Solving Mixed and Conditional Constraint Satisfaction Problems

Article *in* Constraints · April 2003

DOI: 10.1023/A:1022394531132

2 authors:

Esther Myriam Gelle
Helbling Technik AG
39 PUBLICATIONS **435** CITATIONS
SEE PROFILE

Boi Faltings
École Polytechnique Fédérale de Lausanne
441 PUBLICATIONS **9,514** CITATIONS
SEE PROFILE

Constraints, 8, 107–141, 2003

Solving Mixed and Conditional Constraint Satisfaction Problems

ESTHER GELLE esther.gelle@ch.abb.com
Information Technologies Department, ABB Corporate Research Ltd., CH-5405 Baden-Daettwil, Switzerland

BOI FALTINGS boi.faltings@epfl.ch
Artificial Intelligence Laboratory (LIA), Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne, Switzerland

Abstract. Constraints are a powerful general paradigm for representing knowledge in intelligent systems. The standard constraint satisfaction paradigm involves variables over a discrete value domain and constraints which restrict the solutions to allowed value combinations. This standard paradigm is inapplicable to problems which are either:

(a) mixed, involving both numeric and discrete variables, or

(b) conditional,[1] containing variables whose existence depends on the values chosen for other variables, or

(c) both, conditional and mixed.

We present a general formalism which handles both exceptions in an integral search framework. We solve conditional problems by analyzing dependencies between constraints that enable us to directly compute all possible configurations of the CSP rather than discovering them during search. For mixed problems, we present an enumeration scheme that integrates numeric variables with discrete ones in a single search process. Both techniques take advantage of enhanced propagation rule for numeric variables that results in tighter labelings than the algorithms commonly used. From real world examples in configuration and design, we identify several types of mixed constraints, i.e. constraints defined over numeric and discrete variables, and propose new propagation rules in order to take advantage of these constraints during problem solving.

Keywords: (Conditional) constraint satisfaction, local consistency, numeric constraints, mixed constraints, refine operator, propagation rule

1. Introduction

Constraint satisfaction is a general framework for solving NP-hard combinatorial problems. It has been shown to be generally applicable to a wide variety of problems in areas as diverse as computer vision, scheduling and planning, product configuration, design, and diagnosis. A constraint satisfaction problem (CSP) consists of a set of *variables*, the values of which are restricted by *constraints*.

A good example is product configuration, where individual components are chosen from a component catalogue and have to be arranged in a product structure, so as to

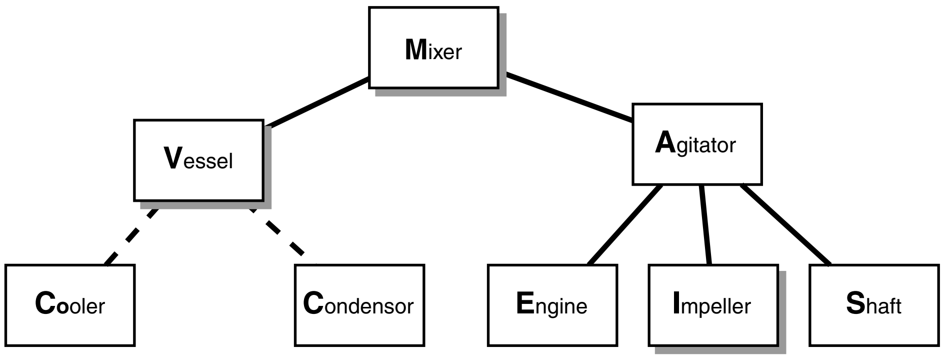

Figure 1. Product structure of a mixer. It shows the decomposition of a mixer in sub-components. Solid lines define required components, whereas dashed lines define optional ones, e.g. condenser and cooler.

satisfy manufacturing constraints and customer requirements. Consider for example the task of configuring an industrial mixer as presented in [43]. The product structure of a mixer is presented in Figure 1. An industrial mixer consists of a vessel and an agitator. The agitator is itself decomposed into engine, impeller, and shaft. Optionally, the vessel can include a cooler or a condenser. The components' characteristics are represented by properties which take values from a given domain. Typical customer requirements define the context in which the mixer is used, e.g. a reactor for chemical experiments, a storage tank, etc., or the chemical properties of the product to be mixed. In this example, we refer to the context as mixing task. The result of the configuration task is a completely specified product, in which all necessary components have been chosen and their properties, such as the vessel volume or the slurry pressure, have been set. The task of configuring an industrial mixer can be conveniently represented as CSP:

- Components and component properties are the variables of the CSP. We denote the components by their initial, and use a dot notation to refer to properties of components. The component set is thus $\{M, MT, V, Co, C, A, S, E, I\}$ and the properties are $MT.slurrypressure$, $V.volume$, etc.
- Constraints define relations over the variables, for example $C(M, V.volume) := \{(reactor\ [0, 100])(storagetank\ [0, 1000])(mixer\ [0, 1000])\}$ relating the mixer type to the volume of the mixer vessel or $I.power = MT.slurrydensity * I.rps^3 * I.pos * I.diameter^5$ defining the impeller power in function of the slurry density of the mixing task, the rounds per second, the position, and the diameter of the impeller (see Table 1).

Table 1. Some constraints appearing in the mixer example. $C(I, I.ratio)$ is a mixed constraint since I is a discrete and $I.ratio$ a numerical variable according to the definition given in Section 2

Numeric constraints

$1/2 * V.diameter \leq V.height \leq 2 * V.diameter$
$I.diameter = I.ratio * V.diameter$
$I.rps \leq V.diameter/I.diameter$
$I.power = MT.slurrydensity * I.rps^3 * I.pos * I.diameter^5$
$E.power \geq 2 * I.power$

Discrete constraints

C(MT, MT.slurryviscosity, V.volume, I.entry) :=
{(*blending low small notcentered*)(*blending low large side*)
otherwise I.entry = *top*}

C(MT.heattransfer, M) :=
{(*true reactor*)(*false storagetank*)(*false mixer*)}

C(MT, C) :=
{(*dispersion condenser*)}

Mixed constraints

C(M, V.volume) :=
{(*reactor* [0, 100])(*storagetank* [0, 1000])(*mixer* [0, 1000])}

C(I, I.ratio) :=
{(*axialturbine* 0.399414)(*denteddisk* 0.5)(*anchorstirrer* 0.949219)
(*hellicalribbon* 0.949219) *otherwise I.ratio* = 0.329102}

C(I, I.position) :=
{(*radialturbine* 5) (*axialturbine* 1.5)(*propeller* 0.35)
otherwise I.position = [0.15, 0.2]}

C(V, V.volume, V.diameter, V.height) :=
{(*hemispherical* $volume = 1/12 * \pi * diameter^3 +$
$1/4 * \pi * diameter^2 * (height - 1/2 * diameter)$)
(*cylindrical* $volume = 1/4 * \pi * diameter^2 * height$)}

Activity constraints

$Mixer = reactor \overset{ACT}{\rightarrow} Cooler$
$Vessel.volume \geq 150 \overset{ACT}{\rightarrow} Condenser$

The mixer example exhibits several characteristics typical of configuration and also design tasks:

1. The problem formulation contains discrete AND numeric variables, i.e. the problem is *mixed*. Similarly, constraints can be defined over both types of variables leading to *mixed constraints*.

2. Some variables exist only under certain conditions, i.e. the existence of variables in a solution is *conditional*. This is the case for *condenser* and *cooler* in the mixer example.

3. Numeric constraints, especially inequalities, define entire sets of values as allowed, rather than single values. A constraint problem defined by numeric constraints often results in sets of contiguous solutions, called *solution sub-spaces*. *Local consistency* methods provide a good approximation of such spaces, using a reasonable amount of computing power.

According to these characteristics, we call such a problem *mixed conditional constraint satisfaction problem (mixed CondCSP).*

This paper describes a new two-phase resolution method for mixed CondCSPs, which solves a problem exhibiting all of the above characteristics in an integral search framework. The proposed algorithm first analyzes the variable dependencies and compiles them into a tree of static constraint problems. Local consistency is applied to prune away inconsistent branches during the generation of the tree. As a result, the remaining static CSPs are locally consistent. Each CSP is then solved according to a depth-first search on midpoint values of variable labels. Local consistency is again applied propagating the current instantiations to mid-point values to further narrow down variable labels. And finally, since local consistency plays a key role when solving mixed CondCSPs, we propose new propagation rules that handle numeric and mixed constraints.

2. Conditional Constraint Satisfaction

As demonstrated in the mixer example, real world constraint satisfaction problems involve discrete and numeric variables. Constraints are either defined over a single type of variable or they involve both. We therefore distinguish between discrete, numeric, and mixed constraints:

- A *discrete* constraint $C^{Dis}_{X_1,\dots,X_k}$ defines a set of allowed value tuples $(x_{1_j},\dots,x_{k_j})$ either extensionally or by a predicate such that $(x_{1_j},\dots,x_{k_j}) \in D_1 \times \cdots \times D_k$.
- A *numeric* constraint $C^{Num}_{X_1,\dots,X_k}$ is a relation $E \circledcirc 0$ where $\circledcirc \in \{=, \geq, \leq, <, >\}$ and E is an expression built from constants, variables, and unary or binary operations $\{+, -, /, *, exp, sqrt, \dots\}$ over the reals.
- A *mixed* constraint $C^{Mix}_{X_1,\dots,X_m}$ is a relation defined over the discrete variables $X_1,\dots,X_{k-1}$ and the numeric variables $X_k,\dots,X_m$.

Some of the constraints in the mixer configuration problem are presented in Table 1. Discrete constraints such as $C(MT.heattransfer, M)$ are represented extensionally. Numeric constraints are given as formulas. Mixed constraints are defined either extensionally, by listing admissible value combinations such as $C(M, V.volume)$, or intentionally by a formula such as $C(V, V.volume, V.diameter, V.height)$ in.

Another typical form of knowledge appearing in configuration tasks are constraints that define conditions under which a variable has to exist. Such *activity constraints* express that a set of components requires another component in a specific context, in order to function appropriately. The constraint $Mixer = reactor \overset{ACT}{\rightarrow} Cooler$ represents the fact that a mixer of type reactor used in chemical reactions requires a cooler. $Vessel.volume \geq 150 \overset{ACT}{\rightarrow} Condenser$ is another example, which specifies that a condenser is required if the vessel provides volume for large quantities to be processed (Table 1).

2.1. Some Definitions

We introduce the model of *Conditional Constraint Satisfaction* (CondCSP) in order to adapt standard constraint satisfaction problems to encode solutions with a varying number of variable instances. A variable of a CondCSP can be active or not. A variable is assigned one of its possible values, if and only if, it is active.

Definition 1. [Active variable] Let $\mathcal{V}$ be the set of all variables occurring in the constraint problem. Then, for all $X_i \in \mathcal{V}$ $active(X_i) \leftrightarrow X_i = x$ holds with $x \in D_i$.

Constraints restricting value combinations of variables are called *compatibility constraints* in this model. Such compatibility constraints can be discrete, numeric, or mixed as discussed earlier. In contrast to a static CSP, a compatibility constraint in a CondCSP is only *relevant* to a problem if all its variables are active. It follows that a compatibility constraint is trivially satisfied if at least one of its variables is not active. This can be made explicit by considering a compatibility constraint only in those parts of the search space in which all variables of the constraint are active.

Definition 2. [Relevant compatibility constraint] For any $C_{X_1,\dots,X_j}$: $active(X_1) \wedge \cdots \wedge active(X_j) \leftrightarrow relevant(C_{X_1,\dots,X_j})$.

Additionally, a CondCSP can post constraints on a variable's activity in a given context of value assignments.

Definition 3. [Activity constraint] An activity constraint[2] $C_{X_1,\dots,X_j} \overset{ACT}{\rightarrow} X_k$ is defined by the logical implication $C_{X_1,\dots,X_j} \rightarrow active(X_k)$ where $C_{X_1,\dots,X_j}$ is a single constraint, which expresses an *activation condition* under which the variable X_k becomes active. An activity constraint is *satisfied* by an assignment of values to variables $\{X_1 = x_1, \dots, X_j = x_j\}$ if NOT$(C_{X_1,\dots,X_j}(x_1, \dots, x_j))$ OR $active(X_k)$ is true.

