1. Introduction

1.1 Constraint Satisfaction and Constraint Programming: A Brief Lead-In

Brian Mayoh[1], Enn Tyugu[2] and Tarmo Uustalu[2]

[1]Aarhus University, Computer Science Dept,
Ny Munkegade 540, DK-8000 Aarhus, Denmark,
brian@daimi.aau.dk

[2]The Royal Institute of Technology, Dept of Teleinformatics,
Electrum 204, S-164 40 Kista, Sweden

This paper presents the authors' vision about the achievements and expected further developments in the paradigm and techniques of constraint solving, and in applying these in programming

1.1.1 Introduction

Recently, constraints have become a hot topic in several computer science communities. Constraints are fashionable these days. However, apart from this trendy side of the matter, we believe the field to have a steadier significance. After all, the rise of the constraint paradigm resulted from certain developments within AI and computing science. Constraint solving is basically about search in huge search spaces, which in bad cases possess little or almost no guidance-providing structure, and lots of practical problems that daily pop up in AI applications and in computing in general can only be solved by wise search management.

One possible way to classify various constraint problems is the following:

- *Synthesis problems.* We are given requirements in a form of a huge set of constraints, and our task is to find an object that satisfies this set (fully, or at least the important ones of them, or all, but only up to certain precision). Examples are simulation (=reconstruction), where the laws (e.g. physical) that govern the situation or process under investigation are the

constraints, and various design (=construction) problems, where the requirements on the end product and the properties of the available construction elements are the constraints. For example, in program synthesis, the I/O relations of the expected program and of the available modules act as constraints; in scheduling, constraints are temporal.

- *Analysis problems.* These are about objects that have visible attributes (an outer appearance) and invisible attributes (an inner essence). The relationships between the visible and invisible attributes are known to us. Our task is, given the values of some object's visible attributes, find what the values of its invisible attributes could be. Examples are all sorts of fault diagnosis problems (medical, technical, etc.), where malfunction manifestations are the visible attributes and faults are the invisible ones; and vision, where a bitmap is what we see, and the lines, angles, shapes etc. that give rise to this, are the underlying essence we are interested in.

Problems that involve much search will always remain, and this sets forth three challenges:

- Find out commonalities between different approaches that tackle search problems, and from upon this basis, advance the general *philosophy* of constraint manipulation. Because of the infamous trade-off between generality and usefulness (efficiency), this requires delicacy, but can give valuable broad insights. Besides, this helps to develop common terminology, and to avoid duplication in the efforts that different communities undertake.

- Seek for efficient constraint satisfaction *algorithms*, both universally applicable and domain-specific. This is very practical and of immediate use.

- Make machines solve our problems elegantly and efficiently, *program machines in new ways.*

A good philosophy of constraints is a basis for constraint programming languages that users will like, and good algorithms are a basis for constraint programming languages that machines will like.

This sketch is a brief presentation of our subjective vision about these three challenges, i.e. about the achievements and expectable further developments in the paradigm and techniques of constraint solving, and in applying these in programming.

1.1.2 How It All Started

The concept of a constraint network was formed gradually. Sketchpad (Sutherland 1963), one of the first interactive graphical interfaces, solved geometrical constraints. Relations in the form of equations or tables were used as problem specifications in several CAD systems, but no generalization was made.

One of the earliest generalizations close to constraint networks of today was the concept of computational models, which was initially developed for specifications of engineering problems and used in a problem solver Utopist (Tyugu 1970), which was a value propagation planner.

Research in image processing led Montanari to the first systematic algebraic treatment of constraint networks (Montanari 1974), which originally appeared as a technical report already in 1970, and contained a path consistency algorithm. A very basic consistency technique—Waltz filtering—originated also from a work on image processing (Waltz 1972).

An IFIP workshop in Grenoble in 1978 on applications of AI and pattern recognition in computer aided design (Latombe 1978) gave strong impetus to the research in constraint solving. Present were a number of people who later have made significant contributions in the area: Montanari, Sussman, McDermott etc. The AI people discussed intelligent problem-solving with engineers and discovered a very promising application field for their methods. In particular, applications based on value propagation (then called constraint propagation) ideas were discussed, and a number of reports on research in this direction was published thereafter. A good example system is CONSTRAINTS (Sussman, Steele Jr. 1980).

Elegant, but for some reason not widely acknowledged work in finite-domain constraint satisfaction was done in France in the end of the 1970s by Lauriere, who developed a system called ALICE and applied it to several practical problems, including prediction/detection problems in geology (Lauriere 1978).

