beek. From local to global relational consistency. In U. Montanari and F. Rossi, editors, *Proceeding of the International Conference on Constraint Programming*, volume 976, Cassis, France, 1995. Springer-Verlag. A full version to appear in *Theoretical Computer Science*, 1996.

[31] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. Tokyo, Japan, 1988.

[32] European Computer Research Center. *Eclipse User's Guide*, 1993.

[33] Moreno Falaschi, Maurizio Gabbrielli, Kim Marriott, and Catuscia Palamidessi. Compositional analysis for concurrent constraint programming. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science*, pages 210–221. IEEE Computer Society Press, 1993.

[34] R. Fikes. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures.* PhD thesis, Carnegie Mellon University, 1968.

[35] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

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

[37] E. C. Freuder. Exploiting structure in constraint satisfaction problems. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, NATO ASI series, pages 51–74. Springer-Verlag, 1994.

[38] E. C. Freuder, editor. *Principles and Practice of Constraint Programming–CP96*, number 1118 in LNCS. Springer-Verlag, 1996.

[39] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 90–107. Springer-Verlag, 1995.

[40] T. Frühwirth, A. Herold, V. Küchenoff, T. Le Provost, and P. Lim. Constraint logic programming – an informal introduction. In *Logic Programming in Action*, number 636 in LNCS, pages 3 – 35. Springer-Verlag, 1992.

[41] M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards independent And-Parallelism in CLP. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, LNCS. Springer Verlag, September 1996.

[42] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. In *ACM Transactions on Programming Languages and Systems*. ACM, 1996. To appear.

[43] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *Proceedings of the International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[44] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *Proceedings of the International Logic Programming Symposium*, Portland, OR, December 1995. MIT Press, Cambridge, MA.

[45] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, Pa., 1979.

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

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

[48] K. Govindarajan, B. Jayaraman, and S. Mantha. Optimization and Relaxation in Constraint Logic Languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1996.

[49] R. Gross-Brunschwiler. *Implementation of Constraint Database system using a compile-time rewrite aproach*. PhD thesis, ETH, 1996.

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

[51] S. Grumbach, J. Su, and C. Tollu. Linear constraint databases. In *Proceedings of the LCC; To appear in LNCS Springer-Verlag volume*, 1995.

[52] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Truly concurrent constraint programming. In U. Montanari and V. Sassone, editors, *CONCUR96 – Concurrency Theory*, volume 1119 of *LNCS*, 1996.

[53] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, To appear.

[54] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. G. Bobrow. Programming in hybrid constraint languages. In Antsaklis, Kohn, Nerode, and Sastry, editors, *Hybrid Systems II*, volume 999 of *LNCS*. Springer Verlag, November 1995.

[55] M.R. Hansen, B.S. Hansen, P. Lucas, and P. van Emde Boas. Integrating relational databases and constraint languages. *Computer Languages*, 14(2):63–82, 1989.

[56] M. Haralick and J. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[57] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computational Model. In *Proceedings of the International Conference on Logic Programming*. MIT Press, 1990.

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

[59] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[60] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[61] P. Van Hentenryck. Constraint logic programming. *Knowledge Engineering Review*, 6(6):151 – 194, 1991.

[62] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(fd). In A. Podelski, editor, *Constraint Programming : Basics and Trends.* Springer Verlag, LNCS 910, 1995.

[63] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[64] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[65] A. Heydon and G. Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, DEC Systems Research Center, Palo Alto, CA, 1994.

[66] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 260–268. Springer Verlag, August 1992.

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

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

[69] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19 & 20:503–581, 1994.

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

[71] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 1992.

[72] M. Jampel, E. Freuder, and M. Maher, editors. *Over-Constrained Systems*, number 1106 in LNCS. Springer-Verlag, 1996.

[73] P. Jeavons, D. Cohen, and M. Gyssens. A test for tractability. In E. C. Freuder, editor, *Principles and Practice of Constraint Programming*, number 1118 in LNCS, pages 267–281. Springer-Verlag, Boston, MA, 1996.

[74] J.P. Jouannaud, editor. *Proceedings of the 1st Conference on Constraints in Computational Logic (CCL)*, number 845 in LNCS. Springer-Verlag, 1994.

[75] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1990.

[76] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, pages 26–52, 1995.

[77] P. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for data models with constraints and classes. In *Symposium on Principles of Database Systems*, 1993.

[78] A.D. Kelly, A. Macdonald, K. Marriott, P.J. Stuckey, and R.H.C. Yap. Effectiveness of optimizing compilation for clp(r). In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

[79] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of the International Joint Conference on Artificial Iintelligence*, pages 541–547, Montreal, 1995.

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

[81] J.-L. Lassez, M.J. Maher, and K. Marriott. *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann Publishers, Inc, 1988.

[82] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1), 1978.