In our model, we allow the formulation of activation conditions as numeric constraint, or a set of value assignments (a discrete constraint), instead of a single value assignment. Similar to a compatibility constraint, we say that an activation condition is *relevant* if all the variables on which the condition is defined are active. Finally, a set of initial variables $\mathcal{V}_I$ is defined that are always active, and thus, present in every solution.

Definition 4. [CondCSP] A *Conditional constraint satisfaction problem* $\mathcal{P} = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, \mathcal{V}_I \rangle$ is defined by:

- A set of variables $\mathcal{V}$ representing all variables that may potentially become active, and be part of a solution. Each variable $X_i \in \mathcal{V}$ has associated a domain $D_i \in \mathcal{D}$ representing the set of possible values for the variable.
- A nonempty set of *initial variables* $V_I \subseteq \mathcal{V}$. These variables have to be part of every solution.
- A set of *compatibility constraints* $\mathcal{C}^C \subseteq \mathcal{C}$ on subsets of $\mathcal{V}$ representing allowed value combinations for these variables.
- A set of *activity constraints* $\mathcal{C}^A \subseteq \mathcal{C}$ on subsets of $\mathcal{V}$ specifying constraints between the activity of a variable, and possible values of problem variables.

A solution S to a CondCSP is defined as:

i. an assignment of values to a set of variables $\subseteq \mathcal{V}$, such that S satisfies all constraints in $\mathcal{C}^{\mathcal{C}} \cup \mathcal{C}^{\mathcal{A}}$;

ii. such that all variables of V_I are assigned in S.

A *minimal solution M* to a CondCSP is a solution to the CondCSP which is subset minimal, i.e. there is no solution S' satisfying all constraints such that $S' \subset S$.

Minimal solutions of a CondCSP only consider those assignments for which there exist no other assignment that has less identical variable-value pairs satisfying all constraints. The definition of CondCSPs is similar to dynamic CSPs given in [27], with the difference that CondCSPs may be defined over numeric and discrete variables, and consequently, the conditions in activity constraints can be discrete or numeric constraints.

Example. Consider the following CondCSP $\mathcal{P}_{Small}$ given by

- the variables $X_1 \in \{a, b\}$, $X_2 \in \{c, d\}$, $X_3 \in \{e, f\}$, $X_4 \in \{g, h\}$ of which X_1 and X_2 are initially active,
- the compatibility constraints $C_{X_1X_2} = \{(a, d), (b, c)\}$ and $C_{X_2X_3X_4} = \{(c, e, h), (c, f, g), (d, f, h)\}$,
- the activity constraints $X_1 = b \overset{ACT}{\rightarrow} X_3, X_3 = e \overset{ACT}{\rightarrow} X_4, X_2 = c \overset{ACT}{\rightarrow} X_4, X_4 = g \overset{ACT}{\rightarrow} X_3$

Out of $2^4 = 16$ possible value assignments, only 3 are minimal solutions to this problem: $\{X_1 = a, X_2 = d\}$, $\{X_1 = b, X_2 = c, X_3 = e, X_4 = h\}$, $\{X_1 = b, X_2 = c, X_3 = f, X_4 = g\}$. Although the value assignment $\{X_1 = a, X_2 = d, X_3 = f, X_4 = h\}$ also satisfies all constraints, it is not a minimal solution, since $\{X_1 = a, X_2 = d\}$ is a subset of it. ■

The CondCSP model allows explicit reasoning on the existence of variables and integrates neatly constraint processing and variable creation. In the original solution method for CondCSPs given in [27], a new variable is introduced by an activity constraint, only if the instantiated variables satisfy the condition, i.e. if a well-founded partial solution allows for the introduction of the new variable. The algorithm given in their paper is based on value enumeration, and can be roughly described as follows:

1. All activity constraints are checked for a condition satisfied by the set of already assigned variables. If such an activity constraint is found, the variable to be activated is added to the set of active variables.
2. Each relevant compatibility constraint containing the latest variable activated is checked against the already assigned variables.
3. A not yet assigned variable is chosen from the set of active variables and assigned a value from its domain, which has not yet been tried.
4. If an inconsistency is detected during a constraint check, the algorithm first backtracks to the next value of the latest variable that had a value assigned. If a value that led to the activation of a new variable becomes inconsistent, the previously activated variable has to be removed from the set of active variables. This is implemented by an ATMS recording the value assignments that lead to a variable activation as justifications. ATMS-caching "remembers" the conditions under which an activity constraint has to be applied. It also prevents cycling, because justification labels derived from the assigned variables in the condition are kept minimal by the ATMS.

Example. The following trace is generated when applying the resolution method described in [27] to $\mathcal{P}_{Small}$, assuming that the variable values are instantiated in the order $X_1, \ldots, X_4$:

1. X_1 is assigned the value a.
2. X_2 is assigned the value c, $X_2 = c \overset{ACT}{\rightarrow} X_4$ activates the variable X_4, the activity of X_4 is justified by $X_2 = c$.
3. $X_1 = a, X_2 = c$ fails on constraint $C_{X_1X_2}$, backtrack.
4. X_2 is assigned the value d, X_4 is deactivated since it is no longer justified.
5. $\{X_1 = a, X_2 = d\}$ is a solution, backtrack to find further solutions ... ■

The weakness of this algorithm is that the context, which leads to the introduction of a new variable, consists of particular value assignments. It does not handle well the following situations appearing in mixed CondCSPs:

1. the activation condition is a discrete constraint with several value combinations,
2. the activation condition is a numeric constraint defining infinitely many value combinations. In the mixer example, the existence of a condenser depends on the value of the vessel volume chosen from a continuous range, for example 150..1000. This means that sets of values can be chosen so as to satisfy, or not, the condition.

Activating variables based on partial solutions, is thus, not an efficient way to solve a mixed CondCSP. We propose a new algorithm which generates standard CSPs, by considering combinations of constraint conditions. Existing algorithms can then be applied to solve each of the standard CSPs.

2.2. Reduction of a CondCSP to a Set of Standard CSPs

The algorithm presented in [27] loops through all activity constraints, applying the first constraint whose condition is relevant to the current context of value assignments. A more efficient way of solving a CondCSPs takes into account the order in which variables are activated by activity constraints. This order is identified by analyzing the context of active variables in which the condition of an activity constraint becomes relevant, i.e. the variables it is defined on are active (Definition 3). Given, for example, the activity constraint $X_1 = b \overset{ACT}{\rightarrow} X_3$, X_3 will be activated if X_1 is active and assigned the value b. Activity constraints may also introduce variables that are used in the condition of a second activity constraint, e.g. $X_3 = e \overset{ACT}{\rightarrow} X_4$ cannot be applied before $X_1 = b \overset{ACT}{\rightarrow} X_3$, since the variable X_3 activated by the latter, is used in the condition of the former.

Analyzing the dependencies between activity constraints allows us to identify an order in which they can be applied, eliminate those variables which will never be activated (reachability check), and detect cycles in the definition of the activity constraints (cycle check). The dependency between activity constraints can be represented in a directed graph, in which each activity constraint represents a node, and there is a directed edge between two nodes a and b, if the activity constraint of a activates a variable that is used in the activation condition of b.

The main steps to be carried out in order to generate the problem spaces defined implicitly in a CondCSPs are the following:

1. Create a single directed graph from the set of activity constraints representing the dependencies between the constraints, starting from the root node defined by the set of initially active variables.
2. Eliminate cycles in the original graph by clustering all nodes in such cycles into a super-node.
3. Derive one total order (there might be several) from the partial order given by the resulting directed acyclic graph (DAG), traverse the graph in this order, and apply the activity constraints.
4. When an activity constraint is applied in the context of already activated variables, the minimal solutions are found in two subspaces: 1) the problem space, in which the new variable is active, and the activation condition has to be satisfied, or 2) the problem space, in which the condition must not be satisfied. As a consequence, the minimal solutions to a set of activity constraints are found in the combination of the individual subspaces.

The CondCSP is reduced to a set of standard CSPs, the problem spaces, each defined on a different set of variables, and depending on them, a different set of compatibility constraints.

Example. The first step only considers activity constraints that are connected to the root node by at least one path. In Figure 2(a), the graph of $\mathcal{P}_{small}$ is shown. A set of activity constraints can form a cycle in this graph, e.g. they are in a cyclic dependency.

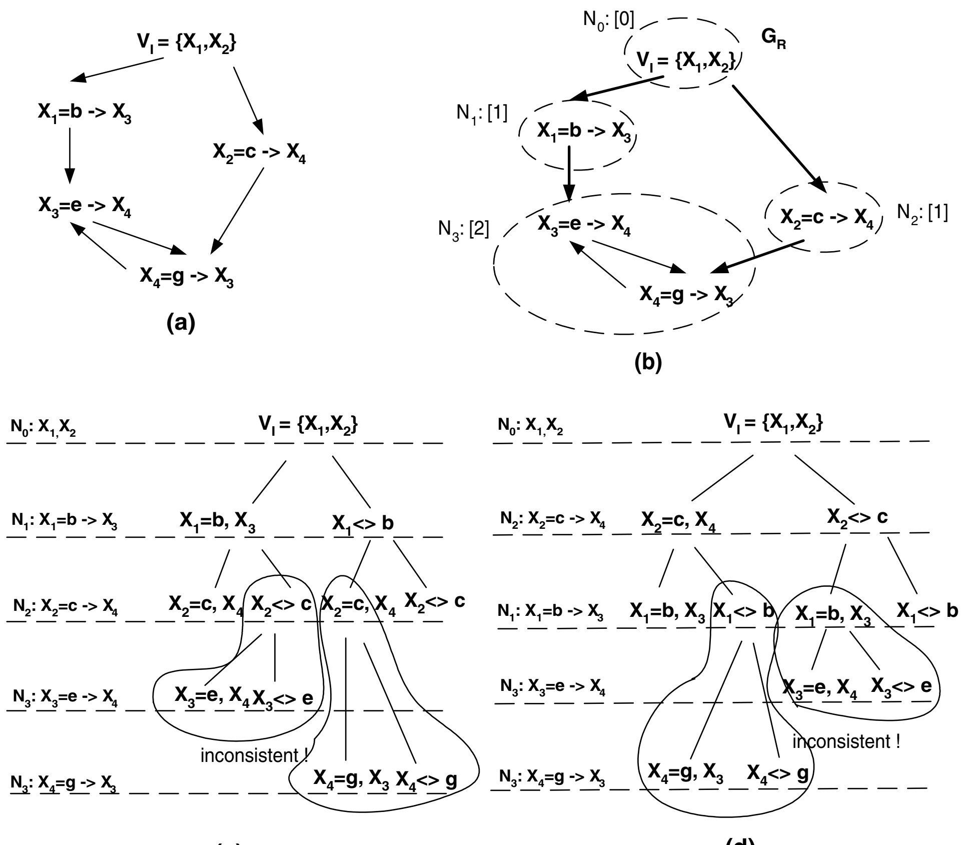

Figure 2. (a) Set of activity constraints with the dependencies between activity constraints shown in a directed graph, (b) the reduced graph of the same set of activity constraints with super-node N_3, the distance to the root node of each node in the graph is shown in brackets, (c) search tree showing a possible order in which the activity constraints are applied, (d) another ordering of activity constraints. The encircled branches are inconsistent, due to the compatibility constraint between X_1 and X_2.