1.1.3 The Constraint Satisfaction Problem (CSP)

There is much confusion in constraint terminology, partly for historical reasons, partly due to the young age of the area. A *constraint network (CN)* (if cleanly described) involves three components:

- *Variables.* These are something that have names and can take values which are elements of some universal domain.
- *Constraints.* These, too, have names, and can take values. Their values are relations of finite arity on the universal domain, and these values are usually given. A relation can be given either extensionally (by plainly enumerating the tuples it contains) or intensionally (by some effective characterization of its extension).
- *A connection (binding) function.* This important component of a network is a function from constraint names to tuples of variable names.

A *valuation* is a value assignment for the variables of a given CN, i.e. a function that maps the variables into elements of the universal domain. A valuation *satisfies a constraint* if, under it, the constraint holds on the variables it connects. A valuation *satisfies a CN* if it satisfies all of its constraints. A goal

of *solving* a CN is to find either one satisfying valuation (*solution*) or all such ones, and this is obviously a search problem formulation.

It is quite straightforward to represent CNs as *labelled hypergraphs*. Variable names and values correspond to nodes and their labels respectively. Similarly, constraint names and values correspond to hyperarcs and their labels respectively. Finally, the connection function of a CN corresponds to the incidency function of a hypergraph.

1.1.4 Relation to Logic

There is a clear meaning for CNs in 1st order predicate logic (FOPL). Variables correspond to (individual) variables, constraints correspond to predicates, and the connection function helps to turn the "name part" of the CN into a formula, according to the following prescription. Guided by connections, form atoms from the predicate and variable names, and conjoin these. The interpretation of the formula is partly open: the interpretations of the predicates are pre-determined, while the interpretations of the variables are yet to be found.

As classical CSP benchmark, the Zebra problem, is described in the next chapter; figure 1.4.1 gives its constraint net, and figure 1.4.2 gives the logical reformation

In logical terms, CN solving amounts to completing a partial model. In fact, we need not speak about partial models, but just restrict the class of models we consider to those where the predicates are interpreted as prescribed by a given CN. In this context, CN solving is simply model-construction.

Algorithms for model-construction are something that usually is too down-to-earth to interest pure logicians. The issue of whether a model class can be constructed for some theory, i.e. consistency of a theory, is, of course, of high importance, but the question whether a model can be found for some simple formula of FOPL in some given model class, and if yes, then how, is not interesting.

In applied logic, however, the situation is different. In applications of logic to computer science (e.g. in program reasoning), one often faces the following problems, and therefore there is research going on in finding efficient algorithms to solve them:

- *Model-checking.* Given a formula and a model, check whether the formula is satisfied by the model. E.g. given a specification and a program, does the program meet the specification?

- *Model-construction.* Given a formula, construct a model that satisfies it. E.g. given a specification, construct a program that meets it.

- *Entailment-checking.* Given two formulae, does one entail the other (wrt the model class under consideration)? E.g. given two specifications, does one refine the other?

- *Entailer-construction.* Given a formula, construct another that entails it (the other formula must be in some sense better manageable). E.g. given a specification, construct another that refines it.

On the basic level, CN solving is mostly about model-construction, as we have pointed out. But there are signs already, that in the future, the three other listed problems, especially the third, gain acuteness.

The formulae that result from CNs are of very simple structure—conjuctions of atoms. Model-checking for such formulae is usually quite trivial (it depends on how the relations are given). But this is not the case with model-checking in general: verification of a transition system against a temporal-logic-of-actions formula is far from trivial. The situtation may change also in the constraint community, when partial constraint satisfaction (see subsection 1.1.6) and higher-order constraints (see subsection 1.1.7) gain more attention.

In the cc programming paradigm (see subsection 1.1.8.2), one of the two fundamental operations that can be applied to a program's data state (which is a constraint store) is `ask`, and this checks whether the store yields a given constraint, i.e. in logical terms, whether one formula entails another.

Stepwise specification refinement is a technique in program synthesis relying on entailer-construction. Roughly speaking, a given initial specification is gradually transformed into stronger and more specific ("finer") versions, the last of which is turned into a program. In constraint solving there is a clear analogue to specification refinement—the consistency techniques (see subsection 1.1.5.1). These techniques gradually transform a CN into more and more explicit versions (atoms in the corresponding formulae become more and more restricting), and the last version is solved directly (a model for the last formula is found directly).

As a part of the work on the foundations of the cc framework (see subsection 1.1.8.2), Saraswat et al. have been developing a theory of *constraint systems* as logical theories (Panangaden et al. 1991; Saraswat 1992a). A constraint system consists of a set of *tokens*, all carrying partial information about certain states-of-affairs, and an *entailment relation* between finite sets of tokens. Constraints on states-of-affairs can be stated as finite sets of tokens (understood as conjunctions of primitive constraints). The only requirements that an entailment relation is required to fulfill are transitivity and the property that any set of tokens must entail all of its subsets. In constraint systems, the concept of constraint is very general: one abstracts away from the extensions of constraints, being interested only in the entailment relation between them.