[83] A. Levy, I.S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proceedings of the VLDB Conference*, 1994.

[84] A. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1992.

[85] Liang-Liang Li, Mike Reeve, Kees Schuerman, André Véron, Jacques Bellone, Claudine Pradelless, Zissis Palaskas, Takis Stamatopoulos, Dominic Clark, S. Doursenot, Chris Rawlings, Jack Shirazi, and Guisseppe Sardu. APPLAUSE: Applications using the ElipSys parallel CLP system. In *Proceedings of the International Conference on Logic Programming*, pages 847–848, 1993.

[86] A. K. Mackworth. Constraint satisfaction. *Encyclopedia of Artificial Intelligence*, pages 285–293, 1992.

[87] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[88] A. K. Mackworth and E. C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 25:57–62, 1993.

[89] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.

[90] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of the International Conference on Logic Programming*. MIT Press, 1987.

[91] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[92] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 1992.

[93] K. McAloon and C. Tretkoff. Logic, modeling and programming. *Annals of Operations Research*, To appear.

[94] M. Meier. *Grace User Manual*, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`.

[95] S. Minton. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1/2:4–31, 1996.

[96] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.

[97] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7, 1974. Also Technical Report, Carnegie Mellon University, 1970.

[98] U. Montanari and F. Rossi. A concurrent semantics for concurrent constraint programming via contextual nets. In V. A. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*. MIT Press, 1995.

[99] U. Montanari and F. Rossi, editors. *Principles and Practice of Constraint Programming*, number 976 in LNCS. Springer-Verlag, 1995.

[100] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 314–330, 1990.

[101] B. Myers. The Amulet User Interface Development Environment. In *CHI'96 Conference Companion: Human Factors in Computing Systems*, Vancouver, B.C., April 1996. ACM SIGCHI.

[102] G. Nemhauser and P. Wolsey. *Integer and Combinatorial Optimization*. J. Wiley and Sons, 1988.

[103] U. Neumerkel. Extensible Unification by Metastructures. In *Proceeding of the META'90 workshop*, 1990.

[104] D. K. Pai. Least constraint: A framework for the control of complex mechanical systems. In *Proceedings of the American Control Conference*, pages 1615–1621, Boston, MA, June 1991.

[105] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1994.

[106] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, 1988.

[107] P. Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1), August 1993.

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

[109] F. Rossi and U. Montanari. Concurrent semantics for concurrent constraint programming. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, NATO ASI Series. Springer-Verlag, 1994.

[110] M. Sannella. The SkyBlue constraint solver and its applications. In V. A. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 385–406. MIT Press, 1995.

[111] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[112] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993. ACM Doctoral Dissertation Award and Logic Programming Series.

[113] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, To appear. Extended abstract appeared in Proceedings of the ACM Symposium on Principles of Programming Languages, San Francisco, January 1995.

[114] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991. ACM.

[115] E. Shapiro. *Algorithmic Program Debugging.* ACM Distiguished Dissertation. MIT Press, 1982.

[116] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Survey*, 21(3), 1989.

[117] G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 324–343. Springer-Verlag, Berlin, 1995.

[118] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence, to appear*, 1992.

[119] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 301–315, 1992.

[120] D. Srivastava, R. Ramakrishnan, and P. Revesz. Constraint objects. In *Proceedings of the International Workshop on the Principles and Practice of Constraint Programming*, Orcas Island, WA, May 1994.

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

[122] G. L. Steele. *The definition and implementation of a computer programming language based on Constraints*. PhD thesis, MIT, 1980.

[123] I. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the IFIP Spring Joint Computer Conference*, 1963.

[124] P. Tarau, A. Davison, K .De Bosschere, and M. Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP '96, Bonn, 1996.

[125] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

[126] L. Vandeurzen, M. Gyssens, and D. Van Gucht. On the desirability and limitations of linear spatial query languages. In M. J. Egenhofer and J. R. Herring, editors, *Proceedings of the Symposium on Advances in Spatial Databases*, volume 951 of *LNCS*, pages 14–28. Springer Verlag, 1995.

[127] D. L. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975.

[128] E. Zaffanella. Domain independent ask approximations in CCP. In U. Montanari and F. Rossi, editors, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, number 976 in LNCS, Cassis, France, 1995. Springer-Verlag.

[129] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

[130] Y. Zhang and A. K. Mackworth. Specification and verification of constraint-based dynamic systems. In A. Borning, editor, *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 229 – 242. Springer-Verlag, 1994.

[131] Y. Zhang and A. K. Mackworth. Constraint programming in constraint nets. In V. A. Saraswat and P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 49–68. The MIT Press, Cambridge, MA, 1995.

[132] Y. Zhang and A. K. Mackworth. Synthesis of hybrid constraint-based controllers. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, number 999 in LNCS, pages 552 – 567. Springer Verlag, 1995.