An example of a cycle is the set of constraints $X_3 = e \overset{ACT}{\rightarrow} X_4$, and $X_4 = g \overset{ACT}{\rightarrow} X_3$. If X_3 is active and has the value e, X_4 will be activated. The second constraint $X_4 = g \overset{ACT}{\rightarrow} X_3$ is *redundant*, since it does not add any new information (X_3 is already active). Similarly, $X_3 = e \overset{ACT}{\rightarrow} X_4$ is redundant if X_4 is already active. By clustering nodes taking part in a cycle into super-nodes, the original graph is reduced into an acyclic one (Figure 2(b)). The third step defines one, or several, orders in which activity constraints can be applied. In Figure 2(c) and (d), either N_1 or N_2 can be chosen as next node after defining the set of initially active variables in N_0. This implies that the activity constraints can be applied in two different orders: N_0, N_1, N_2, N_3 shown in (c) and N_0, N_2, N_1, N_3 shown in (d). In the search trees of (c) and (d), only the constraints and variables newly added by applying the activity constraint are shown at a given level. Both orders produce the same solutions. Applying local consistency methods at each step in the search tree leads to a considerable pruning in the example of Figure 2(c) and (d). Due to the compatibility constraint $C_{X_1X_2}$, two subspaces can be removed after applying the second activity constraint. Only the problem spaces $X_1 = a, X_2 = d$, and $X_1 = b, X_2 = c, X_3, X_4$ contain minimal solutions. ■

2.2.1 *Creating a Dependency Graph from the Activity Constraints*

To determine the order in which activity constraints should be applied, we introduce the dependency relation *DR* between two activity constraints.

Definition 5. [Dependency relation between activity constraints] An activity constraint $C_n \overset{ACT}{\rightarrow} X_n$ is *dependent* on a second activity constraint $C_1 \overset{ACT}{\rightarrow} X_1$, denoted by $C_n \overset{ACT}{\rightarrow} X_n\ DR\ C_1 \overset{ACT}{\rightarrow} X_1$, if there exists a chain of activity constraints $C_i \overset{ACT}{\rightarrow} X_i, i = 2, \ldots, n-1$ such that $X_i \in Vars(C_{i+1}), i = 1, \ldots, n-1$. A special case occurs for $n = 1$.

If $X_1 \in Vars(C_n)$, the constraint $C_n \overset{ACT}{\rightarrow} X_n$ is *directly dependent* on $C_1 \overset{ACT}{\rightarrow} X_1$. If, additionally, $C_1 \overset{ACT}{\rightarrow} X_1$ is dependent on $C_n \overset{ACT}{\rightarrow} X_n$, there is a cycle in the chain of dependencies, and we speak of *cyclic dependency*. The dependency between activity constraints can be represented by a directed graph $\mathcal{G} = \langle \mathcal{X}, \mathcal{U} \rangle$ [19] where:

- The set of nodes $\mathcal{X}$ is the set of activity constraints. The set of initially active variables, V_I form the root node, since they are special activity constraints: $X \in V_I$ is an activity constraint, the condition of which always holds, e.g. $true \overset{ACT}{\rightarrow} X$.
- The edges $\mathcal{U}$ are defined as follows: there is a directed edge from one node to another, if the condition of the second node contains a variable that is activated by the first. The arrow always points to the dependent node. Transitive edges are not represented in the graph.

A directed graph containing cycles is made acyclic by collapsing the nodes corresponding to activity constraints in a cycle into a super-node. Since there may be exponentially many cycles, it is indicative to identify sets of *strongly connected* nodes. A set of nodes of a graph are strongly connected, if and only if, there exists a (directed) path between each pair of nodes, i.e. all the directed edges linking the set of strongly connected nodes participate in at least one cycle. A simple depth-first traversal of the nodes in a directed graph allows us to identify all strongly connected components in a graph in $\mathcal{O}(max(|\mathcal{X}|, |\mathcal{U}|))$ [19]. From these strongly connected components, a reduced graph

$\mathcal{G}_R$ is built as follows:

- Each strongly connected component in $\mathcal{G}$ is a node in $\mathcal{G}_R$.
- A directed edge exists between two nodes in $\mathcal{G}_R$, if there is at least one directed edge between a node of the first strongly connected component, to a node of the second strongly connected component in the original graph $\mathcal{G}$.

Example. The nodes in Figure 2(b) are N_0 consisting of the initial variables, N_1 formed by the activity constraint $X_1 = b \overset{ACT}{\rightarrow} X_3$, N_2 formed by $X_2 = c \overset{ACT}{\rightarrow} X_4$, and the super-node N_3 containing the cycle of activity constraints $X_3 = e \overset{ACT}{\rightarrow} X_4$, and $X_4 = g \overset{ACT}{\rightarrow} X_3$. ■

Lemma 1 *The reduced graph $\mathcal{G}_R$ is a DAG.*

Proof: True by construction. ■

The directed edges in this reduced graph represent the DR relationship (without the transitive arcs).

2.2.2 Ordering Activity Constraints

We know from graph theory, that for a directed acyclic graph (DAG) the property of a strict partial order holds [19].

Lemma 2 *Given a set of activity constraints in which no cyclic dependency occurs. Then, the dependency relation establishes a strict partial order $\prec$ on the activity constraints.*

Proof: Since no cyclic dependency occurs, the dependency relation can be stated as a DAG. For a DAG, the property of strict partial order trivially holds. ■

Activity constraints in the DAG can be ordered by comparing the paths from the root node to a given node. In Figure 2(b), node N_1 will be applied before N_3, whereas N_1 and N_2 can both be applied after executing N_0. In general, more than one possibility to reach a node in the reduced graph may exist. As shown in Figure 3, an activation condition can be

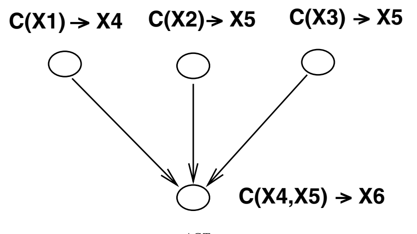

Figure 3. To establish a context in which $C(X_4, X_5) \overset{ACT}{\rightarrow} X_6$ can activate the variable X_6, either the combination of $C(X_1)$, $C(X_2)$ or $C(X_1)$, $C(X_3)$ has to be satisfied. In order to establish the minimal context, one would have to compute the combinations in a AND-OR tree. Instead, we compute a safe approximation by considering all entering edges related by AND, and computing the longest path.

defined on more than one variable, each variable activated by a different constraint. This implies that several edges may enter a node in the reduced graph, some of them activating a different, and others the same, variable in the activation condition. Since all variables in the current condition have to be active, the conditions of the predecessors activating the different variables have to be satisfied simultaneously, whereas it is sufficient to consider one of the conditions activating the same variable. The minimal context in which the variables of the current condition are activated, is therefore, defined by an AND-OR relation over the conditions of the predecessor constraints. Computing this minimal context for each activity constraint would be computationally expensive. We therefore, compute a safe approximation by assuming that all predecessor conditions have to be verified, thus combining them by an AND relation. We compute the length of the path from the root node to each node in the reduced graph with AND relations between edges entering the same node. This gives us a means to compare and order the nodes. To do so, each edge in the reduced graph is labeled with unit length 2(b). The distance of the current node to the root node is influenced by the distance of the last predecessor activated. In other words, the longest path from the root node to the current node gives a safe approximation for when the current node can be applied. The shortest path algorithm [19] is adapted to compute the longest path from the root node to each node by labeling the distance between two nodes -1 instead of 1. The distance computed for every node in the graph produces an ordering of the nodes in the increasing distance from the root node. Nodes with the same distance can be treated in arbitrary order, they are called *incomparable* (nodes N_1 and N_2 in Figure 2(b)).

2.2.3 Applying Activity Constraints

The explicit generation of problem spaces is conducted by traversing the reduced graph in a given order, applying the activity constraint at each node. Two cases have to be treated:

1. A node in $\mathscr{G}_R$ corresponds to a single activity constraint. The activity constraint is applied to the already existing problem spaces.
2. The node is a super-node. No ordering between the activity constraints can be established.

Case 1 According to the definition of an activity constraint, either the activation condition is relevant, i.e. its variables are active (Definition 3), and the new variable is added to the current problem space, or not. The addition of one activity constraint to a problem space leads to a disjunction, which can be translated into at most, two different problem spaces. In order to explicit these problem spaces, we require the definition of the complement $\overline{C}$ of a compatibility constraint C, as the set of value combinations not allowed by C. The complement exists under the assumption of a closed world, i.e. those tuples that are not given in the constraint are disallowed. For a discrete constraint, it is the set of disallowed tuples, and for a numeric constraint, it is the complement of the solution set defined by the constraint C. We restrict the acceptable numeric conditions to the form of inequality constraints, since the complement of an equality constraint would require the addition of two strict inequalities.

Let P be the current problem space, which consists of a set of active variables $\mathcal{V}_P$, a set of relevant constraints $\mathcal{C}_P$, and $C \overset{ACT}{\rightarrow} X$, an activity constraint that is added to P.[2] Under the assumption that all variables in C are active; i.e. C is relevant, the activity constraint $C \overset{ACT}{\rightarrow} X$ splits P into three parts:

- a problem space P_1 with C, and variable X such that

$$\mathcal{V}_{P_1} = \mathcal{V}_P \cup \{X\},$$
$$\mathcal{C}_{P_1} = \mathcal{C}_P \cup \{C\},$$

- a problem space P_2 with $\overline{C}$, and X such that

$$\mathcal{V}_{P_2} = \mathcal{V}_P \cup \{X\},$$
$$\mathcal{C}_{P_2} = \mathcal{C}_P \cup \{\overline{C}\},$$

- a problem space P_3, only with $\overline{C}$ such that

$$\mathcal{V}_{P_3} = \mathcal{V}_P,$$
$$\mathcal{C}_{P_3} = \mathcal{C}_P \cup \{\overline{C}\}.$$

We can prove the following lemma:

Lemma 3 *Given a problem space P and an activity constraint* $ac : C \overset{ACT}{\rightarrow} X$ *such that* $Vars(C) \subseteq \mathcal{V}_P$. *Only the subspaces* P_1, P_3 *of the CondCSP*: $P \wedge ac$ *contain minimal solutions.*

Proof: Let $C \overset{ACT}{\rightarrow} X$ be an activity constraint. Since all variables of C are active in P, the three sub-spaces P_1, P_2, P_3 contain the solutions of $P \wedge C \overset{ACT}{\rightarrow} X$. The solutions of P_3 are subsets of solutions in P_2 because $\mathcal{V}_{P_3} \subseteq \mathcal{V}_{P_2}$, and $\mathcal{C}_{P_3} = \mathcal{C}_{P_2}$. No solution s of P_3 can be a subset of a solution in P_1, and vice versa. Assume that s is a nonempty solution of P_1 and P_3. This means that s must satisfy C and $\overline{C}$ simultaneously. Since $C \cap \overline{C} = \varnothing$ by definition, there is no solution for both P_1 and P_3. ■

In Figure 2(c) and (d), applying $X_1 = b \overset{ACT}{\rightarrow} X_3$ leads to two problem spaces, the first with $X_1 = b$ and X_3, and the second with $X_1 \neq b$. It follows directly from the lemma, that an always require constraint (AR constraint [27]) of the form $C \overset{ACT}{\rightarrow} X$ with C: $active(Y)$, $Y \neq X$ only creates one problem space.

Consider now the combination of a set of activity constraints. We assume here that all activation conditions are relevant, and that there is no cyclic dependency between activity constraints. We can prove the following:

Theorem 1 *The minimal solutions of the conjunction of a set of activity constraints* $ac_i : C_i \overset{ACT}{\rightarrow} X_i$ *with* $i = 1, \ldots, n$ *in a CondCSP are found in the cross-product* $\{P_{11}, P_{13}\} \times \cdots \times \{P_{n1}, P_{n3}\}$ *formed of all subspaces* P_{i1}, P_{i3}.

Proof: By induction on the problem spaces. ■

The combination of subspaces in the search trees shown in Figure 2(c) and (d) leads to the following problem spaces containing minimal solutions: $X_1 = b, X_2 = c, X_3, X_4$ and $X_1 = a, X_2 = d$. Adding $X_3 = e, X_4$ to the first space in this example, does not add information, since X_3 and X_4 are already active in this subspace. Thus, the constraint $X_3 = e \overset{ACT}{\rightarrow} X_4$, and also $X_4 = g \overset{ACT}{\rightarrow} X_3$ are redundant in this space.