These were some links between CN solving and logic. Probably there will be many more upcoming. From among the work done within the AI community, we can mention (Bibel 1988; Mackworth 1992). The latter paper presents a very basic comparison of various ways of treating finite-domain constraint satisfaction in terms of different fragments of FOPL (including propositional logic).

1.1.5 CN Solving Techniques

Provided that the universal domain can be effectively enumerated, the most straightforward CN solving technique is backtrack search for a satisfying valuation. Blind backtrack search is only applicable, if the domain is finite, since in this case such search is always terminating. But even for finite domains, backtrack search is grossly inefficient, and therefore researchers have tried to develop more efficient algorithms for various special cases. Infinite domains require domain-specific algorithms.

Many algorithms assume binary CNs. A CN is *binary* if all its constraints are either unary or binary. Another assumption often made in algorithms is that, for every subset of the variables of a CN, there is exactly one constraint connecting them. Any CN can easily be transformed into such a form. If, for some subset of variables, there are several connecting constraints, replace these by one, whose extension is the intersection of the old ones. If, for some subset of variables, there is no connecting constraint, then connect them with a mock one of suitable arity, which is universally true (i.e. essentially non-constraining).

1.1.5.1 Consistency Techniques A major idea in the various improvements to the brute-force backtracking is to cut down the search space by first modifying the original CN into a more "explicit" one, and then running brute-force backtracking on this new CN. The modification is done by repeatedly enforcing consistency on certain (small) sub-CNs with distinguished constraints. A sub-CN is *consistent* if every solution of its distinguished constraint can be extended to a solution of the sub-CN (what an inaccurate term from the logical point-of-view!!). A sub-CN is made consistent by tightening up the distinguished constraint (i.e. by restricting its extension). It is easy to see that each solution of the modified CN is a solution of the original CN. Moreover, if consistency is enforced carefully (and most algorithms do that), then even the converse holds, and the original and modified CNs are equivalent. As the extensions of constraints become smaller in the course of modification, backtrack search on the modified CN is more efficient than on the original CN (it becomes possible to backtrack earlier in the failing branches).

The early consistency-enforcing algorithms processed arcs and paths. An *arc* is a sub-CN consisting of two variables, two unary constraints (one per each variable), and a binary constraint, the distinguished constraint being one of the unary constraints. A *path* is a sub-CN consisting of three variables, three unary constraints (one per each variable), and three binary constraints (one per each (unordered) pair of variables), the distinguished constraint being one of the binary constraints. Making an arc or path consistent amounts to applying simple operations on relations, much like those that one encounters in relational databases. A CN is called *arc-consistent* (resp. *path-consistent*) if all its arcs (resp. paths) are consistent. The goal of an arc- (resp. path-)consistency algorithm is to make a CN arc- (resp. path-)consistent.

A problem with making one sub-CN consistent is that this may make other sub-CNs inconsistent. That is why repetitions are generally needed if we insist on achieving simultaneous consistency of several sub-CNs. By subtle bookkeeping over the changes that modifications introduce into the original CN, one can get algorithms with low worst-case time complexity, but the space complexity increases. The first arc- and path-consistency algorithms, AC-1, AC-2, AC-3, and PC-1, PC-2 were proposed in (Mackworth 1977; Mackworth, Freuder 1985). They were impoved by Mohr and Henderson's (1986) AC-4 and PC-3 . Finally, Van Hentenryck, Deville and Teng (1992) gave a generic arc-consistency algorithm AC-5, which can be instantiated to reduce to AC-3 and AC-4, and, for a number of important special classes of constraints (functional, anti-functional, and monotonic constraints, and constraints that are piecewise of any one of these kinds), can be instantiated to yield special fast algorithms.

If a CN (as a hypergraph) is dense, arc- and path-consistency algorithms may not improve the CN, and one might be tempted therefore to try to make larger sub-CNs consistent. Here a difficulty arises. Sub-CNs with more than two variables generally involve loops of constraints, and so a general algorithm for solving them is backtrack search. We face a dilemma: either to pre-process a CN extensively (which involves some backtrack search), and have the search space for the final backtrack search for the solutions of the modified CN smaller, or to pre-process less, and do all the backtrack search in the final end and in a larger search space. To choose adequately the sub-CNs to be made consistent is a critical problem. Guidance can be sought from the global structure of the network (see Section 1.1.5.2).