Case 2 The constraints in a super-node have to be applied. A search in the combination of activation conditions of all constraints in the super-node has to be conducted. The consistent environments are retained and added to the existing problem spaces. Some activity constraints are *redundant*, and can be removed from the search space:

Corollary 1 *Given a problem space P with X active, and an activity constraint ac : $C \overset{ACT}{\rightarrow} X$ such that C is relevant in P. The union of the minimal subspaces P_1 and P_3 is identical to the original space P.*

Proof: Since all variables of C are active in P, the subspaces produced by ac are:

$$P_1 : \mathcal{V}_{P_1} = \mathcal{V}_P \cup \{X\}, \mathcal{C}_{P_1} = \mathcal{C}_P \cup \{C\},$$

$$P_3 : \mathcal{V}_{P_3} = \mathcal{V}_P, \mathcal{C}_{P_3} = \mathcal{C}_P \cup \{\overline{C}\},$$

$\mathcal{V}_{P_1} = \mathcal{V}_P = \mathcal{V}_{P_3}$ because X is already active in P. Furthermore, $\mathcal{C}_{P_1} \cup \mathcal{C}_{P_3} = \mathcal{C}_P$ because $\{C\} \cup \{\overline{C}\}$ is the set of all value combinations over the domains of $Vars(C)$. Thus, $P = P_1 \cup P_3$. ■

Example. In Figure 2, the super-node N_3 leads to the combinations $X_3 = e, X_4$ or $X_3 \neq e$, which are added to all branches in the search tree that contain X_3, but have not yet activated X_4. Similarly, $X_e = g, X_3$ or $X_4 \neq g$ are added to those branches that already contain X_4, but not yet X_3. In this example, the only branches at which an addition causes nonredundant information are found to be inconsistent due to the compatibility constraint between X_1 and X_2. ■

COMPUTATIONAL COMPLEXITY: Although removing transitive edges, finding strongly connected components in the direction graph, and applying the longest path algorithm to the reduced graph are polynomial-time algorithms, the complexity for generating all problem spaces of a CondCSP is still in $\mathcal{O}(2^{|C^A|})$ where $|\mathcal{C}^A|$ is the number of activity constraints. For each activity constraint the number of problem spaces is doubled in the worst case. The worst case is reached for *incomparable* activity constraints, i.e. those constraints which cannot be related to each other in the strict partial order.

2.2.4 The Algorithm CondCSP

Given a conditional constraint satisfaction problem $\mathcal{P} = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$ with numeric and discrete variables, the algorithm **CondCSP** shown in Figure 4 successively generates all standard CSPs (problem spaces) containing the minimal solutions. It consists of a simple

```
1.  procedure CondCSP($(\langle \mathcal{V}, \mathcal{C}^C \cup \mathcal{C}^A, V_I \rangle)$):
2.      $Vact \leftarrow V_I$
3.      $\mathcal{C}^C_{rel} \leftarrow$ relevant-constraints($Vact, \mathcal{C}^C$)
4.      $\mathcal{C}^A_{ord} \leftarrow$ compute-order($\mathcal{C}^A$)
5.      CondCSP-main($Vact, \mathcal{C}^C_{rel}, \mathcal{C}^A_{ord}, \mathcal{C}^C$)

6.  function compute-order($\mathcal{C}^A$):
7.      $\mathcal{G} \leftarrow$ directed graph from $\mathcal{C}^A$ and $V_I$
8.      $\mathcal{G}_R \leftarrow$ collapse strongly connected components in $\mathcal{G}$into super-nodes
9.      compute longest paths for each pair $(N_0, N_i)$ with root node $N_0$, $N_i \in \mathcal{G}_R$
10.     $\mathcal{C}^A_{ord} \leftarrow$ order $N_I$ according to longests paths in $\mathcal{G}_R$
11.     return $\mathcal{C}^A_{ord}$

12. procedure CondCSP-main($Vact, \mathcal{C}^C_{rel}, \mathcal{C}^A_{ord}, \mathcal{C}^C$):
13.     $ac \leftarrow$ get-relevant-ac($Vact, \mathcal{C}^A_{ord}$)
14.     remove $ac$ from $\mathcal{C}^A_{ord}$
15.     if $ac \neq \emptyset$ then
16.         $P_1 \leftarrow$ with-condition($ac, Vact, \mathcal{C}^C_{rel}, \mathcal{C}^C$)
17.         if $P_1 \neq \emptyset$ then CondCSP-main($V_{P_1}, C_{P_1}, \mathcal{C}^A_{ord}, \mathcal{C}^C$) fi
18.         $P_2 \leftarrow$ with-neg-condition($ac, Vact, \mathcal{C}^C_{rel}$)
19.         if $P_2 \neq \emptyset$ CondCSP-main($V_{P_2}, C_{P_2}, \mathcal{C}^A_{ord}, \mathcal{C}^C$) fi
20.     else new solution: $\langle Vact, Crel \rangle$ fi

21. function with-condition($C_i \overset{ACT}{\rightarrow} X_i, Vact, \mathcal{C}^C_{rel}, \mathcal{C}^C$):
22.     $Cnew \leftarrow C^C_{rel} \cup \{C_i\}$
23.     $Vnew \leftarrow Vact \cup \{X_i\}$
24.     $Cnew \leftarrow Cnew \cup$ relevant-constraints($Vnew, \mathcal{C}^C$)
25.     if locally-consistent?($\langle Vnew, Cnew \rangle$)
26.         return($\langle Vnew, Cnew \rangle$)
27.     else return($\emptyset$) fi

28. function with-neg-condition($C_i \overset{ACT}{\rightarrow} X_i, Vact, \mathcal{C}^C_{rel}$)
29.     $Cnew \leftarrow C^C_{rel} \cup \{\overline{C_i}\}$
30.     if locally-consistent?($\langle Vact, Cnew \rangle$)
31.         return($\langle Vact, Cnew \rangle$)
32.     else return($\emptyset$) fi
```

Figure 4. Algorithm generating the standard CSPs containing minimal solutions.

tree traversal starting at the root node, composed of the set of initially active variables V_I. Initially, **Relevant-constraints** is applied to identify all compatibility constraints that are relevant for the set of active variables (in this case V_I). An ordered list of activity constraints $\mathcal{C}^A_{ord}$ is generated by **compute-order** using the *DR* relation and the representation of a reduced graph $\mathcal{G}_R$. The main algorithm **CondCSP-main** then proceeds as follows. The next activity constraint *ac* is retrieved from $\mathcal{C}^A$, only if all the variables in its condition are active (**get-relevant-ac**). Then, the algorithm successively constructs the two sub-problem spaces by applying **with-condition** and **with-neg-condition** to *ac*, the set of activated variables *Vact*, and the set of constraints $\mathcal{C}^C_{rel}$, relevant for *Vact*. **With-condition** adds the new variable generated by *ac* to *Vact*, the condition of *ac* to $\mathcal{C}^C_{rel}$, and applies **relevant-constraints** to the *Vact* augmented by the new variable. **With-neg-condition** only adds the complement of *ac*'s condition to $\mathcal{C}^C_{rel}$. Both calls result in a new set of active variables, and a set of relevant constraints, i.e. a new standard CSP. Additionally, if the local consistency check (**locally-consistent?**) results in empty labelings, the according subproblem space can be pruned away. Finally, **CondCSP-main** is applied recursively. The algorithm stops when all activity constraints in $\mathcal{C}^A_{ord}$ have been considered.

2.3. The Mixer Example Revisited

Consider the mixer example shown earlier. Initially, two variables are given, variable M representing an instance of an industrial mixer which can be of type mixer, storage tank, or reactor, and the mixing task MT, which can be a dispersion, entrainment, suspension, or blending. We extend the example to take into account the following activity constraints:

$$AC_1\ M = reactor \wedge V \overset{ACT}{\rightarrow} Co,$$

$$AC_2\ V.volume \geq 150 \overset{ACT}{\rightarrow} C,$$

$$AC_3\ M \overset{ACT}{\rightarrow} V,$$

$$AC_4\ V \overset{ACT}{\rightarrow} V.volume : real : [0, 1000].$$

AC_1 and AC_2 describe the optional variables cooler *Co*, and condenser C of the product, depending on values of M and $V.volume$. The first constraint states that if the mixer chosen is a reactor, the mixer vessel requires an integrated cooler. The second constraint requires a condenser if the vessel volume is large. And finally, AC_3 generates a vessel for the mixer, and AC_4 a property volume for the vessel. One possible ordering of the activity constraints is AC_3, AC_4, AC_1, AC_2. After having generated the vessel and its

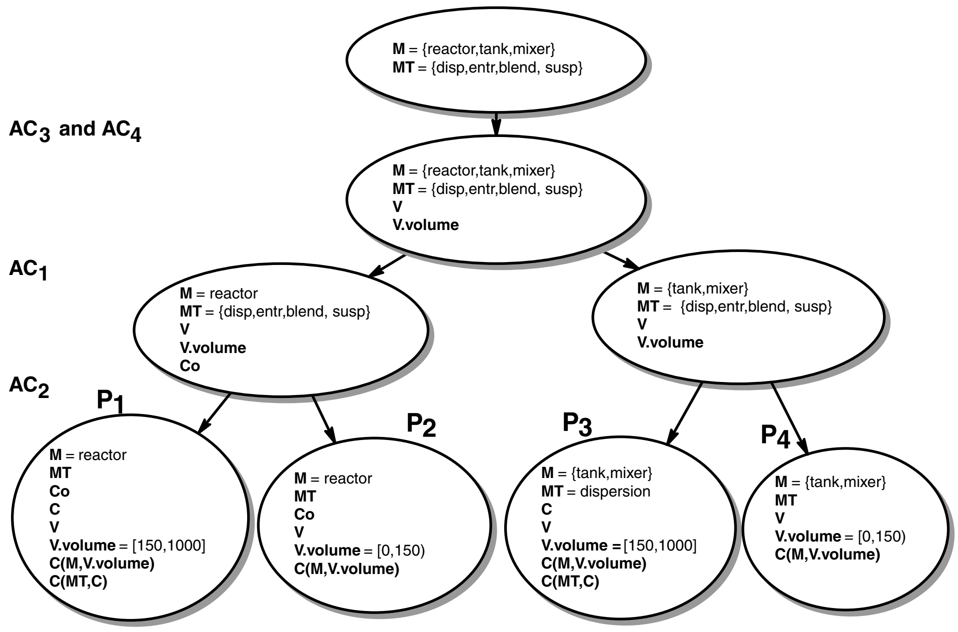

Figure 5. Four standard CSPs are defined by two activity constraints introducing a cooler and a condenser for the mixer configuration.

volume, constraints AC_1 and AC_2 are applied. They generate four problem spaces with the variables:

$$P_1: \mathcal{V}_{P_1} = \{M = reactor, MT, V, V.volume, Co, C\},$$
$$P_2: \mathcal{V}_{P_2} = \{M = reactor, MT, V, Co\},$$
$$P_3: \mathcal{V}_{P_3} = \{M = \{storagetank,\ mixer\}, V, V.volume, C\},$$
$$P_4: \mathcal{V}_{P_4} = \{M = \{storagetank, mixer\}, MT, V, V.volume\}.$$

Some of the compatibility constraints restrict their solution subspaces:

$C(M, V.volume)$:	the volume of a mixer of type reactor must be smaller than 100,
$C(MT, C)$:	a condenser is only necessary for mixing tasks of type dispersion.

The variable $V.volume$ with domain $[0, 1000]$ makes relevant the compatibility constraint $C(M, V.volume)$ in all problem spaces, and $C(MT, C)$ becomes relevant in the spaces P_1 and P_2. If each subspace generated in the tree is checked for local consistency, as proposed in Figure 4, P_1 is detected to be inconsistent, due to the constraints $C(M, V.volume)$, $M = reactor$ and the condition $V.volume \geq 150$. Local consistency algorithms can be applied to the intermediate problem spaces in this tree in order to prune them as early as possible, using the relevant constraints and conditions. We therefore, introduce in the second part of this paper, new algorithms for local consistency in mixed CSPs, and search techniques for solving standard mixed CSPs.

2.4. Related Work

A number of approaches exist in order to treat constraints that are added dynamically during problem resolution. The most famous is constraint logic programming, in which constraint satisfaction methods implement the mechanism. A number of languages for solving combinatorial problems exist. CHIP [22] has been designed to solve discrete problems, Prolog III [11] and Prolog IV [3] involve integers, booleans, and reals.

A different approach has been taken by Mittal and Falkenhainer [27]. They found that a changing number of variable instances in a constraint problem is best encoded by activity and compatibility constraints. Their algorithm embeds the activation of variables smoothly into the traditional constraint solving algorithm for discrete variables. Approaches similar to this include the model of composite CSP [32], and of free logic [7]. The model of composite CSP can be seen as the result of three concepts: dynamic constraint satisfaction, hierarchical domain constraint satisfaction, and meta-problems in constraint satisfaction. A variable value in composite constraint satisfaction can be atomic or composite, in the sense that it defines an entire sub-problem. The advantage of the composite CSP formalism is that consistency and search algorithms in constraint satisfaction are easily adapted. Bowen and Bahler [7] handle the conditional existence of variables in a mathematically well-founded fashion by formulating a CSP as a set of sentences in first-order free logic. Their approach subsumes dynamic CSPs introduced by Mittal and Falkenhainer, as they handle infinite variable domains and quantified constraints. However, the existence of each new variable has to be proven in the logical framework, which is very expensive computationally.

Recent research has focused anew on conditional constraint satisfaction. In [33], ways for detecting redundancies and inconsistencies in activity constraints are described. A new definition of conditional CSP is presented in [36] in order to keep the computational complexity of the decision problems for conditional CSPs in *NP*. In this definition, the minimality condition is replaced with a fixpoint condition, which also allows a generalization of activity constraints to include disjunction and default negation, without changing the computational complexity. It is also shown that the formalism of CondCSP, has more expressive power than the standard CSP paradigm, since it allows the representation of problems that are nonmonotonic in their nature. A representation of such problems as standard CSP can be achieved for conditional problems by using Null values to represent inactive variables. This turns out to be a very awkward representation in many cases, especially when numeric variables are involved [17, 36].

3. Search in Standard CSPs

Research in the domain of discrete standard CSPs has shown that local consistency techniques, such as arc-consistency, which removes locally inconsistent value pairs, applied during backtrack search avoid part of the thrashing behavior ([44, 26, 28]). After each value instantiation, local consistency is applied to the future, not yet instantiated variables. Existing search algorithms embedding local consistency vary in the degree of filtering: Forward checking (FC) [20] prunes labels of the variables connected by constraints to the currently instantiated variable, directional arc-consistency lookahead (DAC-L), and bidirectional arc-consistency lookahead (BDAC-L) [38] apply one pass of arc-consistency to all future variables, and maintaining arc-consistency (MAC) [31], apply arc-consistency among all the variables [6]. Another parameter that influences efficiency in solving discrete CSPs is the variable ordering heuristic, i.e. the way in which the next variable to instantiate is chosen from the set of future variables. A general variable ordering heuristic is the first-fail principle [20]. It tries to maximize the possibility of discovering failures early, by instantiating the variable next which is most constraint. Measures for constrainedness include minimum domain size, and maximum degree (maximum number of neighbor variables to which the variable is connected by a constraint [13]). Combinations of several strategies have been successfully applied in [6] and [16].

While there has been much research involved in solving discrete CSPs, numeric constraints still pose many difficulties. This is mainly due to the fact that a numeric variable can take infinitely many possible values in its domain. It has, therefore, been natural to consider filtering techniques as a means for narrowing down the search space as early as possible during search. In [12], a series of negative results are reported when applying the original algorithm of Waltz to numeric constraint problems, including the fact that the algorithm does not guarantee a locally consistent labeling, and often fails to terminate. These initial results have been improved in [14], where a correct propagation rule for reducing one variable label (also called refine operator) is proposed for binary numeric constraints. It relies on the propagation of interval values through sets of constraints defined on the same pair of variables, called total constraints. Furthermore, this rule takes into account all the local extrema of the region defined by the total constraint, and not only intersections of interval bounds with the constraints, as proposed in bound consistency [25], tolerance propagation [23], CLP(intervals) [2], or box consistency [39].

Convexity conditions for tractable global consistency in numeric domains are defined in [34, 35]. The global consistency method relies on a discretized representation of the constraint regions as 2-k-trees. This representation allows the combination of regions of simultaneous constraints, e.g. defined on the same variables, using logical instead of numeric operators. If the constraint problem satisfies the convexity conditions, (3, 2)-relational consistency,[3] which can be assured in polynomial time ($\mathcal{O}(N^5)$ with N the number of variables), is sufficient for establishing global consistency. In the general case, a constraint problem can be decomposed into sub-problems verifying the convexity conditions. The drawback is that the discretized representation of the constraints is space-consuming and the polynomial complexity of the algorithm ($\mathcal{O}(N^5)$ with N the number of variables) is high.

Search methods using interval analysis [29] have been used extensively in nonlinear optimization, and in solving equation systems resulting in single solutions. In a typical branch and bound approach, the constraint region is subdivided into a finite number of subregions (sets of intervals) that have to be tested for the optimum using interval analysis. If a test fails on an interval region, it is guaranteed not to contain a solution, and can be discarded. In Figure 6, interval splitting is applied to find the maximum in Y of the shaded region. Boxes 1, 2, and 3 can be discarded by interval analysis. Further splits in box 4 are necessary to find a reasonable approximation of the optimum. Consistency algorithms with a good pruning capability are thus important, in order to reduce the number of necessary splittings. Newton and its successor, Numerica [41] are search algorithms for numeric CSPs which apply interval methods coupled with consistency techniques. During a branch and bound search, box-consistency is applied to narrow down the variable labels. Both algorithms are mainly thought for constraint systems, the solution set of which is discrete, i.e. in which the numeric solutions are isolated, and for finding global optima in a constrained problem [42]. Other approaches using interval narrowing techniques for solving numeric CSPs include CIAL [8, 9], tolerance propagation [23], and dynamic backtracking [24].

In constraint logic programming, the integration of solvers for processing various types of constraints has a long tradition. Two types of integration are distinguished: algorithms based on delay mechanism, and those based on local consistency and propagation. In the former, the original constraint problem is divided into two subsystems. The first is solved while, the latter is delayed. Values propagated from the first resolved subproblem, are then propagated to the second, in the hope that it can be simplified. Examples are Prolog III and CLP(R) [21], in which the linear part of a constraint problem on reals is solved first, while the nonlinear part is delayed. Another approach employing several constraint solvers is proposed in ([30, 37]). First, each solver is applied to the appropriate numeric

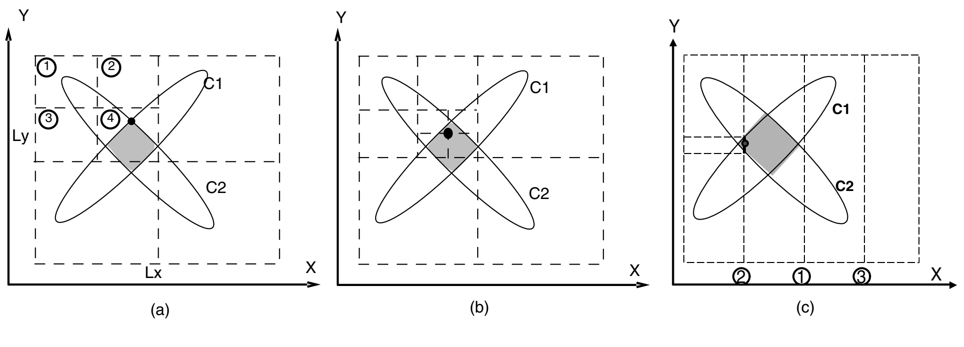

Figure 6. Search in numeric CSPs. (a) interval method, (b) backtrack search instantiating the midvalue of an interval, (c) backtrack search in which instantiated midvalues of intervals are propagated using forward checking.

or discrete subproblem. In the second step, a global solution is reached by combining partial solutions into a global one. If this combination fails, new partial solutions have to be generated. A drawback is also that mixed constraints cannot be used by the solvers to narrow down the search space, but only to check the global solution. To obtain a tighter integration of different solvers in one search framework, the use of narrowing operators to solve mixed constraints is discussed in [1]. The narrowing operators are defined as closure operators over lattices, the so-called *approximation domains*. These ideas have also been integrated into PrologIV. In our work, we use the idea of approximation domains in order integrate mixed constraints in the original constraint framework, as proposed by Mackworth [26].

3.1. A New Method to Solve Mixed CSPs

In this paper, we concentrate on search methods for mixed CSPs that are underconstrained and characterized by many contiguous solutions represented as solution subspaces. The mixer example and similar configuration or design problems fit into this category, as they are mostly defined by inequality constraints. In order to *approximate the solution subspaces* we employ arc-consistency, also called local consistency, as a filter during search. Local consistency methods identify inconsistent subspaces that can be removed from the search space. However, the resulting solution subspaces are only locally consistent, that is they are not guaranteed to contain a solution. Such a guarantee can be obtained by applying so-called *safe* algorithms [41], once a value assignment or an approximation of the solution spaces have been found.

Consistency techniques require a compact representation of the allowed combinations of consistent values. Such combinations of values are called *labels*. The results of arc-consistency can be directly encoded by variable labels which represent the locally consistent values of a variable. A *locally consistent labeling* corresponds to a labeling representing the results of a local consistency method.

A generic way to solve a mixed CSP is to exploit similarities between search in discrete and numeric CSPs. In general, numeric CSPs are solved using interval methods. These methods create a tree of intervals through bisection (Figure 6(b)), and then intervals are pruned using interval analysis. Discrete CSPs, on the other hand, are solved by enumerating combinations of variable values. We combine these two methods and propose a systematic enumeration as follows:

- Bisection generates a tree of intervals from which a tree of midpoint values is obtained.
- The definition of midpoint value depends on the domain representation: discrete variable labels can be represented by a set of integer intervals, e.g. the label $\{a, b, d\}$ of a discrete variable domain $\{a, b, c, d\}$, arbitrarily ordered by a:1, b:2, c:3, d:4, is represented by the list of intervals $[1, 2], [4, 4]$. The midpoint value for a discrete interval $[a, b]$, $a, b \in \mathbb{N}$ can, for example, be computed by $\mathit{floor}(a + b/2)$. Continuous variable labels are represented by a set of real intervals. The midpoint value is thus the midpoint of one of the intervals $[c, d]$, with $c, d \in \mathbb{R}$ given by the formula $(c + d/2)$.

- The tree of midpoint values is searched in a depth-first manner starting, for example, with the first interval in a variable label. However, the locally consistent intervals are kept for each variable. If a consistency check fails, the algorithm can directly continue with the next interval in a label.

The algorithm works as follows: **split** instantiates discrete and numeric variables successively to a single value m, the midpoint, of the current interval I of their label L, and checks the obtained assignments against the constraints in **treat-value**. When a consistency check fails, the algorithm first backtracks to the interval to the left of the failed value $[\mathit{left}(I), m)$, and then to the interval to the right $(m, \mathit{right}(I)]$, and searches them recursively. Domain splitting continues until a given distance w between adjacent values is reached. w is 1 for discrete domains, and is chosen for numeric domains such that a reasonable number of values in the label are checked. It is clear that solutions can be missed if one of the solution subspaces is smaller than a box with side length $2w$.

An example of a simple backtrack algorithm with value instantiation is presented in Figure 6(b). The dot indicates one possible midpoint solution. Here a solution is found after three levels of splitting, i.e. in the worst case the centers of sixteen cubes have to be tested.

The algorithm, as we applied it to the examples, is enhanced by local consistency. The function **check** in Figure 7 applies forward checking to prune neighbor labels using the refine operators defined in the next section. It takes the set of labels L, the constraint set $\mathscr{C}$, and the already obtained assignments $\mathscr{S}$ as input, and returns the changed labels and an ok-status indicating if an inconsistency has been found in the assigned variables and the constraints, or not. If no inconsistency has been detected (*ok* is true), **recursive-search** is applied to the next variable, otherwise the latest assignment is removed from the assignment list $\mathscr{S}$, and **unwind-labels** resets all labels that have been reduced by **check** to the state before the latest assignment. An example of the power of local consistency during search is given in Figure 6(c). Already at the second split on variable X, locally consistent values for Y are found. Y is then instantiated to the midpoint of the interval of the locally consistent values.