Some authors (especially Montanari and Rossi) used to call consistency-enforcing '*relaxation*', though individual constraints become tighter in this process and the overall CN typically remains equivalent to the original one. This was motivated by the consistency-enforcing process being one always dampening in a stable state where no futher changes can occur. In a way, 'relaxation' is a more beautiful term than 'consistency-enforcing', but it must be noted that a number of researchers apply the word 'relaxation' in relation to weakening of constraints in partial constraint satisfaction (see subsection 1.1.6), which is a very different thing.

Yet another name for consistency-enforcing, '*consistency propagation*', is most adequate in situations where CNs can be made consistent "in one pass", without repetitions, e.g. in case of tree-structured binary CNs.

An important special form of (hyperedge) consistency enforcing is *value propagation*, which is applicable if the constraints of a CN are functional, i.e. if the value of some one variable participating in a constraint becomes uniquely determined once the values of the other variables of that constraint are known. If it is known in advance that a CN has a solution (e.g. in analysis situations), so that no conflicts can arise despite that in propagation there possibly are several potential sources for values of some variables, the variable values can be decided in one pass.

1.1.5.2 Network Structure Based Techniques Dechter and Pearl have worked on how to exploit the structure of a CN in choosing an appropriate tactic for solving a CN. They have proposed a number of techniques, which include the following:

- *Adaptive consistency enforcing.* This is a consistency technique that avoids repeated considering of sub-CNs. The sub-CN to be made consistent next is decided at run-time.

- *Cycle-cutset decomposition.* This technique is based on two facts: one is that by fixing the values of certain variables, the connectivity of a CN can be decreased, and the other is that tree-structured CNs can be solved very efficiently (by a repetition-free arc-consistency algorithm).

- *Tree clustering.* This technique operates on the so-called dual graphs of binary CNs.

For descriptions of these techniques, see (Dechter, Pearl 1988; Dechter, Pearl 1989; Dechter 1990).

In (Montanari, Rossi 1991a) it is pointed out that, if a CN was formed incrementally by a series of substitutions of smaller CNs for single constraints (in graph terms, by hyperedge replacements), then this CN can be solved without repetitions, by making the "building blocks" consistent in the order inverse to that of substitutions. Montanari and Rossi call this *perfect relaxation.* The problem with perfect relaxation is: how to find a appropriate decomposition of a given CN into a series of substitutions, such that it is not too costly to make the substituted CNs consistent. In some cases, however, the decomposition ("the evolution history") of a CN is known, and then perfect relaxation may be useful. An natural example of an evolving CN is the constraint store in CLP (see Section 1.1.8.1) (Montanari, Rossi 1991b).

1.1.5.3 Domain-Specific Techniques We do not intend to say much on domain-specific techniques. Although most practical and much exploited, they are not too interesting from a philosophical point of view due to their limited applicability.

The best-studied domain is *rational/real arithmetic.* Typically, linear equations and inequalities are considered, and solving systems of these is the classical problem of linear programming, the most famous method in this area being the simplex algorithm. In case if we only have linear equations, the Gaussian elimination is sufficient. An algebraic technique of Gröbner bases (Buchberger 1985) can be applied to tackle non-linear real equations, whereas another algebraic technique of quantifier elimination (Collins 1975) can handle arbitrary predicates definable in arithmetic.

Another useful domain is that of *Boolean values.* The techniques are various Boolean unification algorithms (see e.g. Martin, Nipkov 1990), Gröbner bases, and saturation methods.

A consistency technique for the domain of inexact arithmetical data, i.e. intervals, is tolerance propagation (Hyvönen 1992).

1.1.6 Partial Constraint Satisfaction and Approximate Constraints

Often, it is unnecessary or too costly to find out an exact solution to a CN, or exact solutions do not even exist. Then different goals might be posed:

- To satisfy as many constraints as possible.
- Find the least degree of priority such that all the constraints with higher priority can be satisfied simultaneously, and satisfy these. This assumes a priority ordering on the constraints.
- To satisfy all constraints, but up to some precision. This assumes that we have a metric for measuring errors.

Freuder and Wallace have written a paper on maximal constraint satisfaction (Freuder, Wallace 1992), containing many references. Borning and colleagues have worked a lot on the so-called constraint hierarhies and introduced several priority orderings and error metrics (see e.g. Borning et al. 1987; Borning et al. 1992).

One uses constraint networks to describe some reality. If the reality is complex, one cannot do without simplifications in specifying the structure of the network and the extensions of its constraints. If "real relations" are approximated by more restrictive ones, the worst that can happen is losing part of solutions. Thus, in cases where there is no risk that all solutions disappear, approximations may be a convenient means to make the search easier. A typical example is approximating a non-functional relation by a function.