Our approach is very generic in that the search engine is the same for discrete and numeric variables, and that special *refine operators* can be associated to each constraint type (discrete, numeric, and mixed) to prune variable labels. The search algorithm, together with forward checking, is particularly efficient when the solution space consists of few contiguous regions. The reason is that the regions become constrained during instantiation, and local consistency applied to a very constrained problem forces a decision on backtracking quickly. Furthermore, local consistency can drastically reduce the interval size, and thus the number of splitting iterations in numeric constraint problems. Finally, the local consistency algorithm for numeric domains described in the next section results in improved accuracy.

```
1.  procedure recursive-search($\mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S}$)
2.      if all variables of $\mathcal{V}$ are assigned in $\mathcal{S}$ then
3.          return on first solution: $\mathcal{S}$
4.      else
5.          $\mathcal{V} \leftarrow$ reorder-variables($\mathcal{V}$)
6.          $X \leftarrow$ first variable in $\mathcal{V}$ unassigned in $\mathcal{S}$
7.          $I_X \leftarrow$ next interval from $L_X \in \mathcal{L}$
8.          split($I_X, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S}$)
9.      fi

10. procedure split($I_X, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, width, \mathcal{S}$)
11.     $m \leftarrow$ midpoint($I_X$)
12.     treat-value($m, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, \mathcal{S}, width$)
13.     if $right(I) - left(I) > width$ then
14.         split($X, [left(I_X), m), \mathcal{V}, \mathcal{L}, \mathcal{C}, w$)
15.         split($X, (m, right(I_X)], \mathcal{V}, \mathcal{L}, \mathcal{C}, w$)
16.     fi

17. procedure treat-value($m, X, \mathcal{V}, \mathcal{L}, \mathcal{C}, \mathcal{S}, width$)
18.     $oldLabels \leftarrow \mathcal{L}$
19.     $L_X \leftarrow m$, $L_X \in \mathcal{L}$
20.     $\mathcal{S} \leftarrow$ add $X = m$ to $\mathcal{S}$
21.     $\mathcal{C} \leftarrow$ add-type-3-constraint($\mathcal{C}, X = m$)
22.     $newLabels \leftarrow$ check($X, \mathcal{L}, \mathcal{C}$)
23.     if $newLabels \neq \emptyset$ then
24.         recursive-search($\mathcal{V}, newLabels, \mathcal{C}, width, \mathcal{S}$)
25.     fi
26.     $\mathcal{S} \leftarrow$ remove $X = m$ from $\mathcal{S}$
27.     $\mathcal{C} \leftarrow$ remove-type-3-constraint($\mathcal{C}, X = m$)
28.     unwind-labels($newLabels, oldLabels$)
```

Figure 7. Search algorithm for a mixed CSP with interval splitting on backtracking.

3.2. Variable Ordering and Value Enumeration Strategies

As a second strategy to improve the efficiency of search, **recursive-search** integrates dynamic variable ordering (**reorder-variables**). Before the next variable is instantiated, variables are reordered according to the following strategies:

- Value enumeration for discrete variables may also refine numeric variable labels due to mixed constraints.

- Discrete value combinations occurring in a general mixed constraint of type 3 (see section on local consistency for mixed constraints) impose further numeric constraints on the problem.

Discrete Variables occurring in a general mixed constraint of type 3, combining a discrete constraint with a numeric one, are enumerated first, since their activation adds numeric constraints to the problem (**add-type-3-constraint**). Then, other discrete variables are enumerated subsequently constraining numeric variables through mixed constraints. For discrete variables, the ordering max degree + min domain (first fail principle), and for the numeric variables, the ordering max degree + max domain performs well on typical underconstrained problems, e.g. derived from configuration or design tasks.

In constraint problems containing many inequalities, the enumeration of values in the numeric labels results in very similar value assignments at a distance w of the previous assignment. The general algorithm described in Figure 7 can easily be changed to enumerate all possible assignments in the discrete search space combined with a unique instantiation for the numeric variables. This is the value enumeration strategy we chose for the examples on the mixer configuration, and the steel-structure design presented later.

3.3. Local Consistency for Numeric and Mixed Constraints

The function **check** in Figure 7 applies forward checking using the **refine** operator on each pair of variables involving X. The condition that has to be satisfied by the refine operator for achieving arc-consistency on ternary constraints is shown in Figure 8 ([15]). In numeric CSPs, higher-arity constraints are common and can be transformed into an equivalent CSP of ternary constraints [34]. For each refine step, the resulting label of the second variable is intersected with its original label and updated, unless it is empty. Otherwise, propagation is stopped, and false is returned. For each type of constraint, discrete, numeric, and mixed, specific refine operators are implemented. An implementation of the discrete refine operator can be found in the literature ([4, 5, 40]).

3.3.1 A Numeric Refine Operator

The refine function over numeric constraints has been implemented using as a basis the propagation method proposed in [14], as well as its extension to ternary constraints defined in [15, 17]. In the original paper, it is shown that arc-consistency for binary numeric constraints can indeed be reached by propagating all constraints between the same variables simultaneously. A particular difficulty is that the conjunction of several constraints, the *total constraint*, can form complex region structures, especially when ternary constraints having two variables in common, but differing in the third one, are considered. One propagation step must compute the projections of subsets of these regions. A simple refine operator is defined which can carry out this propagation

```
1.  procedure check$(X_i, \mathcal{L}, \mathcal{C})$
2.      $N \leftarrow \{(X_j, C^{tot}_{X_iX_j\ldots}) | i \neq j \wedge C^{tot}_{X_iX_j\ldots} \in \mathcal{C}\}$
3.      while $N \neq \emptyset$ do
4.          remove element $(X_i, X_j, C^{tot}_{X_iX_j\ldots})$ from $N$
5.          $L_{new} \leftarrow L_{X_j} \cap$ refine$(L_i, L_j, C^{tot}_{X_iX_j\ldots})$
6.          if $L_{new} = \emptyset$ then return(false) fi
7.          if $L_j \neq L_{new}$ then $L_j \leftarrow L_{new}$ fi
8.      od
9.      return$(\mathcal{L})$

10. procedure refine$(L_i, L_j, C^{tot}_{X_iX_j\ldots})$
11.     $L \leftarrow$ all values $v_j$ of $L_j$ such that
12.     there exists a value $v_i$ in $L_i$ such that $\forall C_{X_iX_jX_k} \in C^{tot}_{X_iX_j\ldots}$
13.     there exists also a value $v_k$ for the third variable $X_k$ such that
14.     $C_{X_iX_jX_k}$ is satisfied by $v_i, w_j, v_k$
15.     return$(L_j)$
```

Figure 8. Forward checking and refine operator for ensuring local consistency over pairs of variables. $C^{tot}_{X_iX_j\ldots}$ defines the total constraint composed of all ternary constraints on variables X_i and X_j.

without ever explicitly generating the structure of the total constraint. It is based only on counting the succession of local *extrema* and *intersections* of the constraint curves. With the exception of intersections with interval bounds, these points can be preprocessed, and the propagation, is thus, just as efficient as propagating one constraint at a time. For constraints with higher arity, either the binary refine operator is applied during search when all but two variables of the constraint have been instantiated, or the extension to ternary constraints proposed in [15, 17] can be used. Since any constraint with an arbitrary number of variables can be decomposed into an equivalent network of ternary constraints [34], the refine operator for ternary constraints can be applied to solve numeric CSPs of higher arity. It has been shown [17] that, for achieving arc-consistency, the ternary operator computes the correct projection of a 3-dimensional region on a single variable, as long as the feasible regions are topologically simply connected.

As an example, consider the intersection of two parabolas in the intervals $I_X = I_Y = [-10, 10]$ shaded in Figure 9. Propagating constraints individually results in a refinement for Y: $I_Y = [-3.74, 3.74]$. Faltings' operator detects the inconsistent region between -2 and 2: $L_Y = \{[-3.74, -2][2, 3.74]\}$.

If we compare the results of a single refinement step of 2B-consistency [25], box-consistency [2, 41], and single-propagate [14] applied to the example of the two ellipses with the initial interval $[-10, 10]$ for X and Y, we find that:

- X varies between -2.8213 and 2.8213.
- Y varies between -2.8213 and 2.8213.

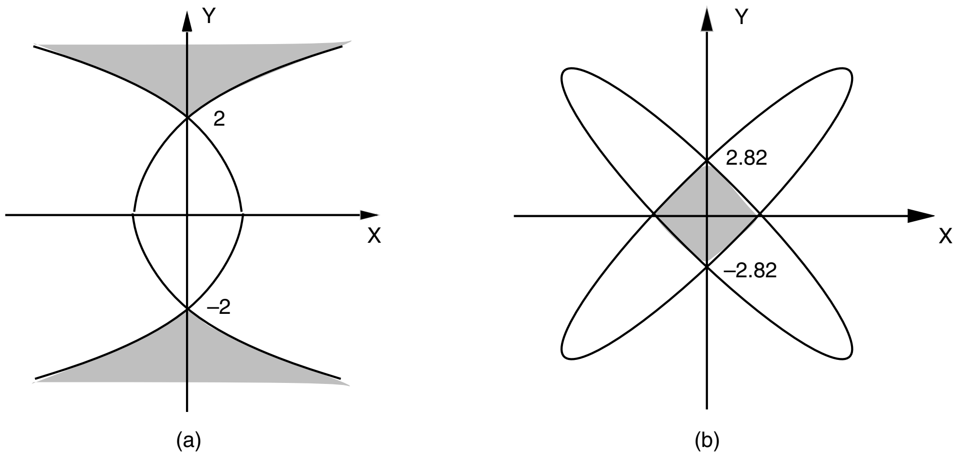

Figure 9. (a) Intersection formed by the parabolas $X+4-Y^2 \leq 0$ and $X-4+Y^2 \geq 0$, and (b) intersection between two ellipses $0.525*X^2+0.95*X*Y+0.525*Y^2 \Leftarrow 0$ and $0.525*X^2-0.95*X*Y+0.525*Y^2 \Leftarrow 0$.

Both bounds are not found by algorithms propagating constraints individually, since they are given by the intersection of the two constraints. $L_X = L_Y = \{[-2.8213, 2.8213]\}$ is exactly the labeling that simple-propagate produces. Box-consistency and 2B-consistency cannot reduce the initial intervals, which is to be expected because the constraints are propagated individually, and the projection of each constraint extends over the interval $[-10, 10]$. Furthermore, it has been shown in [10] that 2B-consistency achieves a weaker filtering than box-consistency, since 2B-consistency requires a decomposition of constraints in which multiple variables occur. The refine operator simple-propagate for binary constraints computes the exact projection of a feasible constraint region onto the axes [14]. This proof has been extended to simply-connected ternary regions in [17]. All other operators cited above will result in nonoptimal labeling, as soon as more than one constraint is defined on the same set of variables.

3.3.2 Mixed Refine Operators

In addition to discrete and numeric constraints, mixed constraints also appear in real world problems. Remember that we defined a mixed constraint as a constraint defined over numeric and discrete variables. Typical examples of mixed constraints are identified in Table 1. We distinguish three different types:

Type 1 Constraints associating discrete to interval values. An example of this type is $C(M, V.volume)$ defined by the tuples $\{(reactor\ [0,100])\ (storagetank\ [0,1000])\ (mixer\ [0,1000])\}$ restricting the vessel volume in function of the mixer type.

Type 2 Numeric constraints to which a discretization operator is applied (see Table 2), for example $nbPiers = \lceil length/typicalSpan \rceil$.

Table 2. A list of operators that define discontinuous functions. The operand r is a real, i, p and q are integers

Operator	Name	Approximation
$i = \lceil r \rceil$	ceiling	$r \leq i < r+1$
$i = \lfloor r \rfloor$	floor	$r-1 < i \leq r$
$i = round(r)$	round	$r - 1/2 \leq i < r + 1/2$
$i = trunc(r)$	trunc	$r-1 < i \leq r$ if $r \geq 0$
		$r \leq i < r+1$ if $r < 0$
$i = mod(p, q)$	mod	$r/q = div(r, q) + i$
$i = div(p, q)$	div	$i = trunc(r/q)$

Type 3 Mixed constraints associating a numeric constraint set to a discrete constraint set. Piecewise defined functions also fall into this category. For example, consider the constraint on the form of the vessel:

C(V, V.volume, V.diameter, V.height)

$$\begin{aligned} := \{&(hemispherical\ volume) = 1/12 * \pi * diameter^3 \\ &+ 1/4 * \pi * diameter^2 * (height - 1/2 * diameter) \\ &(cylindrical\ volume = 1/4 * \pi * diameter^2 * height)\}. \end{aligned}$$

For *mixed constraints of type 1*, the idea is to transform the original mixed constraint into a discrete constraint, on which the discrete refine operator can be applied. Since intervals appearing in a constraint are predefined and do not change, a function that transforms the intervals in a constraint into a discrete set can be defined.

Definition 6. [A transformation function for mixed constraints of type 1]

1. Order the endpoints of all intervals appearing in the constraint on the real axis, and create a set of basic intervals I_{base} which consists of closed intervals for an endpoint, and open intervals for regions between two adjacent endpoints.
2. consider I_{trans} as a subset of the power set of B, such that two adjacent intervals are merged to form a convex interval. I_{trans} forms a lattice.
3. A transformation function $T_{C,V}$ for a given constraint C and a given variable V, as well as the inverse $T^{-1}_{C,V}$ are defined by

$$T_{C,V} : I \rightarrow I_{trans}$$
$$T^{-1}_{C,V} : I_{trans} \rightarrow I$$

The transformation function $T_{C,V}$ associates a single interval I' from I_{trans} to each convex interval I, such that I' is the union of all elements in B that intersect with I. That is, $I' = T_{C,V}(I) = \cap \cup \{I_i | I_i \in I_{base} \wedge I_i \cap I \neq \varnothing\}$.

Consider, for example, the domain of $V.volume$ in $C(M, V.volume)$. The set of basic intervals for this constraint comprises 5 elements: $I_{base} = \{[0,0], (0,100), [100,100], (100,1000), [1000,1000]\}$. The constraint is thus completely discretized using this base set of intervals. $C(M, V.volume)$ is thus implemented as $\{(reactor\ [0,0])(reactor\ (0,100))(reactor\ [100,100])(storagetank\ [0,0])\ldots\}$. If the interval [500, 1000] has to be propagated through this constraint, it is first transformed into the set of intervals $\{(100,1000), [1000,1000]\}$ (the union of which is (100, 1000]). Upon propagation, M is reduced to the label $\{storagetank, mixer\}$. This is simply an implementation of *approximate domains*, as defined in [1] for numeric variables. The transformation function and its inverse are associated to a mixed constraint for each numeric variable. Before propagating an interval, which is part of a variable label, through such a constraint, the interval has to be transformed, according to T, and after the propagation step it has to be transformed back into its continuous representation according to T^{-1}, because its value might be propagated further through numeric constraints. The additional cost for this transformation is in the order of $\mathcal{O}(k * d)$, where k is the number of numeric variables in the constraint, and d the maximal number of landmarks minus one. It is negligible with respect to the complexity of a discrete refine operator.

Mixed Constraints of Type 2

Include numeric constraints with discretization operators. Typical constraints of this type use real-to-integer operators like {round, mod, div, ceiling, floor} to convert the intermediate real result into an integer. Numeric consistency algorithms are defined for functions with a continuous boundary. These operators, however, define discontinuities as shown in Figure 10. One way to treat such equations is to approximate the equation by numeric constraints, according to the definitions given in Table 2. The result of one propagation

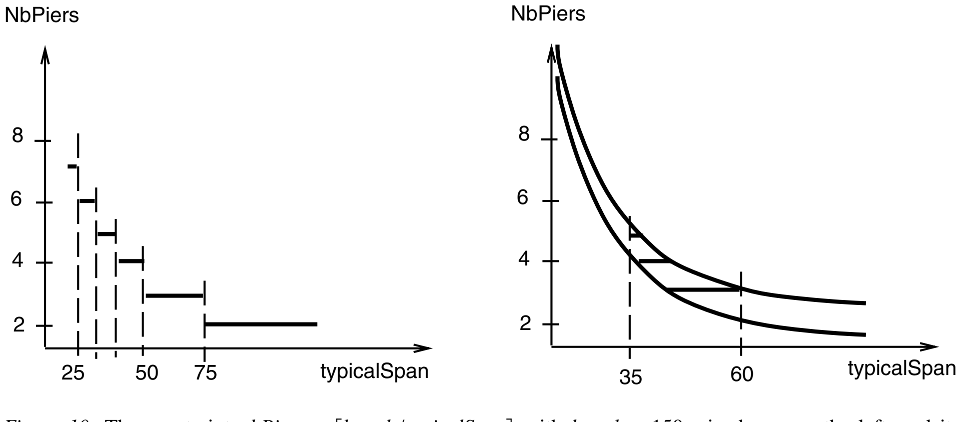

Figure 10. The constraint $nbPiers = \lceil length/typicalSpan \rceil$ with $length = 150m$ is shown on the left, and its approximation by two inequalities on the right side. Within the propagated interval of [35, 60] for typical span, several solutions exist: $nbPiers = 3, 4, 5$.

step for the integer variable will then be rounded inward to the next integer by the transformation function $T([a, b]) = [\lceil a \rceil, \lfloor b \rfloor]$ with $a \leq b$, $a, b \in \mathbb{R}$, and $T^{-1}([a, b]) = [a, b]$.

Mixed Constraints of Type 3

Associating discrete values with numeric constraints, as for example the constraints on the vessel form $C(V, V.volume, V.diameter, V.height)$, are propagated during search (**add-type-3-constraints** and **remove-type-3-constraints** in Figure 7). The strategy is to enumerate the values of discrete variables in such constraints first, in order to define exactly those subspaces in which the numeric part of the constraint is valid.

4. Practical Examples

We have tested several real world examples from configuration and design. The first example is the mixer configuration we mentioned in the introduction. Since this problem has only two activity constraints, maximally four problem spaces result from the combinations of the activation conditions and their complements, $V.volume \geq 150$, $V.volume < 150$, $M = reactor$ and $M \neq reactor$. Our CondCSP algorithm generates three locally consistent solution spaces (Table 3), and shows that the configuration containing a cooler and a condenser is inconsistent. This is due to the combined conditions in the first problem space, which are $V.volume \geq 150$ and $M = reactor$, and the compatibility

Table 3. Locally consistent solution spaces of the mixer problem

M	{mixer, storagetank}	reactor	{mixer, storagetank}
M.vessel	{cyl, hemi}	{cyl, hemi}	{cyl, hemi}
MT	dispersion	{susp, entr, blend, disp}	{susp, entr, blend, disp}
MT.heattransfer	false	true	false
MT.slurrypressure	{low, high}	{low, high}	{low, high}
MT.slurryviscosity	{low, high}	{low, high}	{low, high}
MT.slurrydensity	[1, 2000]	[1, 2000]	[1, 2000]
A.impellers	radial-turbine	$D_{Impellers}$	$D_{Impellers}$
E.power	[0.2, 5000]	[0.2, 5000]	[0.2, 5000]
I.diameter	[0.4573, 3.4658]	[0.4573, 6.9883]	[0.4573, 6.9883]
I.entry	top	D_{entry}	D_{entry}
I.position	5	D_{pos}	D_{pos}
I.power	[0.1, 2500]	[0.1, 2500]	[0.1, 2500]
I.ratio	0.329102	D_{ratio}	D_{ratio}
I.rps	[1, 23.0281]	[1, 29.2402]	[1, 29.2402]
V.diameter	[1.389, 10.5309]	[0.4817, 21.2344]	[0.4817, 21.2344]
V.height	[0.6948, 21.0618]	[0.2408, 42.4688]	[0.2408, 42.4688]
V.volume	[150, 1000]	[0.01, 100]	[0.01, 150]
V.opt1	—	cooler	—
V.opt2	condenser	—	—

constraint $C(M, V.volume)$ (Table 1), which requires the vessel volume to be less than 100 liters for a mixer of type reactor.

We choose the problem space with $M = \{mixer, storagetank\}$ and $V.volume = [150, 1000]$ in order to find single value assignments using the algorithm in Figure 7, with forward checking and dynamic variable ordering. The search heuristic used was to enumerate exhaustively the integer and discrete domains, but generating only one solution for the numeric variables. According to this search strategy, twelve possible solutions were identified in the solution space. They were found without backtracking, although the constraint graph of this example has cycles. The variable ordering strategy consisted of ordering the discrete variables according to max degree + min domain size, and considering first the M.vessel variable used in the mixed constraint. The numeric variables were ordered according to the max degree + max domain size strategy.

A second example (Figure 11) concerns the construction of steel structures, such as the single story steel building shown in Figure 12. It consists of various steel elements: frames, columns, joists, and girts, which have specific profiles, e.g. I-shaped cross section. Characteristics of profiles available on the Swiss market, such as surface or mass, are introduced in the form of tables.

The building has to resist external loads, such as wind and snow. In order to resist the horizontal loads, a longitudinal and transversal bracing system is necessary (indicated by the crosses in Figure 12). Typical stability constraints from the Swiss Norm SIA 161 were introduced as constraints: the bending of joist and girts, the biaxial bending of girt, the buckling of columns, and axial forces in diagonals of the bracing system. Geometric constraints concerning volume, surface, width, length, height of the building, spacing of the elements, frontal and lateral door width and height are also considered. Some of the variables and constraints are exemplified in Figure 11.

The design of this steel structure is formulated as a mixed CSP. Variables are the different steel elements, the spacing, length, width and height of the building, as well as the characteristics of the profiles. After transforming the constraints into ternary constraints, there are approximately 70 constraints defined over more than 100 variables. We are interested in a solution for a building having a length of 36 and a width of 24 meters, situated at 900 meters of altitude. There is a door on the front side, as well as on the

Constraints	**Variables**
$N_{k1} = K_1 * 235 * A_{Cl}$	Cl: Frame column
$K_1 = \frac{1}{\phi_1 + \sqrt{\phi_1^2 - \lambda_{k1}^2}} \leq 1$ If $K_1 > 1$ then $K_1 = 1$	E: Frame spacing
$\phi_1 = 0.5 * (1 + 0.34 * (\lambda_{k1}^2 - 0.2) + \lambda_{k1}^2$	B: building width
$\lambda_{k1} = \frac{H}{i_{Cl}} * \frac{1000}{93.91}$	Q_{tot1}: total load
$N_{q1} = 0.5 * Q_{tot1} * B * E$	Eft: joist spacing
$1.1 * N_{q1} \leq N_{k1}$	Cvt: No of braced bays widthwise
$235 * A_{Dlt} \geq 1.1 * 1.5 * F_{lt} * \sqrt{1 + (\frac{E*Cvt}{Eft*Cvl})^2}$	Cvl: No of braced bays lenghtwise

Figure 11. Variables and constraints employed in the steel-structure example. The full set of constraints can be found in [18].

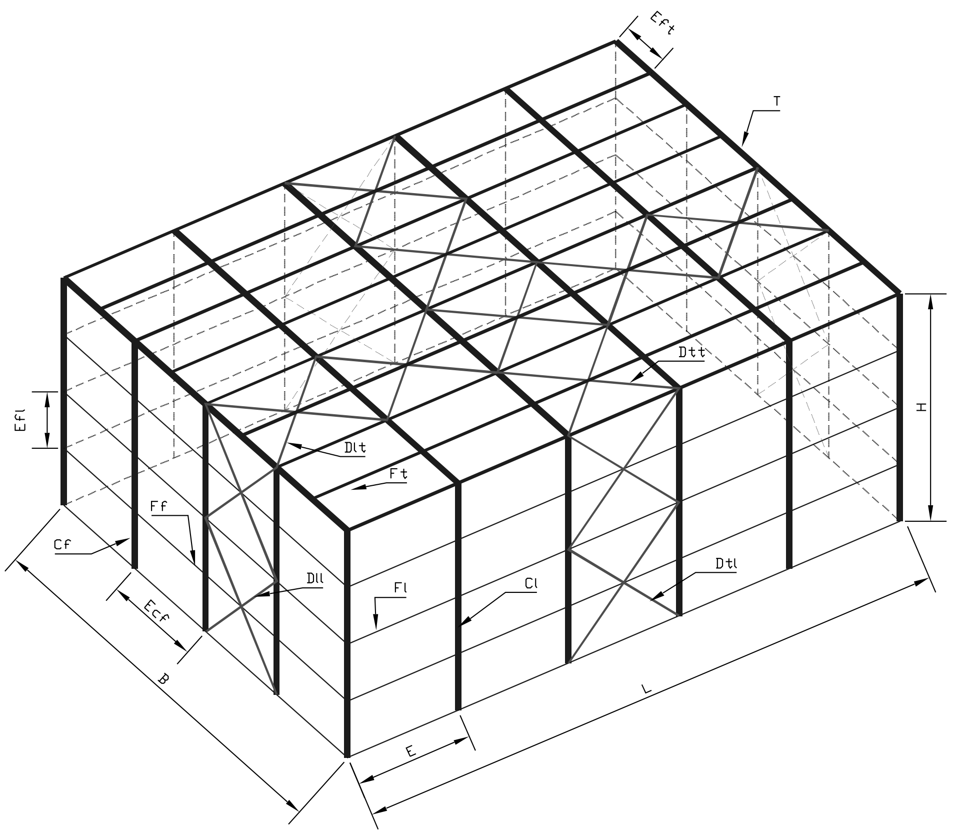

Figure 12. A single story steel building with its most important parameters.

lateral side of the building, constraining the spacing. The external loads are a lateral wind load of 800, and frontal wind load of 500. There is one solution space which takes into account the construction cost, choosing for all elements besides the girts and the transversal girt profile, the profile with minimal surface.