1.1.7 Modular and Higher-Order CNs

Conventional CNs are flat, i.e. they do not have a modular structure. By this we mean that if there is some "homogeneity" in a CN with huge number of variables and constraints, we cannot take advantage of it, when writing down a description of the CN, or when reasoning about it, simply because there are no concepts in the constraint jargon for expressing this.

A way to formalize one kind of "homogeneity" are the so-called *dynamic CNs* (Guesgen, Hertzberg 1992). The term 'dynamic CNs' is quite unfortunate, since dynamicity in this context has nothing to do with time and change, and we will use '*modular CNs*' instead. The idea is as follows.

It happens often that a subset of variables of a CN participates together in several constraints, and that only certain valuations of this subset's variables can be extended to satisfy all of them. In such a case, we have to do with an implicit constraint on these variables. If this implicit constraint has a meaning on the

conceptual level, it might well be worth of having an own name. Suppose we give it a name, and we reorganize the constraints which connected our variables so that they connect the new constraint instead. Then we find ourselves in a situation, where constraints connect variables and/or constraints. The values of constraints are relations on single elements and/or tuples of elements of the universal domain. At this stage the distinction between variables and constraints becomes blurring. Now there is only one step to be taken—to abandon variables completely. Guesgen and Hertzberg do this by regarding variables as constraints connecting nothing, i.e. 0-ary constraints. (In some respect, this identification is not very neat, since the extension of a 0-ary constraint ought to be a subset of the direct product of an empty family of sets.) The networks where constraints connect constraints are called modular CNs.

In modular CNs, constraint values are relations between tuples. One could also think of giving another meaning for constraint-connecting constraints, where their values would be relations between relations. Such networks (let us call them *higher-order CNs*) ought to be a promising research direction.

In (Tyugu, Uustalu 1994), it is shown that computability statements with nested implications in structural synthesis of programs (see e.g. Mints, Tyugu 1983) can be viewed as higher-order functional constraints.

1.1.8 Programming with Constraints

1.1.8.1 The CLP Framework The most natural programming paradigm for combining with constraints is logic programming (LP).

Roughly speaking, in the conventional LP, data are ground terms of some language, and control is governed by a resolution strategy. A program's data state is the set of variable instantiations made so far (since instantiations equate variables to terms, a single instantiation is generally just a little concretization of some variable's value), and its control state is the set of atomic goals yet to be demonstrated. Instantiations happen at unification, which is part of the resolution step. The final values of variables are determined by the set of instantiations accumulated in the course of the program's run. Instantiations are equality statements and can well be viewed as constraints.

From this observation, it is not a long way to the following generalization. As computation of variable values in LP is always about solving a constraint network, although a simple one, why not liberalize the form of constraints? Let us choose an interesting domain and a set of predicates over it with fixed interpretation (e.g. real numbers and the machinery for writing down linear equations). Now, besides the usual predicates, whose meaning is defined by program clauses (we now call them *control predicates*), we have *constraint predicates* for which we allow no defining clauses, as we assume their meaning is known. We modify the concept of clause, so that a clause body now has a constraint part and a control part, which are finite sets of constraint and control atoms, respectively. We also modify the resolution rule, so that constraint atoms play no active role in resolution—they are merely accumulated, similarly to instan-

tiations. Now, likewise as it is checked at the resolution step in conventional LP whether unification succeeds (otherwise, backtrack occurs), there must be a check at resolution steps in our generalization, but a much stronger one. Namely, it must be verified that it is consistent to add to the current constraint store the constraint atoms from the input clause and the instantiations the unification suggests. This is not cheap.

The paradigm we just outlined is called *constraint logic programming (CLP)* (Jaffar, Lassez 1987; Jaffar, Lassez 1988). A CLP interpreter must have two components: an *inference engine* which deals with resolutions, and a domain-specific *constraint engine* which maintains the constraint store in a standard form, and, upon a request from the inference engine, is able to inform it whether the new constraints it suggests can be consistently added to the store.

There are a number of CLP systems around, some of them commercially available. Examples are CHIP (Dincbas et al. 1988), CLP($\mathcal{R}$) (Jaffar, Michaylov 1987), Prolog-III (Colmerauer 1990), CAL (Aiba et al. 1988), Trilogy (Voda 1988). For references and a tutorial survey on CLP, see either (Cohen 1990) or (Frühwirth et al. 1992).

In the mainstream paradigm of concurrent logic programming, originally due to Shapiro, OR-parallelism is restricted to processing of the guards of the definition clauses of a given predicate, after which a decisive commitment is made in favour of one of them (for a survey on concurrent LP, see (Shapiro 1989)). AKL (Franzén et al. 1991) is a constraint programming language stemming from this tradition that facilitates deep guards, i.e. guards involving user-defined predicates. The rôle of guards in concurrent LP languages is similar that of `ask` actions in cc languages (on these, see the next subsection).

1.1.8.2 The cc Framework Saraswat et al. have developed a *concurrent constraint programming* paradigm, called cc (Saraswat, Rinard 1990). cc languages are similar to Milner's CCS in that a program is a set of agent definitions. But the communication mechanism of cc is radically different from that of CCS: communication in cc occurs through a constraint store, which is a program's data state. The basic actions of agents are `ask`ing and `tell`ing constraints. An `ask` action succeeds if the store entails the given constraint, fails if the given constraint is inconsistent with the store, and is suspended otherwise. A `tell` action adds a constraint to the store, and succeeds if the store remains consistent, otherwise it fails. Complex behaviours are built from simpler ones by means of prefixing, indeterministic choice, interleaving, hiding, and mutual recursion. Although cc is outwardly different from committed choice concurrent LP paradigm, it adequately captures it.

The first denotational and SOS interleaving semantics of cc appeared in (Saraswat, Rinard, Panangaden 1990).

Both entailment relations of constraint systems as well as as agent definitions of programs in cc languages can be seen as production rules of graph grammars. Hence, given a program in a cc language, the information necessary to determine its semantics can be encoded in the form of a single graph gram-

mar. Montanari and Rossi have developed methods of deriving different true concurrency semantics of cc programs from their graph grammar representations. These are: partial order semantics (Montanari, Rossi 1991c; Montanari, Rossi 1993b), event structure semantics (Montanari, Rossi 1992), and contextual net semantics (Montanari, Rossi 1993a).

A logical semantics for the cc paradigm can be given using the *formulas-as-agents* and *proof-as-computation* interpretation of intuitionistic logic (Lincoln, Saraswat 1991). Indeterminacy can be properly handled by moving to the setting of linear logic (Lcc), and if one wants to allow process abstractions to be passed as messages in communications, higher-order logic is needed (HLcc) (Saraswat, Lincoln 1992).

An example of a cc language is Janus (Saraswat, Kahn, Levy 1989; Saraswat, Kahn, Levy 1990). Janus is a language for distributed programming, and enjoys the pleasant property that its computations cannot abort because of the store having become inconsistent as a result of a uncoordinated tells by several agents. This is achieved by severe syntactic restrictions on programs. A completely visual programming environment, called Pictorial Janus is under development (Kahn 1992), where exactly the same visual terms are used to depict a program, its execution states, and the whole history of these.(Saraswat 1992b) presents a thorough account of the state of the cc art.

The novel Oz (Smolka et al. 1990) language extends the cc model with object-orientation (higher-orderness), avoiding thereby the clumsiness of stream communication, which is the usual communication mechanism in the mergers of concurrent LP and object-orientation.

1.1.8.3 Constraint Imperative Programming The constraint imperative paradigm (CIP), proposed by Borning and colleagues and implemented in the object oriented languages Kaleidoscope90, '91, and '93 (Borning et al. 1992; Freeman-Benson, Borning 1992; Lopez et al. 1994), is conservative in that it seeks to keep to traditional programming idioms. In particular, it remains faithful to the conventional understanding of stores as valuations, as opposed to the store-as-constraint approach of CLP and cc. As imperative variables are subject to destructive assignments and always possess values, the task of the constraint handler of a CIP system is not to find one set of permissible valuations of a program's variables, which is typical of declarative constraint programming, but to reinstate the permissibility of the store (by changing some values), whenever an assignment to some variable happens to have spoilt this. Different constraints may have different degrees of priority.

A variable can be blocked from automatic adjustments due to violation of its constraint and from potentially triggering adjustment of the other variables of its constraint by annotating it either read-only or write-only in the statement of the constraint.

Mentally, the idiom of dynamic variables-of-state can always be replaced with that of static variables-of-history (streams). Doing so, the relation between two successive states of the store can be semantically considered as determined

by strong constraints of change and weak constraints of stay between old and new values, with old values read-only.

1.1.9 Conclusion

Despite the illusory freedom we experience from time to time, life is, in fact, pretty constrained. We have to fulfill expectations, obey regulations, stay alive... and all of that simultaneously. Luckily, in our daily doings, most of us manage to cope satisfactorily with the constraints that our wonderful world imposes on us. And in these days, we have even algorithms at our disposal to solve them "scientifically". So there is no reason for depression. What we need is a deeper insight into the nature and habits of these tiny tyrants and just some more algorithms—in order to make machines see our problems in the way we do, and have them helping us.

References

Aiba, A., Sakai, K., Sato, Y., Hawley, D. J., Hasegawa, R. 1988. Constraint logic programming language CAL. In Proc. Int'l Conf. on Fifth Generation Computer Systems, Tokyo, Japan, Dec 1988, 263–276. Tokyo: Ohmsha Publishers

Bibel, W. 1988. Constraint satisfaction from a deductive viewpoint. Artificial Intelligence **35**(3), 401–413

Borning, A., Duisberg, R., Freeman-Benson, B. N., Cramer, A., Woolf, M. 1987. Constraint hierarchies. In Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, Orlando, FL, USA, Oct 1987, 48–60. ACM

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

Buchberger, B. 1985. Gröbner bases: An algorithmic method in polynomial ideal theory. In Bose, N. K. (ed.), Multidimensional systems theory, 184–232. Dordrecht: D. Reidel

Cohen, J. 1990. Constraint logic programming languages. Communications of the ACM **33**(7), 52–68

Collins, G. E. 1975. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Proc. 2nd GI Conf. on Automata Theory and Formal Languages, 515–532. LNCS 33 Berlin: Springer-Verlag.

Colmerauer, A. 1990. An introduction to Prolog-III. Communications of the ACM **33**(7), 69–90

Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41**(3), 273–312

Dechter, R., Pearl, J. 1988. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence **34**(1), 1–38

Dechter, R., Pearl, J. 1989. Tree clustering for constraint networks. Artificial Intelligence **38**(3), 353–66

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., Berthier, F. 1988. The constraint logic programming language CHIP. In Proc. Int'l Conf. on Fifth Generation Computer Systems, Tokyo, Japan, Dec 1988, 693–702. Tokyo: Ohmsha Publishers

Franzén, T., Haridi, S., Janson, S. 1991. An overview of the Andorra Kernel Language. In Eriksson, L.-H., Hallnäs, L., Schroeder-Heister, P. (eds), Proc. 2nd Int'l Workshop on Extensions of Logic Programming, ELP'91, Stockholm, Sweden, Jan 1991, 163–179. LNAI 596 Berlin: Springer-Verlag.

Freeman-Benson, B., Borning, A. 1992. Integrating constraints with an object-oriented language. In Lehrmann Madsen, O.(ed), Proc. European Conf. on Object-Oriented Programming, ECOOP'92, Utrecht, The Netherlands, June/July 1992, 268–86. Berlin: Springer-Verlag. LNCS 615

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., Wallace, M. 1992. Constraint logic programming: An informal introduction. In Comyn, G., Fuchs, N. E., Ratcliffe, M. J. (eds), Logic Programming in Action: Proc. 2nd Int'l Logic Programming Summer School, LPSS'92, Zürich, Switzerland, Sept 1992, 3–35. Berlin: Springer-Verlag. LNAI 636

Freuder, E. C., Wallace, R. J. 1992. Partial constraint satisfaction. Artificial Intelligence bf 58(1–3), 21–70

Guesgen, H. W., Hertzberg, J. 1992. A Perspective of Constraint-Based Reasoning: An Introductory Tutorial. Berlin: Springer-Verlag. LNAI 597

Hyvönen, E. 1992. Constraint reasoning based on interval arithmetic: The tolerance propagation approach. Artificial Intelligence **58**(1–3), 71–112

Jaffar, J., Lassez, J.-L. 1987. Constraint logic programming. In Conf. Record 14th Annual ACM Symp. on Principles of Programming Languages, Munich, West Germany, Jan 1987, 111–119. ACM SIGACT/SIGPLAN

Jaffar, J., Lassez, J.-L. 1988. From unification to constraints. In Furukawa, K., Tanaka, H., Fujisaki, T. (eds), Logic Programming '87: Proc. 6th (Japanese) Conf. Tokyo, Japan, June 1987, 1–18. Berlin: Springer-Verlag. LNCS 315

Jaffar, J., Michaylov, S. 1987. Methodology and implementation of a constraint logic programming system. In Proc. 4th Int'l Conf. on Logic Programming, Melbourne, Australia, 1987, 196–218. The MIT Press

Kahn, K. M. 1992. Concurrent constraint programs to parse and animate pictures of concurrent constraint programs. In Proc. Int'l Conf. on Fifth Generation Computer Systems, Tokyo, Japan, June 1992. ICOT: Tokyo

Latombe, J.-C. (ed) 1978. Proc. IFIP Workshop on Artificial Intelligence and Pattern Recognition in CAD. Amsterdam: North-Holland

Lauriere, J.-L. 1978. A language and a program for stating and solving combinatorial problems. Artificial Intelligence **10**(1), 29–127

Lincoln, P., Saraswat, V. A. 1991. Proofs as concurrent processes: A logical interpretation for concurrent constraint programming. Technical report, Systems Sciences Laboratory, Xerox PARC, Palo Alto, CA

Lopez, G., Freeman-Benson, B. N., Borning, A. 1994. Kaleidoscope: A constraint imperative programming language. In this volume

Mackworth, A. K. 1977. Consistency in networks of relations. Artificial Intelligence **8**(1), 99–118

Mackworth, A. K. 1992. The logic of constraint satisfaction. Artificial Intelligence **58**(1–3), 3–20

Mackworth, A. K., Freuder, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problem. Artificial Intelligence **25**(1), 65–74

Martin, U., Nipkov, T. 1990. Boolean unification: The story so far. In Kirchner, C. (ed), Unification. Academic Press

Mints, G., Tyugu, E. 1983. Justification of the structural synthesis of programs. Science of Computer Programming **2**, 215–240

Mohr, R., Henderson, T. C. 1986. Arc and path consistency revisited. Artificial Intelligence **28**(2), 225–233

Montanari, U. 1974. Networks of constraints: Fundamental properties and application to picture processing. Information Sciences **7**(2), 95–132

Montanari, U., Rossi, F. 1991a. Constraint relaxation may be perfect. Artificial Intelligence **48**(2), 143–170

Montanari, U., Rossi, F. 1991b. Perfect relaxation in constraint logic programming. In Furukawa, K. (ed.), Proc. 8th Int'l Conf. on Logic Programming, Paris, France, June 1991, 223–237. Cambridge, MA: The MIT Press

Montanari, U., Rossi, F. 1991c. True concurrency in concurrent constraint logic programming. In Saraswat, V., Ueda, K. (eds), Proc. 1991 Symp. on Logic Programming, 694–713

Montanari, U., Rossi, F. 1992. An event structure semantics for concurrent constraint programming. Submitted for publication

Montanari, U., Rossi, F. 1993a. Contextual occurrence nets and concurrent constraint programming. In Proc. Dagstuhl Seminar on Graph Transformations in Computer Science, Jan 1993, Berlin: Springer-Verlag. LNCS

Montanari, U., Rossi, F. 1993b. Graph rewriting for a partial ordering semantics of concurrent constraint programming. Theoretical Computer Science **109**, 225–56

Panangaden, P., Saraswat, V. A., Scott, P. J., Seely, R. A. G. 1991. What is a constraint system? Technical report, Xerox Parc, Palo Alto, CA

Saraswat, V. A. 1992a. The category of constraint systems is cartesian-closed. In Proc. 7th Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, CA, USA, June 1992, 341–345. Los Alamitos, CA: IEEE Comp. Soc. Press

Saraswat, V. A. 1992b. Concurrent constraint programming: A survey. Technical report, Xerox PARC, Palo Alto, CA

Saraswat, V. A., Kahn, K. M., Levy, J. 1989. Programming in Janus. Technical report, Xerox PARC, Palo Alto, CA

Saraswat, V. A., Kahn, K. M., Levy, J. 1990. Janus: A step towards distributed constraint programming. In Proc. North American Conf. on Logic Programming, Austin, TX, USA, Oct 1990.

Saraswat, V. A., Lincoln, P. 1992. Higher-order, linear concurrent constraint programming. Technical report, Xerox PARC, Palo Alto, CA

Saraswat, V. A., Rinard, M. 1990. Concurrent constraint programming. In Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, San Fransisco, CA, USA, Jan 1990, 232–245. ACM SIGPLAN/SIGACT

Saraswat, V. A., Rinard, M., Panangaden, P. 1990. Semantic foundations of concurrent constraint programming. In Conf. Record 18th Annual ACM Symp. on Principles of Programming Languages, Orlando, FL, USA, 1991, 333–352. ACM SIGPLAN/SIGACT

Shapiro, E. 1989. The family of concurrent logic programming languages. ACM Computing Surveys **21**(3), 413–510

Smolka, G., Henz, M., Würtz, J. 1993. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, DFKI, Saarbrücken

Sussman, G. J., Steele Jr. G. L. 1980. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. Artificial Intelligence bf 14(1), 1–39

Sutherland, I. E. 1963. Sketchpad: A man-machine graphical communication system. In Proc. AFIPS Spring Joint Computer Conf., Detroit, MI, USA, 1963, 329–346

Tyugu, E. 1970. Solving problems on computational models. J. Computational Mathematics and Math. Phys. **10**, 716–33

Tyugu, E., Uustalu, T. 1994. Higher-order functional constraint networks. In this volume

Van Hentenryck, P., Deville, Y., Teng, C.-M. 1992. A generic arc-consistency algorithm and its specializations. Artificial Intelligence **57**(2–3), 291–321

Voda, P. 1988. The constraint language Trilogy: Semantics and computations. Technical report, Complete Logic Systems, North Vancouver, BC

Waltz, D. L. 1972. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, MIT