Finding value assignments in one of the mixed CSPs required on the order of 1 to 4 minutes using a Sparc II, two-processor machine. The current prototype version is implemented in Common Lisp, with the numeric refine operator implemented in Maple. The implementation is not optimized (process communication takes place between Lisp and Maple).

5. Conclusions

In this paper, we contribute to solving: 1) mixed constraint satisfaction problems, defined over discrete and numeric variables, 2) conditional CSPs, and 3) a combination of both, conditional mixed CSPs.

We have presented a generic search algorithm based on value enumeration, which solves mixed conditional CSPs. In order to define an enumeration scheme for the numeric part of the CSP, we derive from the interval tree typically used in interval methods, a search tree consisting of midpoint values. In addition, forward checking is applied to propagate instantiated values.

A new method for handling conditional mixed CSPs is also proposed. It generates explicitly, all those combinations of conditions which lead to the introduction of new variables. Local consistency is applied after each application of an activity condition, in order to prune away inconsistent problem spaces. Additionally, an analysis of dependencies between activity constraints results in their ordering, identifies cycles leading to an infinite generation, activity constraints not related in strict partial order established by the dependency relation, and variables that are never activated. Especially, the identification of activity constraints not related in the strict partial order may help either to decompose the problem into independent parts that can be computed in parallel, or to impose additional values to variables in the condition of these constraints, so as to reduce the computational complexity.

Local consistency is supported by refine operators specifically designed for mixed and numeric constraints. The numeric operator is able to compute the projection of a feasible region defined by several constraints without representing it explicitly. Its pruning power is superior to other methods, which propagate constraints individually. This is important to avoid backtracking and label splitting as much as possible during search.

Of critical importance in solving mixed conditional CSPs, is still the time spent to handle numeric constraints and variables. Backtracking in this part of the CSP can prove to be very inefficient, since numeric labels encode an infinite number of possible values. Finding efficient refine operators for numeric constraints is still an open research field, with many possibilities of improvement.

The lack of a standardized definition of numeric and mixed constraints currently prevents us from benchmarking existing algorithms for solving and pruning numeric and mixed conditional CSPs. Both types of constraint problems vary very much in the way they are formulated and processed. In addition, real world examples in configuration and design for example benefit from characteristics often expressed through the structure of the product to be designed. It is: 1) difficult to generate automatically mixed CSPs (especially concerning the numeric constraints), and 2) to compare the resolution of real world examples to automatically generated ones. Defining a framework in which nonstandard CSPs can be generated, solved, and compared remains a challenging task for the future.

Acknowledgments

This work was supported by the Swiss National Science Foundation under contract nos. 21-39379.93 and 20-45416.95. The authors would like to thank Timo Soininen, Laboratory of Information Processing Science, Helsinki Technology University, and Mihaela Sabin, Constraint Computation Center, New Hampshire University, for the many fruitful discussions on conditional CSPs, as well as Professor Ian Smith, Stress Analysis Laboratory, EPFL, for providing us with real-world design problems.

Notes

1. Historically, the term used to designate constraint problems in which the set of variables in a solution varies was dynamic constraint satisfaction. However, this term is used with different meanings, even within the CSP community. We therefore decided to follow the suggestion of researchers in this community and renamed it conditional constraint satisfaction.
2. According to definition 3, an activity constraint does not introduce more than one new variable. This allows for a simple interpretation of activity constraints and does not restrict the semantics, because a condition C activating n variables $X_1, \ldots, X_n$ can always be written as $C \overset{ACT}{\rightarrow} X_1, \ldots, C \overset{ACT}{\rightarrow} X_n$.
3. (3.2)-relational consistency guarantees that each triplet of constraints having two variables in common has a non-null intersection.

References

1. Benhamou, F. (1996). Heterogeneous constraint solving. In Hanus, M., & Rodríguez-Artalejo, M., ed., *Algebraic and Logic Programming, 5th International Conference, ALP'96*, Vol. 1139 of *lncs*, pages 62–76, Aachen, Germany: Springer.
2. Benhamou, F., McAllester, D., & Van Hentenryck, P. CLP (intervals) revisited. (1994). In Bruynooghe, M., ed., *Logic Programming—Proceedings of the 1994 International Symposium*, pages 124–138, The MIT Press.
3. Benhamou, F., & Touraivane (1995). Prolog IV: Langage et Algorithmes. In *Quatrièmes Journées Francophones de Programmation en Logique, JFPL'95*, pages 51–65.
4. Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1): 179–190.
5. Bessière, C., & Régin, J.-C. (1993). Arc consistency for general constraint networds: preliminary results. In *International Joint Conference on Artificial Intelligence, IJCAI-93*, pages 398–404.
6. Bessière, C., & Régin, J.-C. (1996). Mac and combined heuristics: two reasons to forsake fc (and cbj). In Fréuder, E. C., & Jampel, M., eds., *Principles and Practice of Constraint Programming*, Volume 1118, of *Lecture Notes in Computer Science*. Springer.
7. Bowen, J., & Bahler, D. (1991). Conditional existence of variables in generalized constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 215–220.
8. Chiu, C. K., & Lee, J. H. M. (1994). Interval linear constraint solving using the preconditioned interval gauss-seidel method. In Yap, R. H. C., ed., *Proceedings ILPS'94 Workshop on Constraint Languages/Systems and Their Use in Problem Modelling*, Volume 2, Ithaca, Technical Report 94/19, Department of Computer Science, University of Melbourne.
9. Chiu, C. K., & Lee, J. H. M. (1994). Towards practical interval constraint solving in logic programming. In Bruynooghe, M., ed., *Logic Programming—Proceedings of the 1994 International Symposium*, pages 109–123. The MIT Press.
10. Collavizza, H., Delobel, F., & Rueher, M. (1998). A note on partial consistencies over continuous domains. In *Principles and Practice of Constraint Programming, CP98*, Springer LNCS 1713, pp. 147–161.
11. Colmerauer, A. An introduction to PrologIII. (1990). *Communications of the ACM*, 33(7): 69–90.
12. Davis, E. (1987). Constraint propagation with interval labels. In *Artificial Intelligence* 32: 281–332.
13. Dechter, R., Meiri, I., & Pearl, J. (1989). Temporal constraint networks. In Levesque, H. J., Brachman, R. J., & Reiter, R., eds., *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 83–93, Toronto, Canada, Morgan Kaufmann.
14. Faltings, B. (1994). Arc consistency for continuous variables. In *Artificial Intelligence 65(2)*, pages 85–118.
15. Faltings, B., & Gelle, E. (1997). Local consistency for ternary numeric constraints. In *Proc. 11th Int. Joint Conf. on Artificial Intelligence, IJCAI-97*, pages 392–397.

16. Frost, D., & Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. In Mellish, C. S., ed., *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 572–578, Morgan Kaufmann.
17. Gelle, E. (1998). On the generation of locally consistent solution spaces in mixed dynamic constraint problems. Ph.D. thesis, Swiss Federal Institute of Technology, EPFL.
18. Gelle, E., Faltings, B., Clément, D., & Smith, I. (2000). Constraint satisfaction methods for applications in engineering. *Engineering With Computers*, 16(2): 81–95.
19. Gondran, M., & Minoux, M. (1986). *Graphs and Algorithms*. John Wiley & Sons, Chichester, 1st edition.
20. Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14: 263–313.
21. Heintze, N. C., Michaylov, S., & Stuckey, P. J. (1987). CLP($\mathscr{R}$) and some electrical engineering problems. In Lassez, J.-L., ed., *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 675–703. The MIT Press.
22. Van Hentenryck, P. (1989). Constraint satisfaction in logic programming. In *Logic Programming Series*. MIT Press.
23. Hyvönen, E. (1992). Constraint reasoning based on interval arithmetic: the tolerance propagation approach. In *Artificial Intelligence*, 58: 71–112.
24. Jussien, N., & Lhomme, O. (1998). Dynamic domain splitting for numeric csps. In *ECAI-98*, pp. 224–228.
25. Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *International Joint Conference on Artificial Intelligence, IJCAI-93*, pages 232–238.
26. Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8: 99–118.
27. Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In Dietterich, W., & Swartout, T., eds., *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32, MIT Press.
28. Montanari, U. (1974). Networks of constraints: fundamental properties and applications to picture processing. *Inform. Sci.*, 7: 95–132.
29. Moore, R. E. (1979). *Methods and Applications of Interval Analysis*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
30. Rueher, M. (1994). An architecture for cooperating constraint solvers on reals. In Podelski, A., ed., *Constraint Programming: Basics and Trends*, LNCS 910, Springer.
31. Sabin, D., & Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In Borning, A., ed., *Principles and Practice of Constraint Programming*, Volume 874 of *Lecture Notes in Computer Science*, Springer.
32. Sabin, D., & Freuder, E. C. (1996). Configuration as composite constraint satisfaction. In *Configuration—Papers from the 1996 Fall Symposium*. AAAI Technical Report FS-96-03.
33. Sabin, M., & Freuder, E. (1999). Detecting and resolving inconsistency in conditional constraint satisfaction problems. In *Proceedings of the AAAI'99 Workshop on Configuration*, pages 95–100.
34. Sam-Haroud, D. (1995). Constraint consistency techniques for continuous domains. Ph.D. thesis, Swiss Federal Institute of Technology, EPFL.
35. Sam-Haroud, D., & Faltings, B. (1996). Consistency techniques for continuous constraints. *Constraints*, 1: 85–118.
36. Soininen, T., Gelle, E., & Niemelä, I. (1999). A fixpoint definition for dynamic constraint satisfaction. *Principles and Practice of Constraint Programming, CP'99*, Springer LNCS 1713, pp. 419–433.
37. Tinelli, C., & Harandi, M. (1996). Constraint logic programming over unions of constraint theories. *Lecture Notes in Computer Science*, 1118: 436–450.
38. Tsang, E. (1998). No more partial and full looking ahead. *Artificial Intelligence*, 98(1–2): 351–361.

39. Van Hentenryck, P. (1997). Numerica: a modeling language for global optimization. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI '97)*, pages 1642–1647, Nagoya.

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

41. Van Hentenryck, P., McAllester, D., & Kapur, D. (1997). Solving polynomial systems using a branch and prune approach. *SIAM Journal of Numerical Analysis*, 34(2): 797–827. Also available as Brown University technical report CS-95-01.

42. Van Hentenryck, P., Michel, L., & Deville, Y. (1997). *Numerica. A Modeling Language for Global Optimization*. MIT Press.

43. van Velzen, M. (1993). A Piece of CAKE, Computer Aided Knowledge Engineering on KADSified Configuration Tasks. Master's thesis, Univeristy of Amsterdam, Social Science Informatics.

44. Waltz, D. L. (1975). Understanding line drawings of scenes with shadows. In Winston, P. H., ed., *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill.