ON THE GENERATION OF LOCALLY CONSISTENT SOLUTION SPACES IN MIXED DYNAMIC CONSTRAINT PROBLEMS

THÈSE N^{O} 1826 (1998)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Esther GELLE

Ingénieure informaticienne diplômée EPF

originaire d'Ebikon (LU)

acceptée sur proposition du jury:

Prof. B. Faltings, directeur de thèse
Prof. E. Freuder, rapporteur
Dr O. Lhomme, rapporteur
Prof. I. Smith, rapporteur

Lausanne, EPFL
1998

Abstract

The development of industrial products such as cars, mechanical tools as well as civil engineering structures includes the task of identifying components and arranging them in a product structure. This task is commonly called *configuration*. It can be formalized as a *constraint satisfaction problem* (CSP), which provides a concise mathematical model for combinatorial tasks. A CSP is defined by the variables of interest, each with a domain of possible values, and a set of constraints which restrict the allowed value combinations. The formulation of a configuration task as a CSP gives flexibility required to apply complex optimization criteria during search which often cannot be expressed by a single utility function.

Solving a CSP corresponds to finding a consistent assignment to the variables that satisfies all constraints. Solution methods for CSPs encompass consistency and search techniques. Consistency techniques are preprocessing methods which remove inconsistent value combinations before search thus reducing the search space. Furthermore, a dynamic CSP model (DCSP) makes possible the introduction of new variables and constraints during problem resolution. However, CSP techniques have been weak for handling *mixed* (discrete-continuous) as well as *dynamic* problems.

In this thesis, I present a new algorithm for solving DCSPs with mixed, continuous and discrete, constraints. It isolates and approximates the solution sets of a DCSP by local consistency techniques. To this purpose, local consistency techniques for continuous constraints had to be enhanced and integrated with existing discrete local consistency methods. Some advantages of this algorithm are:

1. It can solve problems in which the existence of a variable depends on a decision taken over a continuous value domain.
2. Different locally consistent solution spaces can be compared, thus providing additional information for decision making.
3. Local consistency techniques are also applied to search for a single solution within the resulting locally consistent solution spaces. These techniques are particularly efficient for searching in continuous domains.

The algorithm proposed is applied to a number of different problems such as conceptual bridge design, design of steel structures, configuration of trains, and industrial mixers.

Résumé

Le développement de produits industriels, comme par exemple une voiture, une pièce mécanique mais également une structure métallique en génie civil, implique l'identification des composants du produit ainsi que leur arrangement dans l'espace. Cette tâche est apellée *configuration*. Elle peut être formalisée comme un *problème de satisfaction de contraintes* (CSP en anglais), qui fournit un modèle mathématique concis pour des problèmes combinatoires. Un CSP est défini par un ensemble de variables caractérisant le problème et un ensemble de contraintes qui restreignent les combinaisons de valeurs des variables dans une solution. La formulation d'une tâche de configuration comme CSP permet d'appliquer des critères d'optimisation complexes pendant la recherche qui ne peuvent guère s'exprimer sous la forme d'une fonction objective.

Résoudre un CSP correspond à trouver une assignation de valeurs aux variables qui satisfait toutes les contraintes. Les méthodes pour résoudre un CSP incluent des techniques de consistance et de recherche. Les techniques de consistance permettent de découvrir des combinaisons de valeurs incompatibles et de les enlever avant de lancer la recherche d'une solution. Elles visent ainsi à réduire l'espace de recherche. En outre, un modèle de CSP dynamique, appelé DCSP en anglais, permet d'introduire de nouvelles variables et contraintes pendant la recherche. Cependant, les méthodes de résolution actuelles de CSP ne sont pas bien adaptées aux problèmes *mixtes* (traitant des données continues et discrètes) et *dynamiques*.

Dans cette thèse, je présente un nouvel algorithme de résolution de DCSP défini sur des contraintes discrètes et continues. L'algorithme trouve une approximation de l'ensemble de solutions du DCSP par des méthodes de consistance locale. Pour parvenir à ce but, des techniques de consistance locale pour les contraintes continues ont été améliorées et intégrées dans un algorithme comprenant des méthodes de consistance locale pour contraintes discrètes. Les avantages de l'algorithme proposé sont:

1. Des problèmes dans lesquels l'existence d'une variable dépend d'un choix dans le domaine continu peuvent être résolus.
2. Les espaces de solution localement consistants peuvent être comparés, ce qui fournit des informations complémentaires utile à la prise de décision.
3. Les techniques de consistance locale sont aussi appliquées à la recherche d'une solution individuelle dans un espace localement consistant. Ces algorithmes sont particulièrement efficaces dans le domaine continu.

L'algorithme proposé est appliqué à différents problèmes réels comme la conception préliminaire de ponts, la conception de structures métalliques en génie civil, la configuration de trains ou de mixers industriels.

Zusammenfassung

Die Entwicklung und Optimierung von Produkten wie Autos, mechanische Werkzeuge und Gebäudestrukturen besteht unter anderem darin, Einzelkomponenten aus einem Katalog zu bestimmen sowie deren Anordnung im Endprodukt festzulegen. Dieser Teil der Produkteentwicklung wird als Konfiguration bezeichnet. Konfiguration ist ein kombinatorisches Problem und kann daher als Constraint Satisfaction Problem (CSP) mathematisch modelliert werden. Ein CSP besteht aus Variablen und Constraints. Jede Variable besitzt eine Wertemenge aus der ihr ein Wert zugewiesen wird. Constraints definieren erlaubte Wertekombinationen über diese Variablen (z.B. (Un)gleichungen, diskrete Wertekombinationen).

Die Lösung eines CSPs weist den Variablen konsistente Werte zu; d. h. Werte, die keine der Constraints verletzen. CSPs werden mit speziellen Such- sowie mit Hilfe von Konsistenz-Algorithmen gelöst. Konsistenz-Algorithmen entfernen inkonsistente Wertekombinationen aus dem Suchraum und vereinfachen dadurch eine anschliessende Suche nach Lösungen. Dynamische CSPs erweitern das Modell, indem sie das Hinzufügen von Variablen und Constraints während der Suche ermöglichen. Die bisherigen Lösungsansätze unterschieden streng zwischen numerischen und diskreten CSPs. Ausserdem existierten Ansätze zur Lösung dynamischer CSPs nur im diskreten Bereich. Komplexe Konfigurationsprobleme erfordern jedoch die Kombination diskreter und numerischer Variablen sowie eine dynamische Formulierung.

In der vorliegenden Dissertation beschreibe ich ein Verfahren, das die Lösung solch kombinierter dynamischer CSPs ermöglicht. Das Verfahren isoliert die verschiedenen Lösungsräume innerhalb des Suchraumes und approximiert diese mit Hilfe von lokalen Konsistenz-Algorithmen. Dies erforderte eine Erweiterung der bestehenden lokalen Konsistenz-Methoden insbesondere für numerische Constraints und auch die Integration von numerischen und diskreten Methoden. Vorteile unserers Verfahrens sind:

1. Verschiedene lokal konsistente Lösungsräume können verglichen werden bevor eine Design-Entscheidung getroffen wird.

2. Konsistenz-Methoden erlauben zudem eine Optimierung der Suchalgorithmen, indem der Suchraum zusätzlich beschränkt wird.

Dieses Verfahren wurde an mehreren Beispielen wie Brücken-Design, Konfiguration von Zügen and Mixern, sowie bei der Konfiguration von Gebäudestrukturen angewendet.

Acknowledgments

I wish to address my sincerest thanks to Professor Boi Faltings who gave me the opportunity to work at the LIA. He introduced me to the fascinating world of constraints and drew my attention to continuous constraints.

I am grateful to Professor Eugene Freuder, Professor Ian Smith, and Dr. Olivier Lhomme, the thesis committee, for their careful reading of my thesis and the many valuable comments I received.

Many of the practical aspects which motivated my thesis are the result of discussions with Professor Ian Smith, Stress Analysis Laboratory, EPFL. I am indebted to Sylvie Boulanger, Steel Structures Institute, EPFL, for fruitful discussions on design principles and for sharing her profound knowledge about bridge design and other aspects of life with me. I would like to thank my colleagues of the LIA for all the stimulating discussions we had in the past. My special thanks go to Eric Sauthier for encouraging me to tackle a Ph.D. and for reading parts of this thesis, to Djamila Sam-Haroud, my latest room-mate, for interesting discussions on continuous constraints the results of which found their way into this thesis, to Monika Lundell for everything I have learnt in course organization and to Rainer Weigel for stimulating discussions on constraint satisfaction problems in general. I am grateful to Claudio Lottaz for reading one of the hardest parts of my thesis treating volumes moving in n-space, to Ruth Stalker, Christian Frei, and Romaric Besançon who provided useful suggestions to enhance my thesis.

Other people involved in thesis reviewing are Timo Soininen of Helsinki Technology University, Mihaela Sabin of New Hampshire University, and Berthe Choueiry of Stanford University. I owe them a precious insight into parts of my thesis work which open the way to future research.

My deepest gratitude goes to my parents for their generous support throughout all my life and to my friend, Ulrich Jousten, for his constant patience and encouragement during the hardest periods of thesis writing. He is one of the courageous persons to have read my thesis in its entirety.

This work was supported by the Swiss National Science Foundation under contract no. 21-39379.93 and 20-45416.95

Contents

1 Introduction **1**

1.1 The characteristics of design . . . 2
1.2 Constraint satisfaction problems . . . 5
1.3 Topics of this research . . . 6
1.3.1 Continuous consistency techniques . . . 6
1.3.2 Consistency techniques for mixed constraints . . . 7
1.3.3 Dynamic constraint satisfaction . . . 8
1.4 Contributions . . . 9
1.5 Guide to the thesis . . . 10

2 Definitions and Concepts **13**

2.1 Introduction . . . 13
2.2 Notational Conventions . . . 13
2.3 Example: Configuration of industrial mixers . . . 13
2.4 Constraint satisfaction problems . . . 16
2.5 Consistency techniques . . . 19
2.6 Search strategies for solving discrete CSPs . . . 21
2.6.1 General search scheme . . . 22
2.6.2 Lookahead strategies . . . 23
2.6.3 Information-gathering strategies . . . 24
2.6.4 Variable and value orderings . . . 25
2.6.5 Comparing search algorithms . . . 25
2.7 Search strategies for solving continuous CSPs . . . 26
2.7.1 Value inference . . . 27
2.7.2 Propagating degrees of freedom . . . 27
2.7.3 Optimization techniques . . . 27
2.7.4 Relaxation and perturbation methods . . . 28
2.7.5 Stochastic methods . . . 29
2.7.6 Algebraic methods . . . 30
2.7.7 Constraint logic programming (CLP) . . . 30
2.7.8 Interval analysis . . . 31
2.7.9 Consistency techniques . . . 32

3 Local consistency techniques **35**
3.1 Fix-point algorithms for local consistency 35
3.2 Local consistency for binary numeric constraints 38
3.2.1 Computing in continuous domains 39
3.2.2 Existing refine operators 40
3.2.3 A refine operator for binary numeric constraints 43
3.2.3.1 An improved propagation rule for a single interval 45
3.2.3.2 The algorithm Simple-Propagate 48
3.2.3.3 Completeness and soundness 49
3.2.3.4 Example 49
3.2.4 Implementation 51
3.2.4.1 Methods for identifying extremal points of regions 52
3.2.4.2 Classes of local extrema 53
3.2.4.3 Local extrema on individual constraint curves 56
3.2.4.4 Intersections between constraints 58
3.2.4.5 Intersections between constraints and interval bounds . . . 59
3.2.4.6 Algorithm for identify-candidates 59
3.2.4.7 Filtering relevant extrema 59
3.3 Local consistency for discrete constraints 62
3.3.1 A refine operator for k-ary discrete constraints 63
3.4 Local consistency for mixed constraints 66
3.4.1 Discrete constraints with interval values 67
3.4.2 Continuous constraints using discretization operators 69
3.4.3 Completeness and soundness 70
3.5 Summary 71

4 Local consistency for ternary numeric constraints **73**
4.1 A refine operator for ternary numeric constraints 73
4.1.1 Local consistency over k-ary constraints ($k \geq 3$) 73
4.1.2 Extension of definitions to ternary numeric CSPs 74
4.1.2.1 Classification of stationary points 77
4.1.2.2 Classification of three-dimensional regions 78
4.2 Refine operator for a single third variable 79
4.2.1 Propagating a single interval through simple regions 79
4.2.2 Example 81
4.2.3 Correctness of the propagation rule 85
4.2.4 Propagating a single interval through regions containing holes 89
4.2.5 Extension of the refine operator 90
4.2.6 Ternary constraints with different third variables 90
4.3 Implementation 93
4.3.1 Identifying stationary points 93
4.3.1.1 Local extrema and saddle points of an individual constraint surface 94

4.3.1.2 Intersections between constraint surfaces 95
4.3.1.3 Local extrema at interval bounds 96
4.3.1.4 Algorithms for filtering and identifying candidates 96
4.3.2 Special cases of stationary points 96
4.4 Summary 99

5 Systematic generation of problem spaces 101
5.1 Introduction 101
5.2 Background 103
5.3 Dynamic constraint satisfaction 104
5.4 Solving discrete DCSPs 108
5.4.1 Original DCSP algorithm 108
5.4.2 Why not use a static CSP formulation ? 109
5.5 Formalizing a design task 113
5.5.1 A design description 113
5.5.2 Components 113
5.5.3 Relations between components or between components and properties 114
5.5.4 Identical components 117
5.6 Generating problem spaces of a DCSP 118
5.6.1 Activity constraints 118
5.6.2 Combining activity constraints 120
5.6.3 General DCSP algorithm (GDCSP) 127
5.6.4 Example 127
5.6.5 Adapting GDCSP to generic constraints 130
5.6.6 Refining the GDCSP-algorithm 130
5.6.7 Completeness and soundness 134
5.7 Finite versus infinite number of problem spaces 134
5.8 Searching within locally consistent solution spaces 136
5.8.1 A generic search algorithm 136
5.8.2 Discrete-continuous constraints 138
5.9 Summary 139

6 Results 141
6.1 Introduction 141
6.2 A binary constraint set 142
6.3 Comparison between Davis' and Falting's propagation rules 143
6.4 Configuration of trains 144
6.5 Configuration of industrial mixers 146
6.6 Preliminary design of bridges 149

7 Conclusions 151
7.1 Scope of this research 151
7.2 Summary of major results 152

7.3 Applicability and limitations 153
7.4 Open research issues 153
7.4.1 Representation of continuous constraints 153
7.4.2 Treatment of equalities 154
7.4.3 Search in continuous and mixed CSPs 154
7.4.4 Extension of the DCSP model 154
7.4.5 Reverse engineering of DCSPs 155
7.4.6 Representation of results 155
7.5 Final conclusion 155

A Examples from configuration and design **157**
A.1 Configuration of an industrial mixer 157
A.1.1 Variables 158
A.1.2 Constraints 159
A.1.3 Locally consistent solution spaces 161
A.2 Preliminary design of bridges 161
A.2.1 Variables 161
A.2.2 Constraints 162

B Topology **165**

C Analysis **167**

D Graph theory **169**

List of Figures

1.1 *Many alternatives exist in conceptual bridge design. Five important decision points for a bridge design are shown: the type of bridge, profile, number of spans, section and the construction method. Commitments to certain key parameters, for example the bridge type, constrain the design considerably. Given a valley of 150 meters with a maximal height of 30 meters and a river, two alternative solutions are traced through the entire decision process: one choosing a constant-depth beam bridge and the second an arch bridge (indicated by solid lines). Courtesy, Sylvie Boulanger, Steel Structures Institute, EPFL* 3

1.2 *Direct dependencies between chapters in this thesis.* 12

2.1 *The physical decomposition hierarchy of an industrial mixer is shown together with functional dependencies indicated by arrows.* 15

2.2 *a) An example of a CSP that is 3-consistent but not 2-consistent, b) a CSP that is arc-consistent but not 3-consistent. In both examples, backtracking might be necessary in order to find a solution.* 20

2.3 *Search tree solving a small CSP by backtracking. The black nodes have been pruned from the search space. Example taken from [Kondrak, 1994]* 22

2.4 *Search tree of the CSP in Figure 2.3 solved by a) FC and b) MAC.* 24

2.5 *The relationship between various lookahead and intelligent backtracking algorithms. a) $<_{cc}$: hierarchy with respect to the number of visited nodes and b) $<_{nv}$: hierarchy with respect to the number of consistency checks.* 26

3.1 *General algorithm for ensuring consistency over pairs of variables.* 36

3.2 *a) The projection onto the Y axis of the constraint region can be computed from the intersection of L_X with the constraint. b) If the projections are computed individually for each constraint, intersections between constraints are neglected and the resulting label for Y is locally unsound. c) like b) with the effect that inconsistency is not detected by an individual propagation of the constraints. Taken from [Faltings, 1994]* 41

3.3 *A refine operator based on the intersection of interval bounds of L_X with the constraint boundary may not be able to determine if a locally consistent label exists for Y as shown in a), or find an incomplete label b) or result in a complete and sound label as shown in c).* 43

3.4 *An interval extension based on the first-order Taylor form taken around the center of I_X for the equation $Y = 1-5X+X^3/3$ over the interval $I_X = [2,3]$. The projection of the shaded region onto the Y-axis defines the interval value for the extension, which is $[-4.291, -8.292]$. The consistent values for Y are also shown. Higher order Taylor forms result in still tighter approximations. Courtesy C. Bliek, AI-Lab, EPF.* . . . 44
3.5 *The operator* **refine** *for numeric CSPs. It applies a propagation rule* **simple-propagate** *to each interval of a label and merges the resulting intervals.* . . . 45
3.6 *Two examples of total constraints. The constraint on the left consists of the two feasible regions $Q1$ and $Q2$. When propagating from X to Y, the interval Ix generates the restricted regions $R1$ and $R2$, which project into intervals $Iy1$ and $Iy2$. The example on the right shows that multiple restricted regions $R1$ and $R2$ can result from a single feasible region $Q1$.* . . . 46
3.7 *An arbitrary region bounded by the curve $B(R)$. M denote local maxima in Y and m are local minima in Y. In this example, $\alpha_Y(R, y_1) = 1$ and $\alpha_Y(R, y_0) = 0$.* . . . 48
3.8 *Binary propagation rule* **simple-propagate** *for a single interval. How unbounded regions are detected is discussed in the section on filtering relevant extrema.* . . . 50
3.9 *The table on the left side shows local extrema on constraint curves and intersections between constraints for the example represented in the figure. The graphic shows the restricted regions R_1 and R_2 defined by an ellipsis, a parabola and a line constraint. Dots are local extrema and intersections considered by* **simple-propagate**. *While circles indicate intersections of $I_X = [0.5, 2]$ with the constraints resulting in extrema in Y lying on the boundary of the restricted region, crosses show points that are no extrema in Y.* . . . 52
3.10 *A quadtree representation of the inequality $Y > arctan(1/(X-2)$. Courtesy Claudio Lottaz, AI-Lab, EPFL.* . . . 53
3.11 *There exist three important classes of local extrema: a) local extremum on constraint curve, b) intersection between two constraints and c) intersection between an interval bound and a constraint.* . . . 54
3.12 *Different types of discontinuities on functions.* . . . 55
3.13 *a) A constraint region without continuous boundary b) with a continuous boundary.* . . . 55
3.14 *Different types of singularities on constraint curves.* . . . 56
3.15 *Gradients on the region formed by an ellipsis. In a), the combination of gradient measure and condition indicate a convex maximum and minimum; in b) they indicate a concave maximum and minimum.* . . . 57
3.16 *There are three cases of intersections in two dimensions that are candidates for a local extremum in axis Y: a) no extremum, b) a minimum in X and a maximum in Y and c) a maximum in axis Y.* . . . 59
3.17 *Algorithm for* **identify-candidates**(I_X, C^t_{XY}). . . . 60

3.18 *a) Only one minimum has to be considered because both minima come from a constraint that is a line and gives rise to a single minimum. b) Both minima are counted individually because they define different minima.* 60

3.19 *a) Redundant extrema. b) Non-redundant extrema generated by an equalities* $C^1_{XY} : E^1 = 0$ *and an inequality* $C^2_{XY} : E^2 \leq 0$ *describing the inner region of a circle. The restricted region is the line segment stretching from* e_2 *to* e_1. *c) Non-redundant extrema on the common boundary of two elliptical constraints,* C_1 *defining the inner part of a circle and* C_2 *the outer part of an ellipsis. The first extremum determined by* C_1 *is of type convex maximum and the second determined by* C_2 *is a concave maximum.* 61

3.20 *Propagating* I_X *through the parabola constraint* $Y + X^2 \geq 0$ *may result in intersections that are no extrema in* Y *(the intersections are denoted by crosses in Figure b). In this case, it is sufficient to exhibit one point* (x, y) *such that* $x \in I_X$, $y \in I_Y$ *and* $(x, y) \in C^t_{XY}$ *to prove that the restricted region extends from* $-\infty$ *to* ∞. *In case a), such a point can be found at the intersection of the constraint with one interval bound of* I_Y *(*T_Y*) and in cases b) and c) it is sufficient to test the corners of the box* $I_X \times I_Y$. 62

3.21 *The refine operators of AC-3 like algorithms for binary constraints (left side) and NC for k-ary constraints(right side).* 63

3.22 *The projection of* C^1 *and* C^2 *onto* X, Y *represented as 0-1 matrices. 1 stands for a compatible value pair, 0 for an incompatibility. The intersection of both matrices forms a "triangle" of compatible pairs.* 64

3.23 *Algorithm for* **refine**$(X_i, X_j, C^t_{X_iX_j})$ *adapted to k-ary discrete constraints.* . 66

3.24 *Three types of mixed constraints.* 68

3.25 *Definition of landmarks for the vessel volume.* 69

3.26 *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:* $nbPiers = 3, 4, 5$. 70

4.1 *a) In 2 dimensions,* $\alpha_Y(R, y_0)$ *represents the number of connected intervals in R. In 3 dimensions, we would like to determine the number of connected regions in the* XZ*-plane for a given* y_0 *in a similar manner. The circles mark points at which this number changes.* 76

4.2 *Cutting through a region along* $X = x_e$ *and* $Z = z_e$ *with e being a stationary point results in two sets* S_{XY} *and* S_{ZY}. S_{XY} *has a horizontal tangent* T_X *in* (x_e, y_e) *and* S_{ZY} *has a horizontal tangent* T_Z *in* (z_e, y_e). *The local extremum in* S_{XY} *is a minimum in* Y *and the second in* S_{ZY} *a maximum in* Y. 77

4.3 *Classification of stationary points.* 78

4.4 *A propagation rule for a single third variable.* 81

4.5 *The slice containing all four intersections at* $X_2 = 1/4$ *is shown on the left side. The table on the right side indicates all stationary points in* X_3 *on the boundary of the restricted region with their index* α. 83

4.6 A three dimensional plot of the total constraint formed by $X_1^2 + 1/2 * X_2 + 2 * (X_3 - 6) \geq 0$ and $X_1^2 + X_2^2 + X_3^2 - 25 \leq 0$. 83
4.7 *The slice containing the two intersections at $X_2 = 1/4$ is shown on the left side. The second slice shows the intersection at $X_1 = 0$. The table on the right side indicates all stationary points in X_3 on the boundary of the restricted region with their index α.* 84
4.8 *A three dimensional plot of the total constraint formed by $X_1^2 + 1/2 * X_2 + 2 * (X_3 - 3) \geq 0$ and $X_1^2 + X_2^2 + X_3^2 - 25 \leq 0$.* 84
4.9 *a) Two regions with a smooth boundary touching at point P. b) More than two regions can touch at a single point if they have corners in the boundary.* 87
4.10 *The three figures show how the intersection between the sweep plane L and R and changes when L passes through a stationary point of type $T1$ or type $T2$.* 88
4.11 *Saddle points e_2 and e_3 on a region with a hole, a torus (upper image) and saddle points e_1, e_2 on a simple region, a cudgel (lower image). The left sides show what happens in the intersection of L with the region.* 89
4.12 *A torus is treated correctly by the propagation rule because no new region can start or disappear at a saddle point.* 90
4.13 *False gaps may appear if the existence of a hole at saddle points is not detected. The refine operator based solely on the identification of stationary points fails if the existence of a hole at a saddle point is hidden by local maxima and minima (mushroom with wormhole). An extended refine operator with the additional information about holes can of course treat the region correctly.* 91
4.14 *Revised version of* **compute-intervals** *computing a locally consistent label for Y given the stationary points E of which all saddle points are labeled with additional information about the existence of holes.* 92
4.15 *The intersection of the projections of two regions onto their common axes results in a region sharing the boundaries of B_1 and B_2 as well as intersections between the boundaries.* 92
4.16 *Concave regions produce stationary points that can be hidden in the projection of the region. P_1, a* below min *of the ternary region, is hidden in the projection of the figure. In addition a new local extremum P_2 appears in the projection, which does not even exist as single point in the ternary region.* 93
4.17 *Propagating intervals through a set of ternary constraints with the same third variable representing a hyperboloid. The point (x_e, y_e, z_e) is a saddle point in Y of the ternary constraint. On the right side, the volume has been projected onto X, Y. The three slices at z_0, z_1, z_e generate the set S that approximates P_{XY}. In this case S covers P_{XY} exactly. If we added a kind of "beak" that possesses no stationary point in Y somewhere inbetween the slides, S would not cover P_{XY} shown by dashed lines exactly. The corresponding volume would resemble a crouched chick.* 94

4.18 *Detailed algorithms of how to find candidates and of the filtering step for a total constraint.* . . . 97

4.19 *Tori in different positions. The planes taken at points between two stationary points show the change in the intersection of L with the torus.* . . . 98

4.20 *The region defined by the inner part of a "mexican hat". Since this figure can be created by rotation, it is symmetric.* . . . 98

4.21 *Three branches meeting at a single stationary point. At least one of them must have a corner at the stationary point.* . . . 99

5.1 *DCSP algorithm for generating minimal solutions.* . . . 110

5.2 *Algorithm transforming a DCSP into a CSP.* . . . 111

5.3 *Transforming a DCSP into a CSP defining only minimal solutions.* . . . 111

5.4 *The first expression reasoning about the existence on the condenser variable can be transformed into a discrete constraint. We suppose here that a condenser can be of type C1 or C2. The second expression links a continuous constraint to the existence of a variable and its translation would require a mixed constraint formulation with NULL values.* . . . 112

5.5 *The problem spaces created by combining two activity constraints of a DCSP. It is assumed here, that the variables of C_1 and C_2 are active. First, the constraint $C_1 \overset{ACT}{\rightarrow} X_1$ is added to the set of initial variables V_I, then $C_2 \overset{ACT}{\rightarrow} X_2$.* . . . 121

5.6 *Dependencies in a DCSP represented as directed acyclic graph. The right hand sides show how problem spaces are combined. At each node only the additional variables and constraints are shown. The resulting problem spaces are the leaf nodes of the trees.* . . . 124

5.7 *General DCSP algorithm for generating minimal problem spaces.* . . . 128

5.8 *Generation of the minimal subspaces P_1 and P_3 adding a single activity constraint $C_i \overset{ACT}{\rightarrow} X_i$ to the problem space $P = \langle Vact, Crel \rangle$.* . . . 128

5.9 *Four problem spaces are created by two activity constraints introducing a cooler and a condenser for the mixer configuration.* . . . 129

5.10 *GDCSP algorithm for generating all minimal problem spaces in design tasks.* 131

5.11 *Pruning of problem spaces due to consistency checks.* . . . 132

5.12 *Function* **with-condition** *generating ensuring local consistency on the solution spaces.* . . . 133

5.13 *Four problem spaces are created by two activity constraints introducing a cooler and a condenser into the mixer configuration. One of the problem spaces, P_1, has an inconsistent solution space due to constraints on the vessel volume. This inconsistency is detected when enforcing local consistency.* . . 133

5.14 *A condition on the number of elements like $f(NbElts) \leq Max$ might generate several feasible intervals.* . . . 135

5.15 *Left: The generation of new elements is restricted by the constraint* $piers.nbPiers \leq 3$. *Three problem spaces are created of which only* P_2, P_3 *are consistent. Right: The generation of new elements is delayed in order to get a chance to reduce the domain of* $piers.nbPiers$ *first. The distance* M *between the two activity constraints can be chosen arbitrarily large.* 136
5.16 *A generic search algorithm.* 137
5.17 *The problem spaces created from* $P_i, i = 2, 3, 4$ *adding the activity constraints* AC_5 *and* AC_6 . *Subsequent search finds three different search spaces according to the values of the vessel type. The search spaces are* S_{i1},S_{i2} *and* S_{i3}. . 139

6.1 *The binary constraint example.* 143
6.2 *The constraints of the train configuration example form cycles in the corresponding hypergraph.* 145
6.3 *The constraints of the mixer configuration example form cycles in the corresponding hypergraph.* 147

B.1 *Point* p_1 *lies on the boundary of region* R *whereas* p_2 *is an interior point of* R. 165

List of Tables

2.1 *Types and subtypes in the mixer example.* 15

2.2 *Mixer configuration formulated as a CSP.* 17

3.1 *A list of operators that define discontinuous functions. The operand r is a real, i,p and q are integers.* 70

5.1 *First table: solutions of problem $\mathcal{P}_1$. Second table: solutions for the minimal model of $\mathcal{P}_1$ corresponding to the minimal solutions obtained by Mittal and Falkenhainer. Variables marked by a hyphen are not active.* 107

6.1 *Solutions for constraint set 1 using various consistency techniques.* 144

6.2 *Variables and their domains used in the train configuration example.* 145

6.3 *Three solutions of the first locally consistent solution space of Figure 7.4.* . . . 148

Chapter 1

Introduction

Everyone designs who devises courses of action aimed at changing existing situations into preferred ones.

[Simon, 1981]

This research has been motivated by a project entitled "Knowledge-based support for conceptual structural design"[1], in which I have been involved since the beginning of my thesis work. Under this general title, a system for conceptual bridge design was developed in collaboration with civil engineers. Constraint satisfaction techniques were integrated into a commercial CAD system, ICAD[2] in order to enhance the capability of the system to find several, qualitatively different solutions to a given design task. Design is concerned with creating an artifact that realizes goals in a given environment. Conceptual design is one of the early design stages, in which important decisions are taken, i.e. necessary parts of the structure are identified and a rough dimensioning is done.

Design problems are challenging because of the inherent complexity, which characterizes the design process. A lacking formalization of the design process itself considerably hampers the use of computers during the elaboration of a solution. Traditional computer-aided-design (CAD) packages are merely concerned with the final stage of design: detail drawing and specification of geometry. Newer packages like ICAD and Concept Modeller[3] also provide tools for a modular knowledge-based representation and thus allow the user to create catalogs of generic components. However, the decision process itself, during which components have to be selected and values chosen, is not sufficiently supported by these tools and is either hard-coded within the knowledge representation or performed by a simplistic demand-driven dependency backtracking scheme. Hence, most designs are still performed with paper and pencil for the computer has not yet been accepted as a support during the decision process. A consequence is that errors that are committed early in the design process are difficult to rectify later and can result in huge additional cost and time requirements.

[1]Swiss National Science Foundation, contract no. 21-39379.93 and 20-45416.95

[2]ICAD , Concentra ©

[3]Concept Modeller, Wisdom Systems ©

Design can be understood in a larger context as a problem solving activity. According to Simon, the way in which design problems are solved is also characteristic of other activities such as diagnosing a patient, organizing a timetable, scheduling flights, and devising a sales plan for a company. Enhancing methods for solving design problems might therefore also prove useful for solving a wider range of real-world tasks.

In the following, the characteristics of a design task are discussed and subtasks that can be automized in the design process are identified. AI-techniques that are suitable for solving these tasks are then proposed.

1.1 The characteristics of design

Commonly, a *design activity* is defined as the "complete specification of design descriptions so as to meet a set of requirements" also called goals [Brown and Chandrasekaran, 1988, Dasgupta, 1991, Coyne et al., 1990, Gero, 1990, Simon, 1981]. The product of a design is called an *artifact*. Typical high-level goals, which an artifact has to satisfy, concern functionality, reliability and performance. When designing a bridge for example, the specification should include beams and supports (called piers) because these are the most basic parts of each bridge. The primary functionality of a bridge is to establish a passage over a an obstacle such as a river, a valley or a road. The bridge structure should satisfy load requirements imposed by daily traffic, laws ensuring its stability and cost criteria imposed by politicians. Design is thus the activity that decides on a set of components and their interconnection in the aim of producing a physically feasible structure that satisfies all the requirements. A design process encompasses at least the following stages:

1. **Specification:** From the initial requirements more detailed requirements are elaborated like cost, stability, forces, kinematics, transport, safety, geometry, production etc. Furthermore, functions and sub-functions are identified, which allow for a first attempt at conceptual solutions.

2. **Conceptual design:** preliminary design forms and layouts are developed and decisions about component shapes and material are made. The result of this stage is a first geometric form and spatial layout.

3. **Detailed design:** detailed drawings are finalized and production documents established.

In contrast to design, configuration tasks are recognized to be well-specified in the sense that a finite set of components (a catalog) is known initially from which the artifact is assembled. The components themselves are not modified nor are there new ones created. In addition, the components can only be connected together in predefined ways [Mittal and Falkenhainer, 1990], which can be described by constraints on discrete variables.

In this thesis, we concentrate on the stage of conceptual design assuming that a specification has already been decided upon and is given as input. This implies that a mapping

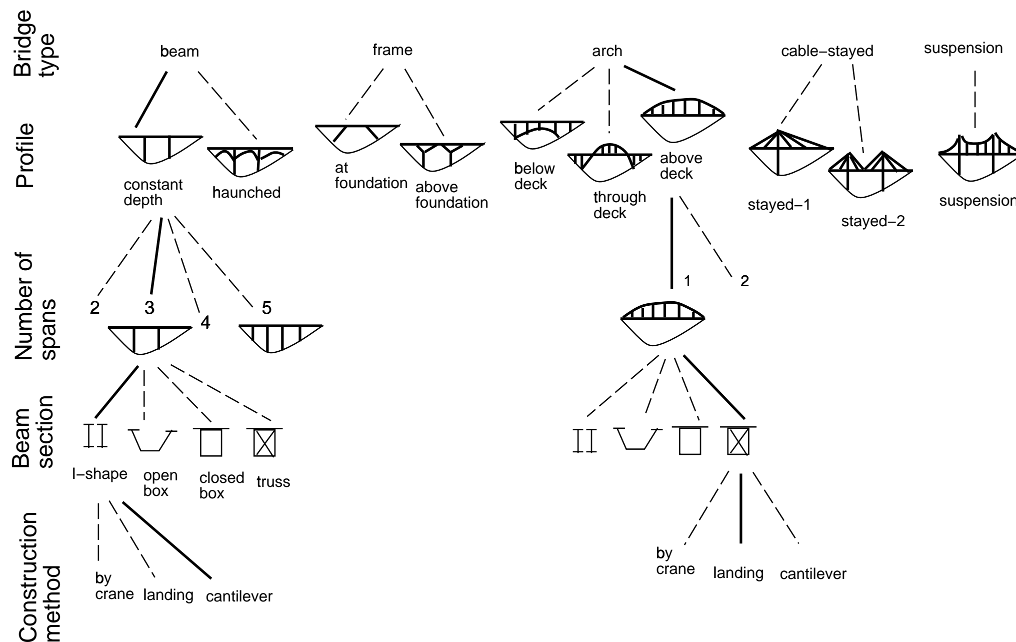


Figure 1.1: *Many alternatives exist in conceptual bridge design. Five important decision points for a bridge design are shown: the type of bridge, profile, number of spans, section and the construction method. Commitments to certain key parameters, for example the bridge type, constrain the design considerably. Given a valley of 150 meters with a maximal height of 30 meters and a river, two alternative solutions are traced through the entire decision process: one choosing a constant-depth beam bridge and the second an arch bridge (indicated by solid lines). Courtesy, Sylvie Boulanger, Steel Structures Institute, EPFL*

between functions and different parts of the structure has already been specified and is available. Design at the conceptual stage can be viewed as a process of constraint specification and satisfaction: The initial set of constraints is defined by the specification outlined at the beginning of the design process. These constraints are defined over variables, which are parameters defining the environment, and parts of the structure. Each variable has as domain a set of possible values. The rest of the process can be seen as the search for a solution that satisfies these constraints. Each decision in the form of a value assignment to a variable in order to satisfy some constraint may impose further constraints on the design. Thus the possible solutions resulting from the overall design description cannot be determined statically but are derived *dynamically.*

A typical example of this dynamic process is bridge design. Given the description of the topology of the environment in which a bridge has to be built, designers have the choice of at least four bridge types: frame, cable-stayed, arch and beam bridge (Figure 1.1). Each type has a different structure constraining further the design: for a beam bridge, for instance, there exists many possibilities of varying the number of piers and their distribution over the length of the bridge. Although we might not be able to generate all different possibilities, our goal is to support the exploration of several alternatives, given certain commitments

of the designer, for example to a bridge type. Another example of dynamic derivation of new variables and constraints is the configuration of industrial mixers. A product that is chemically characterized as dispersion necessitates a condenser whereas, in general, a condenser is optional and not part of the mixer vessel. If a condenser is selected, constraints defined on the condenser become relevant to the current context.

It can be seen in both examples that design usually involves variables of different types: numerical variables (also called continuous variables) like the vessel volume or the pier positions and discrete variables like the number of piers, the bridge type, and the type of mixing task. Hence, constraints defined on these variables are either continuous or discrete or involve both types of variables. Although well-known techniques exist for finding optimal or nearly optimal solutions to a given problem, especially techniques for linear and nonlinear equations 2.7, designers refuse to use them. Optimal solutions with respect to some design criteria like cost, reliability or performance, are often difficult to obtain. First, because not just a single optimum is to be optimized but a set of sometimes conflicting criteria. Second, the utility function, a function measuring such criteria, is in general difficult if not impossible to formalize. How could political reasoning be included in such a utility function ? Consequently, the goal of design is not to find an optimal answer but a "satisficing" one [Simon, 1981], i.e. one which satisfies all constraints. In bridge construction, there might be a conflict between an esthetic distribution of piers and the cost of the bridge structure because placing a pier on a sandy ground would require large foundations. An individual weighting of design criteria explains why different designers will find qualitatively different solutions to exactly the same design problem [Boulanger et al., 1995], [Haroud et al., 1995], [Gelle and Smith, 1996].

The solutions to a design problem have to be synthesized from local information (the constraints). Since such constraints might be interdependent, a partial solution does not always extend to a complete solution. This type of problem is called a *synthesis task*, because solutions have to be synthesized. The reasoning used to solve such tasks is *abductive* in nature: Knowing the functionality of each part of the structure, find an artifact that behaves according to the requirements. Given a set of structural parts S and a set of functions F, the initial knowledge would ideally be described by a set of implications $\{S_1 \to \{F_{11}, \ldots, F_{1m_1}\}, \ldots, S_n \to \{F_{n1}, \ldots, F_{nm_n}\}\}$ with $S_i \in S, F_{ik} \in F$ and the requirements by a subset of functions $F^R \subset F$. The task is to find the set of structural parts that satisfy F^R[4]. A straightforward way to solve this problem is to generate all subsets of S and to test if they generate the function set F^R. Even if a strict enumeration of solutions by assembling local information is theoretically possible, in general, it exceeds available computer capacities and time resources, and is an unrealistic approach. Furthermore, if real-valued parameters are involved, there are an unlimited number of possible valuations. The abductive nature of design problems is well-captured in the model of constraint satisfaction. Resolution algorithms proposed in constraint satisfaction synthesize solutions from local information given in the form of constraints [Freuder, 1978], [Cooper, 1989].

[4]Often, the relationship between a structure and the functions it fulfills is less direct and is expressed by linking the expected behavior derived from the required functionality to the actual behavior of the structure [Gero, 1990].

From the point of view of designers, it is interesting to identify several alternative solutions to a problem or to approximate at least the regions in which solutions can be found. First, a commitment to single values is made in the light of available information and the more complete this information is, the better a choice can be made. Second, even if the comparison of solutions can be difficult, rating alternatives with respect to a set of criteria might indicate what is a good or preferable solution. *Solution regions* appear when the problem is under-constrained; in this case the constraints define entire regions of contiguous feasible values. Especially constraints on continuous variables form feasible regions, which can be represented as shapes in two and as volumes in three dimensions. If the solutions can also be represented as regions, a designer has the additional choice of navigating within them to fix appropriate values. Sometimes designers may be required to change and adapt an existing solution to a similar one. If only a single solution has been computed, the whole search process has to be started from scratch. If, on the contrary, a set of solutions in form of regions is already produced, similar solutions can be found more easily by moving to neighboring points in the same solution region.

1.2 Constraint satisfaction problems

In the previous section, we have seen that proposing sets of solutions might support the designer during the design process. Furthermore, it would be highly desirable to automize the computationally intensive subtask of synthesizing solutions from the given constraints. Once the constraints on the artifact are explicit, any algorithm performing search within the space defined by the constraints can be used to identify solutions. Hence, the next topic is devoted to different types of search methods and to the question of their applicability to solving design tasks.

Typical search methods include linear and nonlinear optimization, genetic programming and tabu search. All of them have in common that they solve a very specific instance of problem, for example, continuous nonlinear equations, linear or integer equations. Most of them are based on optimization techniques and thus find a single solution according to the given utility function. It follows that these methods do not satisfy the requirements for design problems as previously discussed.

A recent method, which has made its appearance in Artificial Intelligence, is *Constraint Satisfaction*. A constraint satisfaction problem (CSP) is defined by a set of variables, each variable having a domain of possible values, and a set of constraints over these variables, which specify allowed values that the variables may take on in a solution. A solution is a consistent assignment, i.e. a set of values from the variable domains of each variable such that all constraints are satisfied. The goal is in general to find one, some or all consistent assignments to a CSP. Since graph-coloring is an instance of a CSP, finding a solution to the constraint satisfaction problem is NP-complete [Mackworth and Freuder, 1985]. Enumerating all possible value combinations and testing them against the constraints is thus computationally intractable for larger instances of CSPs.

In Artificial Intelligence, one approach to solving CSPs consists of preprocessing tech-

niques called *consistency* techniques. The idea is to remove inconsistent value combinations from the set of possible solutions (the cross-product of all variable domains), and to propagate the effects of such a removal through the constraint network. This preprocessing avoids that inconsistent value combinations are revisited during search. In general, k-consistency removes inconsistencies that involve all subsets of size k of n variables. Global consistency allows the implicit representation of all possible solutions and guarantees backtrack-free search. Without making any assumptions about the form of the constraints, ensuring global consistency is exponential in the number of variables. The other extreme, 2-consistency (also called local consistency), removes values that are part of an inconsistent value pair. This method has the advantage that the inconsistent values found can be removed from variable domains directly and that polynomial-time algorithms exist for discrete CSPs.

Furthermore, local consistency techniques are of use during search: in forward checking, the currently instantiated variable is propagated to the neighboring variables connected by a constraint. The more effectively the label of the neighboring variables can be refined by local consistency the faster backtrack search can decide if a solution exists or not. Search in continuous domains can be very costly because the number of potential points to be visited depends on the allowed precision as well as the size of the domain.

It has to be emphasized that local propagation methods like local consistency are more efficient for constraints that are formulated locally; i.e. with a small arity. In fact, 2 out of k variable pairs, where k is the maximal arity of the constraints, have to be considered for propagation. In addition, results may lose accuracy when computed by an algorithm that relies on a specific input form of the constraints, like for example, restrictions to ternary or basic constraints. CSPs containing global constraints may profit from more specialized methods ([Régin, 1996]).

Consistency algorithms are complete by definition, i.e. they never remove solutions. Completeness is motivated by our applications in conceptual design, where in a first stage it is important to localize spaces containing all solutions. Equally important, local consistency techniques can only be applied during search because they are complete. Local consistency algorithms should also be locally sound such that no inconsistencies are added and that all remaining inconsistencies are of higher degree.

1.3 Topics of this research

1.3.1 Continuous consistency techniques

For many years, research in Constraint Satisfaction has concentrated on developing consistency techniques for binary discrete constraint systems minimizing the effort in the subsequent search for solutions as well as on different search strategies. Early attempts at applying consistency techniques to continuous constraints have been discouraging mainly due to an oversimplified refinement step but also because of numerical error propagation and the inherent complexity of numerical systems.[5] Only in recent years, approaches based

[5] There exist problems with three constraints and three variables that are classified as very hard by Numerical Analysts. On the other hand, problems with thousands of variables involved in linear equalities

on interval propagation have raised new interest in consistency techniques for continuous domains. However, most current resolution techniques for continuous variables still rely heavily on domain splitting because the consistency methods they employ often have little effect. Apart from the method proposed by [Faltings, 1994], we know of no local consistency method achieving 2-consistency for continuous domains. Since this method is restricted to binary constraints, our first thesis topic is to extend this algorithm to constraints of higher arity.

Topic 1: Enhancing local consistency techniques for continuous constraints.

Typical properties of such algorithms are:

Solution regions. We assume that the constraints define feasible spaces as sets of contiguous solutions[6] rather than point-solutions. This implies that our local consistency algorithm for continuous constraints is designed for the treatment of regions in continuous space. For fully constrained numeric systems, which consist of n variables and n constraints, there are dedicated algorithms such as Numerica [Van Hentenryck, 1997].

Exactness of solutions. The implementation of any method dealing with real variables poses numerical problems due to rounding errors. This is not characteristic of our algorithm in particular. We do not treat this problem in our thesis, but we assume the existence of safe approximation methods like interval analysis or rounding procedures for computing real values.

Generality and reliability. Two approaches exist in the context of algorithms in the continuous domain: methods that rely on properties of the solution region defined by the constraints and those relying on syntactic properties such as the type of constraint (linear, nonlinear, equation, inequation). In addition, some of them require a preconditioning step to be carried out. Our algorithm belongs to the first category relying on local properties of the feasible constraint regions. Under the condition that these properties can be determined, our algorithm is able to compute locally consistent labellings. Hence, we have to distinguish between the conceptual framework, for which proofs are presented, and the implementational issues of how to find point sets verifying these local properties.

1.3.2 Consistency techniques for mixed constraints

Most techniques in constraint satisfaction, as in other fields, are restricted to a uniform variable and constraint type. Resolution algorithms exist for variables with continuous domains and others for discrete domains but no algorithm integrates the consistency approach

are solved quite easily by dedicated methods.

[6]This assumption is influenced by our application domain, which is design. In design, relations between parameters often are of empirical nature and cannot be determined exactly. Such relations are typically specified by equations of the form $X = Y \pm P\%$ with $P \in \mathbb{N}$, which can be rewritten as a set of inequalities.

for both types of variables.

> Topic 2:
> Enhancing local consistency techniques for mixed constraints defined on continuous and discrete variables and integrating all local consistency techniques into the same framework.

Typical properties of such techniques are:

Constraint representation. The representation of constraints plays an important role in the design of consistency and search algorithms. There exists a fundamental difference between discrete and continuous constraints:

1. Discrete constraints are based on a representation of discrete value combinations and either enumerate allowed tuples explicitly or express them in a logic formula.
2. Continuous constraints use an implicit representation of allowed points in continuous space in the form of an analytical representation.

This inherent difference leads to the design of different consistency techniques dedicated to each constraint type. Another possibility is to discretize the feasible region defined by continuous constraints [Sam-Haroud, 1995]. In this thesis, we discuss the advantages and drawbacks of both representations.

Integration of different local consistency methods. Consistency algorithms for constraints on discrete and continuous variables should be integrated in a manner transparent to the user and without creating an overhead due to switching between different constraint solvers.

1.3.3 Dynamic constraint satisfaction

In order to account for dynamic aspects like variable addition, a *dynamic constraint satisfaction* (DCSP) approach has been developed by [Mittal and Falkenhainer, 1990]. This framework integrates variable creation and the standard CSP paradigm. The original algorithm, however, has been developed only for discrete constraints.

> Topic 3:
> Generate solutions of a dynamic constraint satisfaction problem with continuous and discrete constraints.

Some properties to be respected are:

Generic constraints. It should be possible to formulate constraints over variable types without referring to specific variable instances. From such generic constraints instances are then generated for each combination of variable instances corresponding to the types over which the generic constraint is specified. This is a natural requirement for dynamic

problems.

Restricted variable generation. The generation of solution spaces in a DCSP is driven by the conditional generation of new variables. The number of variables must be bounded a priori otherwise the generation process cannot be guaranteed to stop for any possible input problem.

1.4 Contributions

The main contributions of this thesis are:

1. **Local consistency techniques for continuous constraints**. Local consistency for binary constraints from [Faltings, 1994] is extended in this thesis to ternary constraints. Since every numeric CSP can be transformed into a system of ternary constraints, local consistency can be enforced on general numeric CSPs. The technique again makes use of total constraints, formed of all constraints defined in the same space. All constraints in a total constraint are considered simultaneously in one propagation step and this enables us to take into account intersections between constraints.

   A general correctness proof is given for this method. This proof is based on topological properties of the solution space and is generic in the sense that it does not restrict itself to a specific type of continuous constraint. It also identifies a limitation: feasible regions containing holes require additional local information in order to be treated correctly by this method.

2. **Integrating discrete and continuous propagation into the same framework**. A general framework for computing local consistency in mixed, i.e. discrete and continuous, CSPs is proposed. It is based on specific refine operators for each type of constraint embedded into the general propagation algorithm. A refine operator computes the consistent values for a given variable pair. A clear advantage of this approach is the neat integration of mixed constraints, which are defined on discrete and continuous variables simultaneously.

3. **Generation of locally consistent solution spaces**. A generation method for finding all locally consistent solution spaces in a dynamically formulated CSP is proposed. It is based on a new constraint-driven algorithm that first generates problem spaces (static constraint problems) in a mixed DCSP. Local consistency techniques are used to prune inconsistent problem spaces as early as possible. An analysis of the generation mechanism leads to an identification of those constraints that are responsible for the combinatorial size of the number of problem spaces.

Our approach is demonstrated on examples in design and configuration.

1.5 Guide to the thesis

In this chapter, we have given the motivation behind the research that has led to this thesis. The results consist of two parts: the generation of problem spaces in DCSPs, and local consistency techniques especially for continuous constraints. At a first glance, they seem to be disconnected. The link is made when "marrying" both concepts to achieve the first aim: the generation of solution spaces in a DCSP. This thesis can therefore be read following the dependency links between the chapters as shown in Figure 1.2. The parts on local consistency and generation of problem spaces can be read independently from each other. The contents of this thesis are:

Chapter 2: Definitions and concepts

Gives a short overview of the basic definitions of constraint satisfaction, consistency techniques and search strategies. A small example on configuration, which is reused throughout the thesis, is introduced in this chapter and the definitions are illustrated using the example.

Chapter 3: Local consistency techniques

Describes the integration of discrete and continuous consistency techniques into a fix-point algorithm of type AC-3. Approximations for mixed constraints are also described. The section on consistency techniques in numeric binary CSPs is a reminder of the concepts of local consistency for numeric constraints presented in [Faltings, 1994] and serves as an introduction for chapter 4. This chapter also discusses an implementation of local consistency for binary continuous constraints.

Chapter 4: Local consistency for ternary numeric constraints

Extends results on consistency techniques in numeric binary CSPs to ternary numeric systems thus providing a basis for computing locally consistent labellings for general numeric CSPs. The correctness of this extension is proven and limitations are discussed.

Chapter 5: Systematic generation of problem spaces

Generalizes dynamic constraint satisfaction to problems involving continuous variables. A new method for problem space generation in such a framework is presented, which finds all sets of CSPs in a DCSP. Locally consistent solution spaces result from the systematic generation algorithm. These spaces can be searched for single solutions employing the local consistency methods described in chapter 3.

Chapter 6: Results

Several examples from the field of design and configuration are described, solved in the proposed framework and their results analyzed.

Chapter 7: Conclusions

Reviews contributions of this thesis on local consistency for mixed constraint systems and the generation scheme proposed for dynamic CSPs. Open problems are discussed and future research directions proposed.

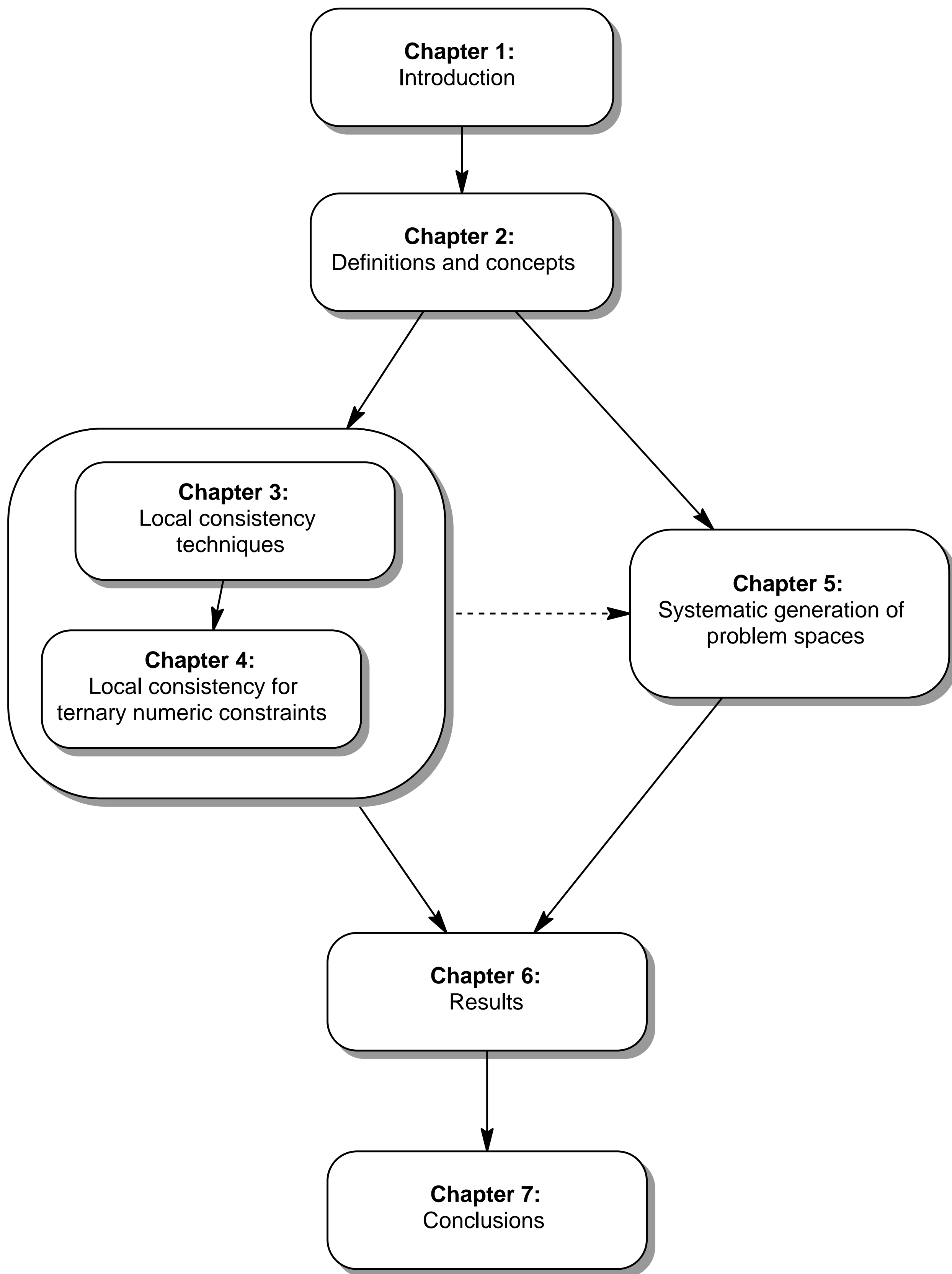


Figure 1.2: *Direct dependencies between chapters in this thesis.*

Chapter 2

Definitions and Concepts

"No, no; I never guess. It is a shocking habit - destructive to the logical faculty."

Sherlock Holmes in *The Sign of Four*, The Strand Magazine (1890)

2.1 Introduction

In this chapter, we will give a short introduction in the area of constraint satisfaction evoking important concepts and definitions in this domain. This is not intended to be a general review but a guide of definitions relevant to our research. First, an example from the domain of configuration is presented, which will be referred to during the entire thesis. We then introduce the notion of a constraint satisfaction problem (CSP) and show how this particular example is modeled as a CSP. Finally, some of the consistency techniques and search strategies that are important for our work are reviewed.

2.2 Notational Conventions

Throughout this work, we use upper case letters for variables, sets and vectors and lower case letters for single values. For example, X, Y, Z or X_i are variables and x_{j_k} is a particular value for variable X_j. We will use the notation $[a, b]$ with $a \leq b, a, b \in \mathbb{R}$ for a closed, (a, b) for an open interval and $[a, b)$ for an interval open to the right.

2.3 Example: Configuration of industrial mixers

This example consists of configuring mixers used in industry. It has been adapted from the F5 example described in [van Velzen, 1993]. We choose two main components for representing the overall task: the mixer, which describes the artifact itself and the mixing task, which specifies requirements induced by the product to be mixed. Industrial mixers can be classified into three types:

- reactors, which are used in chemical experiments where mixing is used to produce reactions

- storage tanks, which keep a product mixed during storage (important for products consisting of several phases)
- conventional mixers, which range from kitchen mixers to a concrete mixing machine on a construction site

The second important component type is the mixing task. It defines the characteristics of the product to be mixed. A chemical product has one or more phases and each phase is either solid, liquid or gas. Depending on the combination of phases, a mixing task is categorized as

- suspension: a mixture in which small particles are distributed throughout a less dense liquid (or gas),
- blending : blend = mix (sorts or grades of spirits, tea, tobacco, etc.) so as to obtain a certain quality,
- dispersion: a mixture of one substance dispersed throughout another, continuous one; an emulsion, aerosol, etc.,
- entrainment: semi-liquid mixture of a pulverized solid or fine particles with a liquid; cement, etc.

The component types involved in the configuration of a mixer are shown in the "part-of" hierarchy in Figure 2.1. Links between components represent the relation *part-of*. Bottom, for example, is a part of Vessel. Dotted links represent *optional* components in the sense that a mixer does not necessarily include a condenser. Each component type is described by its properties, which are either basic or composed: a mixer vessel, for example, consists of the subcomponents Condenser and Cooler and the basic properties volume, height and diameter. All component types with their properties are listed in the appendix A.1. We use the dot notation to refer to properties: $V.condenser$ denotes the subcomponent Condenser of type Vessel and $MT.slurry\ pressure$ the property slurry-pressure of the component type Mixing Task. For simplicity component types are abbreviated by their initials. Some component types are divided into subtypes, which are shaded in Figure 2.1. The component type Mixer, for example, has the subtypes Storage Tank, Mixer and Reactor, and Mixing Task is divided into Dispersion, Suspension, Blending, Entrainment (Table 2.1). This relation between component types is termed *of-subtype* and will be written $V \overset{ST}{=} elliptical$. The description of component type Vessel for example depends on its subtypes: if Vessel is of subtype Elliptical, the small radius and the bottom area of the elliptical surface have to be specified as properties of Vessel because they are required in the computation of the vessel volume. Eventually, the conditions under which optional components like the condenser or the cooler are part of the mixer vessel have to be specified. Such constraints are formulated as logical implications (shown by arrows in Figure 2.1):

- The existence of the condenser depends on the vessel volume:
  $V.volume \geq 150 \rightarrow \exists\ C$
- The existence of a cooler depends on the mixer type:
  $M \overset{ST}{=} reactor \rightarrow \exists\ Co$

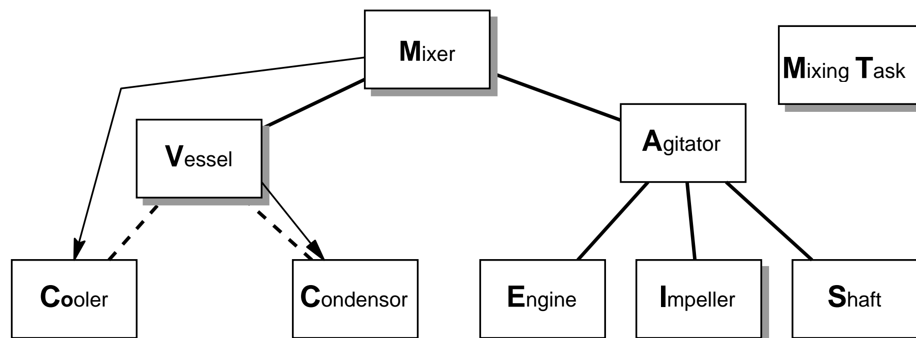


Figure 2.1: *The physical decomposition hierarchy of an industrial mixer is shown together with functional dependencies indicated by arrows.*

Component type	**Abbrev.**	**Subtypes**
Mixer	M	Storage Tank, Mixer, Reactor
Mixing task	MT	Suspension, Entrainment, Blending, Dispersion
Impeller	I	Axial-turbine, Helical-ribbon, Propeller, Silverson-highshare, Dented-disk, Radial-turbine, Scaba, Anchor-stirrer
Vessel	V	Elliptical, Hemispherical, Cylindrical
Elliptical	El	-
Cooler	Co	-
Condenser	C	-
Agitator	A	-
Engine	E	-
Impeller	I	-
Shaft	S	-

Table 2.1: *Types and subtypes in the mixer example.*

Some typical relations between components and properties of a mixer are:

- a relation R_1 between the pressure of slurry particles of the product and the shape of the vessel

$$\begin{aligned} (MT.slurry\,pressure = high \quad &\wedge \quad (V \stackrel{ST}{=} hemispherical \ \vee V \stackrel{ST}{=} elliptical)) \vee \\ (MT.slurry\,pressure = low \quad &\wedge \quad V \stackrel{ST}{=} cylindrical) \end{aligned}$$

- a relation R_2 between the power, position, diameter, slurry density of the product and the revolutions per second of the impeller

$$I.power \quad = \quad MT.slurry\,density * I.position * I.rps^3 * I.diameter^5$$

- a relation R_3 between the vessel type and the volume of the vessel

$$
\begin{array}{lll}
(V \stackrel{ST}{=} hemispherical & \wedge & V.volume = \frac{1}{12} * \pi * V.diameter^3 + \\
 & & \frac{1}{4} * \pi * V.diameter^2 * (V.height - \frac{1}{2} * V.diameter)) \vee \\
(V \stackrel{ST}{=} cylindrical & \wedge & V.volume = \frac{1}{4} * \pi * V.diameter^2 * V.height) \vee \\
(V \stackrel{ST}{=} elliptical & \wedge & V.volume = \pi * El.sradius * \frac{1}{2} * V.diameter)
\end{array}
$$

In the following section, we show how this example can be modeled as a constraint satisfaction problem, in which variables are components or properties of components and the relations are constraints on the variables.

2.4 Constraint satisfaction problems

A constraint satisfaction problem (CSP) involves a set of variables with their associated domains and a set of constraints defined on these variables. One of the first definitions of binary constraint networks goes back to [Montanari, 1974]. Since we also consider constraints of higher arity in our work, we extend this definition:

Definition 2.1 (Constraint satisfaction problem) *A* constraint satisfaction problem *$\mathcal{P}$ is a tuple $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ that consists of*

1. *a finite set of variables $\mathcal{V} = \{X_1, \ldots, X_n\}$ with their domains $\mathcal{D} = \{D_1, \ldots, D_n\}$ such that each variable $X_i \in \mathcal{V}$ has an associated domain of possible values $D_i \in \mathcal{D}$*
2. *a finite set of relations $\mathcal{C}$, called* constraints, *each constraint $C_{X_{i_1},\ldots,X_{i_j}} \in \mathcal{C}$ defined on a subset of variables $Vars(C) = \{X_{i_1} \ldots X_{i_j}\}$ with $Vars(C) \subseteq \mathcal{V}$. Each constraint $C_{X_{i_1},\ldots,X_{i_j}}$ specifies a subset of the Cartesian product of the variable domains as allowed value combinations: $C_{X_{i_1},\ldots,X_{i_j}} \subseteq D_{i_1} \times \ldots \times D_{i_j}$.*

A k-ary CSP is a CSP with $\mathcal{C}$ restricted to j-ary constraints, $j \leq k$.

The goal of any search algorithm is to find at least one valid assignment, a solution, or to show that there is no solution. For a configuration and design task modeled as a CSP, it is in general more interesting to know a set of solutions in order to be able to compare their quality. An instance of a CSP is $\mathcal{P}_1 = \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ defined on the mixer example with:

1. A set of variables $\mathcal{V} = \{X_1, \ldots, X_{11}\}$ with their domains (Table 2.2). Each variable stands for a component instance of a given type or for a property thereof. Variable X_2, for example, represents a component instance for the vessel. The domain of such a variable is either a set of subtypes if the variable represents a component or a basic domain like a set of discrete values or intervals.

2. A set of constraints $\mathcal{C} = \{C_{X_1,X_2}, C_{X_2,\ldots,X_6}, C_{X_7,\ldots,X_{11}}\}$ defined in the introductory example with

$$
\begin{array}{l}
C_{X_1,X_2} == R_1 \\
C_{X_7,\ldots,X_{11}} == R_2 \\
C_{X_2,\ldots,X_6} == R_3
\end{array}
$$

Variable	**Domain**
$X_1 = slurry\,pressure$	$D_1 = \{low, high\}$
$X_2 =$ instance of V	$D_2 = \{elliptical, cylindrical, hemispherical\}$
$X_3 = volume$	$D_3 = [0, 1000]$
$X_4 = diameter$	$D_4 = [0, 1000]$
$X_5 = height$	$D_5 = [0, 1000]$
$X_6 = sradius$	$D_6 = [0, 1000]$
$X_7 = position$	$D_7 = [0, 5]$
$X_8 = diameter$	$D_8 = [0, 1000]$
$X_9 = rps$	$D_9 = [0, 1000]$
$X_{10} = power$	$D_{10} = [0, 1000]$
$X_{11} = slurry\,density$	$D_{11} = [1, 2000]$

Table 2.2: *Mixer configuration formulated as a CSP.*

A constraint can be represented either intensionally by specifying all allowed value combinations or extensionally by a logical predicate. In order to define what a solution is, we first need the notion of an assignment consistent with a given constraint defined as predicate.

Definition 2.2 *A constraint* $C_{X_1,\ldots,X_k} \in \mathcal{C}$ *is* satisfied *by an assignment* $\{X_1 = x_{1_j}, \ldots, X_k = x_{k_j}\}$ *if and only if* $C_{X_1,\ldots,X_k}(x_{1_j}, \ldots, x_{k_j})$ *is true.*

When the constraint is defined as set of allowed tuples, this is equivalent to $(x_{1_j}, \ldots, x_{k_j}) \in C_{X_1,\ldots,X_k}$. A solution of a CSP is a consistent assignment to all variables such that all the constraints are satisfied.

Definition 2.3 (Solution of a CSP) *An instantiation of the variables in* $\mathcal{V}$ *is a set* $\{X_1 = x_{1_j}, \ldots, X_n = x_{n_j}\}$ *such that each variable* X_i *is assigned a value* $x_{i_j} \in D_i$, $j = 1, \ldots, |D_i|$. *A* solution *of a CSP* $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ *is a consistent instantiation* $\{X_1 = x_{1_j}, \ldots, X_n = x_{n_j}\}$ *for the variables* $X_1, \ldots, X_n \in \mathcal{V}$ *such that all constraints in* $\mathcal{C}$ *are satisfied.*

One solution of $\mathcal{P}_1$ is the assignment $\{slurry\,pressure = low$, $V = cylindrical$, $volume = 488.47$, $height = 10.67$, $diameter = 7.63$, $power = 1250$, $I.rps = 1.13$, $diameter = 2.51$, $position = 5$, $sradius = 0\}$. Note that *sradius* may take any value in its domain because it is not constrained by $C_{X_2,\ldots,X_6}$.

We can distinguish between different classes of constraint problems depending on what kind of constraints the problem is restricted to. A *discrete* constraint satisfaction problem restricts the variable domains to a set of discrete values $\{x_{i_1}, \ldots, x_{i_m}\} \in D_i$. For design applications, discrete domains are in general finite[1].

[1] One could imagine that a variable domain might be very large or even infinite. When configuring a computer network, for example, variables are network nodes and their domains all possible routes from a node to all other nodes in the network. A discrete constraint defined on such a variable should then be given by a predicate. Instead of being explicitly enumerated, the values of a domain would then be accessible by a *next* function, which returns the next value of an ordered domain

Definition 2.4 (Discrete constraint) *A* discrete *constraint $C_{X_1,\ldots,X_k} \in \mathcal{C}$ defines a set of allowed value tuples $(x_{1_j}, \ldots, x_{k_j})$ either explicitly or by a predicate such that $(x_{1_j}, \ldots, x_{k_j}) \in D_1 \times \ldots \times D_k$.*

The values of a tuple in a discrete constraint $C_{X_1,\ldots,X_k}$ are given in the order of the variables $X_1, \ldots, X_k$ of the constraint. For a given tuple t, the value of variable X_i in t is designated by $t[X_i]$. The projection of a tuple onto the variables $X_1, \ldots, X_i$ will be written $t[X_1, \ldots, X_i]$. An example of a discrete constraint is C_{X_1,X_2}. It can also be written as a set of tuples:

$$\begin{aligned} C_{X_1,X_2} := \; & \{(high\ hemispherical) \\ & (high\ elliptical) \\ & (low\ cylindrical)\} \end{aligned}$$

In a *numeric* constraint satisfaction problem, the variable domains are defined over the reals. A continuous constraint is any relation over the real domain.

Definition 2.5 (Continuous constraint) *A k-ary* continuous constraint*, also called a* numeric constraint*, $C_{X_1,\ldots,X_k} \in \mathcal{C}$ is a relation $\circledcirc$ over $\mathbb{R}^k$ with $\circledcirc \in \{=, \geq, \leq\}$. A continuous constraint has the syntax $E \circledcirc 0$ where E is a term built from constants, variables and operations $\{+, -, /, *, exp, log, \ldots\}$ over the reals.*

An example of a continuous constraint is $C_{X_7,\ldots,X_{11}}$:

$$I.power = MT.slurry\ density * I.position * I.rps^3 * I.diameter^5$$

In practical applications, also *mixed* constraints defining relations over discrete *and* continuous variables appear. We therefore extend definitions to include mixed, continuous - discrete, constraints.

Definition 2.6 (Mixed constraint, Gelle) *A* mixed constraint *$C_{X_1,\ldots,X_m} \in \mathcal{C}$ is a relation defined by a set $\{C^{Dis_i} \wedge C^{Con_i}\}$ where the first constraint is a discrete constraint $C^{Dis_i}_{X_1,\ldots,X_j}$ defined on the discrete variables $X_1, \ldots, X_j$ and the second a continuous constraint $C^{Con_i}_{X_k,\ldots,X_m}$ defined on a set of continuous variables $X_k, \ldots, X_m$.*

In the mixer example, there is such a relation between the vessel type and the volume of the vessel. Each tuple simply represents the conjunction of discrete with continuous values:

$$\begin{aligned} C_{X_2,\ldots,X_6} := \; & \{(hemispherical \quad volume = 1/12 * \pi * diameter^3 + \\ & 1/4 * \pi * diameter^2 * (height - 1/2 * diameter) \\ & (cylindrical \quad volume = 1/4 * \pi * diameter^2 * height) \\ & (elliptical \quad volume = \pi * sradius * 1/2 * diameter)\} \end{aligned}$$

A fundamental difference between discrete and continuous constraints appears in the continuous part of this constraint. While, in the discrete domain, several constraints defined on the same variables can be redefined as a unique constraint by simply intersecting their tuple sets, this is not possible for continuous constraints. The reason is that there is a

unique way of combining discrete variables by specifying tuples allowed or not allowed. A continuous constraint over k variables, however, consists of a combination of operators like $\{+, -, /, exp, \ldots\}$. It follows that there are many possibilities of defining constraints over the same set of variables.

2.5 Consistency techniques

[Waltz, 1975] was one of the first to use a filtering for obtaining partial consistency of data in image processing. The algorithm defined by Waltz loops through the constraints and for each individual constraint the value set of each variable is refined. [Mackworth, 1977a] and [Montanari, 1974] generalized this concept to any kind of discrete data related by binary constraints. They define arc-consistency as consistency between each pair of variables and path-consistency as consistency between triples of variables satisfying all binary constraints. These consistency techniques do not solve the constraint satisfaction problem completely but they eliminate local inconsistencies, which cannot be part of any solution. Since solving a general CSP is NP-complete, such techniques are useful to remove inconsistencies before the actual backtrack search starts. Arc and path-consistency have been shown to be polynomial time algorithms [Mackworth and Freuder, 1985]. Analogously, [Freuder, 1978] defined k-consistency as consistency between k variables satisfying the constraints. He also provided sufficient conditions for backtrack-free and backtrack-bounded search [Freuder, 1982b], [Freuder, 1982a]. Since then, much research has concentrated on refining these discrete algorithms, especially arc- and path-consistency, and finding lower bounds of time and space complexity [Mackworth and Freuder, 1985], [Bessière, 1994], [Van Hentenryck et al., 1992], [Mohr and Henderson, 1986].

Consistency techniques require a compact representation of the effective combinations of consistent values. Such combinations of values are called *labels*.

Definition 2.7 (k-th order labelling) *A* k-th order labelling *of a constraint satisfaction problem is a set of labels $L_{X_1,\ldots,X_k}$ for every combination of k variables $X_1, \ldots, X_k$, where each label designates a subset of value combinations from $D_1 \times \ldots \times D_k$.*

In discrete CSPs k-th order labellings can simply be represented by value tuples of length k. In continuous CSPs, the k-th order labellings represent arbitrary topological regions in $\mathbb{R}^k$.

Definition 2.8 (k-consistency; [Freuder, 1978]) *Given a set of consistent values for $k-1$ variables so that all constraints are satisfied, a CSP is* k-consistent *if and only if, for any k^{th} variable, there exists a value such that all constraints are satisfied for the k variables.*
Strong k-consistency requires j-consistency to hold for all $j \leq k$.

If a CSP is k-consistent, this does not imply automatically that it is also j-consistent with $j < k$. In Figure 2.2 a), a coloring problem is represented taken from [Freuder, 1982b]. Let X, Y, Z be three nodes where Y can be colored red and blue but X and Z can only

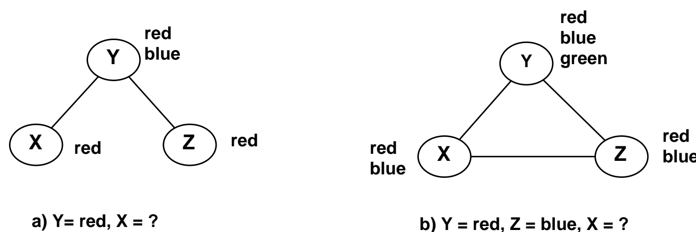


Figure 2.2: *a) An example of a CSP that is 3-consistent but not 2-consistent, b) a CSP that is arc-consistent but not 3-consistent. In both examples, backtracking might be necessary in order to find a solution.*

take on red. The constraints $X \neq Y$ and $Y \neq Z$ are the links in the constraint graph. The resulting CSP is 3-consistent but not 2-consistent. In fact, taken any consistent assignment of two nodes, there exists a value for the third node. However, if we assign color red to node Y there exist no consistent assignment for node X. On the other hand, if each node is allowed colors red and blue, the resulting problem is 2-consistent but not 3-consistent. If node X is colored blue and node Z red, there exist no color for node Y.

In order to represent the combinations of consistent values, k-consistency makes use of a (k-1)-th order labelling. In fact, if there exists no k-th value such that the combination of this value with a given tuple of k-1 values is consistent, the tuple of k-1 values has to be removed. A labelling is *globally consistent* if and only if every value in every label occurs in at least one complete solution. In some cases, a globally consistent labelling or an approximation thereof can be computed efficiently [van Beek, 1992], [Sam-Haroud, 1995]. In general, global consistency is intractable both in computation time and in complexity of representing the final solution. An alternative that is more efficient than global consistency is *local* consistency, which we use here as a generic term for 2-consistency and different approximations of 2-consistency. 2-consistency applies to first-order labellings. Sets of values for individual variables can be represented efficiently by collections of *intervals* in the continuous as well as in the discrete domain. Arc-consistency is strong 2-consistency:

Definition 2.9 (arc-consistency; [Mackworth, 1977a]) *A binary CSP is* arc-consistent *if for any pair of variables X_i,X_j and for any value $x_{i_k} \in L_i$ of X_i there exists a value $x_{j_k} \in L_j$ for X_j so that all constraints are satisfied and $L_m \subseteq D_m$:*

$$\forall X_i, X_j, x_{i_k} \in L_i \; \exists x_{j_k} \in L_j \;\; C_{X_i,X_j}(x_{i_k}, x_{j_k})$$

A *refine operator*[2] implements one step of making the label L_i of variable X_i arc-consistent with the label L_j of variable X_j. Local consistency is especially interesting in constraint problems whose associated graph has no cycles (see appendix D for definitions related to graph theory). An arc-consistent labelling of a constraint problem forming a tree is guaranteed to possess a backtrack-free search order. This means that an arc-consistent labelling guarantees that all variables can be assigned without backtracking when the constraint tree

[2]In the Constraint Logic Programming literature, this operator is often called *narrowing operator*.

is traversed in preorder [Freuder, 1982b]. Another result proven in [Faltings, 1994] is that, in a tree-structured constraint graph, the refine operator only has to be applied a number of times linear in the size of the node set in order to make the labelling arc-consistent. If the graph has cycles, a cycle-cutset decomposition method as described in [Dechter, 1990] can be applied to cut the cycles. A cycle-cutset is a set of nodes that render the constraint graph cycle-free when removed. By instantiating variables of the cycle-cutset and computing local consistency on the remaining constraint tree, the upper-bound complexity of the problem is essentially dominated by the complexity of finding a consistent assignment of the cycle-cutset. Consider again the coloring problem on a complete graph of three nodes in Figure 2.2 b). Each node can be colored in red or blue and node Y additionally in green. The problem is already arc-consistent but if Y is assigned blue or red first, search has to backtrack in order to find a solution. In the cycle-cutset method, a cutset X, for example, is chosen and instantiated to red. The resulting constraint problem is made arc-consistent, which directly yields a solution: $\{X = red, Y = green, Z = blue\}$. The same process can be repeated for $X = blue$. Finding a minimal cycle-cutset is again NP-complete. Dechter proposes instead to embed a fast arc-consistency algorithm on trees into the backtracking scheme. This hybrid algorithm simply keeps track of the changing connectivity of the variables during instantiation until the remaining uninstantiated variables form a tree. It then calls the arc-consistency algorithm.

Locally consistent solutions will be represented by first order labellings. They define spaces in which the solutions must be located:

Definition 2.10 (Problem space, Solution space) *We call a CSP consisting of a set of constraints and a set of variables a* problem space*. Its* exact solution space *is represented by a set globally consistent labellings. A* locally consistent *labelling approximates the solution space of the CSP.* [3]

Once locally consistent solution spaces of a problem have been identified, a single solution, i.e. a consistent assignment of all variables, has to be found within the labelling. This topic is addressed in the next section.

2.6 Search strategies for solving discrete CSPs

Two prominent branches exist in search: Lookahead strategies, which establish different levels of local consistency during search in order to reduce the search space and information-gathering strategies, which analyze failures made during search and learn from them [Kumar, 1992], [Tsang, 1993]. After explaining the general search scheme, we will discuss different search strategies such as forward checking or maintaining arc-consistency in detail.

[3] Whenever we use *solution space* and do not specify its degree of consistency it is to be understood as locally consistent.

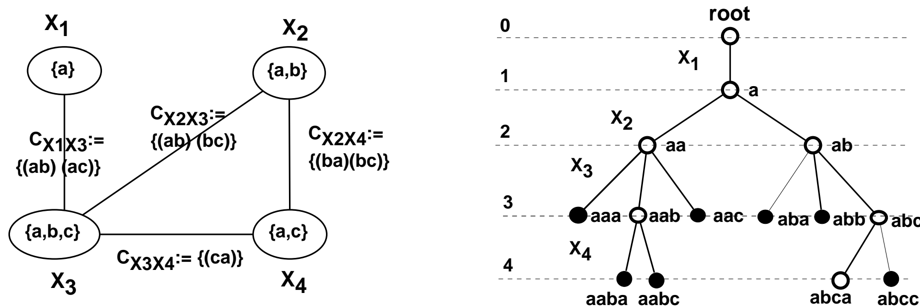


Figure 2.3: *Search tree solving a small CSP by backtracking. The black nodes have been pruned from the search space. Example taken from [Kondrak, 1994]*

2.6.1 General search scheme

A search space can be represented as a tree, in which each node represents a particular state of the search. Edges linking the nodes are transitions between two states. In general, CSPs are solved by instantiating variables and testing partial instantiations against the constraints. These tests are called *consistency checks.* A node in the search tree of a CSP corresponds therefore to a partial instantiation of the variables and an edge between two nodes represents the choice of a value for the next variable. The root node corresponds to the empty tuple. At a given node, a next variable is selected for instantiation from a given order of the variables (order X_1, X_2, X_3, X_4 in Figure 2.3) and a new node is created as successor for each value in the domain of this variable. The variable that is instantiated at some level is called the *current variable*, any variable that is not yet instantiated is a *future variable* and variables already instantiated are called *past variables.* Starting from the root node at level 0, each new level corresponds to the instantiation of a particular variable and the tuples grow from size 0 to size n. If n is the number of variables in the CSP, the nodes at level n are called *leaf nodes.* They represent full instantiations of the variables in the CSP. If a node is reached where the tuple of values becomes inconsistent, it is clear that no future choices will lead to a solution and the node can be pruned. When such a pruning is applied, the leaf nodes represent consistent instantiations, i.e. solutions. In Figure 2.3, node (aaa) can be pruned and its sub-branch neglected because it does not satisfy the constraint between variables X_1 and X_3.

The simplest search paradigm is *generate-and-test* (GT). In this method, the complete set of leaf nodes in the search tree is systematically generated, corresponding to one of the possible value combinations of all variables, and then tested to see if the values satisfies all constraints. The size of the leaf node set is bounded by the size of the Cartesian product of the variable domains, which is D^n, if D is the maximum of all domain sizes.

Efficiency of search can be enhanced with *chronological backtracking* (BT). Variables are instantiated sequentially until sufficiently many variables are assigned a value in order to verify a constraint. If a partial instantiation violates any of the constraints, the algorithm backtracks to the most recently instantiated variable that has still values to choose from. Backtracking filters in the sense that the instantiation of a new variable has to be

consistent with all previously assigned variables. The subspace of the Cartesian product of the domains containing the tuple on which a constraint failed is therefore eliminated by backtrack search. Therefore, node (aaaa) in the example cannot be part of a solution either. The worst-case complexity remains exponential in the number of variables because examples having no solution can be constructed where inconsistency is only detected at one of the leaf nodes. Furthermore, backtracking still suffers from *thrashing* [Mackworth, 1977a]. Thrashing occurs when inconsistencies are rediscovered throughout search. Single values, for example, that cannot be part of a solution will be regenerated each time. Similarly, inconsistencies of value pairs will be rediscovered. In the example, the inconsistency between $X_1 = a$ and $X_3 = a$ is rediscovered during search. This is why local consistency techniques are useful in order to remove inconsistencies before search.

2.6.2 Lookahead strategies

An intermediate scheme is to embed partial consistency techniques with respect to the yet uninstantiated variables into a backtracking algorithm, i.e. to use lookahead strategies [Haralick and Elliott, 1980], [McGregor, 1979]. At each node various degrees of consistency can be performed before instantiating variables. The type of algorithm is determined by the amount of consistency checking performed at a node. The more consistency checking is done at each node of the search tree, the fewer nodes will remain in the tree but the higher the cost of the overall processing. On the one extreme is GT and on the other is maintaining arc-consistency (MAC) [Sabin and Freuder, 1994], [Bessière and Régin, 1996], which integrates arc-consistency into the backtracking scheme. The other algorithms, BT, forward checking (FC) [Haralick and Elliott, 1980], directional arc-consistency lookahead (DAC-L) and bi-directional arc-consistency lookahead (BDAC-L) [Tsang, 1998] are implementations of partial consistency techniques.

In FC, each time a new variable is instantiated, the domains of all variables not yet instantiated and connected through a constraint to the current variable are made consistent with the current value. If the label of one of the variables becomes empty, the algorithm backtracks. DAC-L is similar to FC but maintains directional arc-consistency among the not yet instantiated variables in reverse order. Directional arc-consistency is defined under a total ordering $<$ of the variables. It simply examines each constraint $C_{i,j}$ once such that $X_i < X_j$ and removes any value from D_i that does not have a compatible value in D_j in the reverse order of $<$ [Tsang, 1993]. BDAC-L is similar to DAC-L but maintains directional arc-consistency in both directions between the unlabeled variables. MAC starts with arc-consistent domains and achieves full arc-consistency after each reduction.

A priori, arc-consistency is a generic concept that can be applied to any CSP under the assumption that there exists a refine operator establishing arc-consistency between pairs of variables. If such an operator exist for continuous domains, the various lookahead algorithms can also be applied to numeric CSPs.

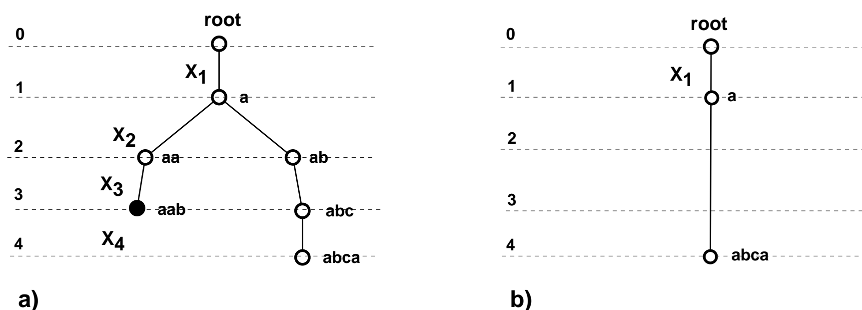


Figure 2.4: *Search tree of the CSP in Figure 2.3 solved by a) FC and b) MAC.*

2.6.3 Information-gathering strategies

Another key observation is that chronological backtracking repeatedly makes the same failures without learning from them. In information-gathering strategies, the search algorithm identifies the culprit for a failure in order to backtrack only to relevant decisions. A *culprit* is a variable with a committed label with which the current variable conflicts. These algorithms are part of the class of *dependency directed backtracking* algorithms.

Backjumping (BJ) [Gaschnig, 1979] only differs from BT when a variable can no longer be consistently instantiated (dead end). At this point, BJ gathers information about the failure by retaining the earliest culprit encountered for each value. From all culprits, one for each value, BJ backtracks to the most recently assigned variable that constrains the current one.

Graph-based backjumping (GBJ) [Dechter, 1990] does not analyze the failure for each value of the current variable like BJ, but it uses the graph-based structure of the CSP. It simply backtracks to the most recent variable constraining the current variable.

In conflict-directed backjumping (CBJ) [Prosser, 1993a] a conflict set is maintained for each variable containing the past variables that failed consistency checks. When there are no more values to be tried for a current variable, CBJ backtracks to the most recent variable in the conflict set of the current variable. The conflict set of the current variable is absorbed by the new variable's conflict set. CBJ has the ability to perform multiple backjumps across conflicts.

Backmarking (BM) improves backchecking (BC) [Gaschnig, 1977], which reduces the consistency checks by only pruning a variable domain constrained by earlier variables at the moment it should be instantiated. FC, on the contrary, prunes the variable domains of not yet instantiated variables as soon as possible. Backmarking avoids repeating compatibility checks that have succeeded. It remembers for each variable the highest level it backtracked to. When the current variable is instantiated anew, consistency checks have to be performed only from the highest level to which it backtracked to the previously assigned variable. These are the only variables whose label could have changed.

Hybrid algorithms [Prosser, 1993b] are algorithms that combine partial consistency techniques with intelligent dependency directed backtracking methods, e.g. FC-BJ, FC-CBJ etc.

Algorithms like GBJ and CBJ use the constraint graph to gain information about which node to backtrack to. These techniques are independent of the variable type and can also be applied to numeric CSPs. BJ, BM and their hybrids, however, work with culprits, e.g. assigned variables that cause conflicts. Storing information about conflicting values is in general costly in continuous domains because this requires an explicit representation of each value or value set.

2.6.4 Variable and value orderings

Instead of considering the variables in random order when searching, heuristics may improve backtrack search [Tsang, 1993]. *Static variable ordering* is established before search begins and does not change during search. *Dynamic variable ordering* relies on characteristics that change during search, it has therefore to be applied repeatedly during search. Static variable orderings take advantage of the structure of the constraint graph. The *minimum width* ordering minimizes the width of the constraint graph [Freuder, 1982b]. The *maximum degree* heuristic orders the variables by decreasing number of neighbors. Dynamic variable orderings try to maximize the possibility of detecting failures early, a principle called *first fail principle* [Haralick and Elliott, 1980]. The idea is to label the variable next that is the most constrained. Some simple measures for constrainedness are the size of its domain or the number of variables connected to it. When the first fail principle based on domain size is used together with lookahead algorithms, the ordering becomes dynamic, as domains change during the execution of these algorithms. A more sophisticated measure is a combination of domain size and maximum degree heuristic, which has been tested in different combinations in [Bessière and Régin, 1996]. Again, variable orderings do not depend on the type of variable domain and can as well be applied in numeric CSPs.

Value orderings can be used to determine which branches are explored first. The idea here is to prefer values that succeed. Value ordering applies well to discrete variable domains but not to continuous domains.

2.6.5 Comparing search algorithms

Before we discuss the different search algorithms and their advantages, we would like to emphasize that, in general, it can be quite difficult to compare these algorithms. Theoretically, the worst-case complexity remains exponential in the number of variables for all of them. Analyzes are therefore often carried out on a more experimental basis in order to predict the run-time behavior of these algorithms. Unfortunately, there is no unique measure of run-time behavior. Also, some authors generate all solutions whereas others only measure the efficiency of obtaining one solution to the CSP. Furthermore, some of these algorithms make extensive use of lookup tables whereas others do not use them. Some measures are however recognized to have an impact on the efficiency of the algorithms, such as the number of nodes visited in the search tree or the number of consistency checks performed. In the subsequent discussion, we will only refer to comparisons among these algorithms with respect to the number of nodes visited and the number of consistency checks performed. If not mentioned otherwise, variables are ordered statically.

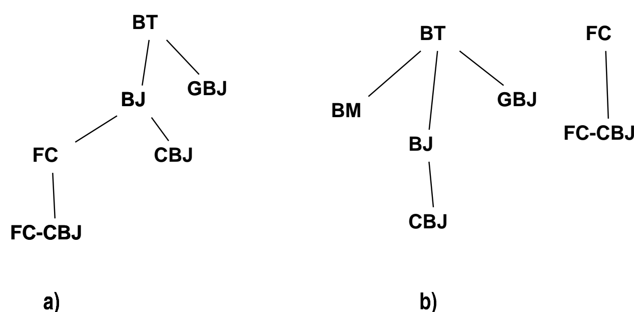


Figure 2.5: *The relationship between various lookahead and intelligent backtracking algorithms. a) $<_{cc}$: hierarchy with respect to the number of visited nodes and b) $<_{nv}$: hierarchy with respect to the number of consistency checks.*

[Haralick and Elliott, 1980] compare various lookahead strategies and conclude on an experimental basis that BM and BC basically generate the same search tree as BT, because they only concentrate on saving consistency checks. On top of that, they often fail at the deepest level in the search tree. Lookahead algorithms will perform more consistency checks but also detect incompatibilities earlier and therefore generate less nodes. In general, an efficient implementation of FC outperforms the other algorithms. On the other hand, [Prosser, 1993b] showed that there exist problems where BT performs less consistency checks than FC. He concluded in his experimental study that the hybrid FC-CBJ seems to perform best in most cases. Hybrids with BM can perform worse than BM itself, because the advantages of backmarking may be lost when jumping back. Orthogonally, [Sabin and Freuder, 1994] as well as [Bessière and Régin, 1996] show how MAC combined with a refined variable ordering heuristic outperforms FC-CBJ or FC on hard and large instances of random CSPs. Removing values from labels during tree search, however, can degrade intelligent backtracking in some cases [Prosser, 1993a], [Sabin and Freuder, 1994]. A more theoretical framework comparing different algorithm has been established by [Kondrak and van Beek, 1995]. With the aid of two relations $<_{cc}$ (uses no more consistency checks than) and $<_{nv}$ (visits no more nodes than), different algorithms are related as shown in Figure 2.5. The hierarchy in Figure b) reveals that FC and BT cannot be compared for the number of consistency checks, as predicted by Prosser.

These experimental studies reveal that there is not a "best" strategy but that there is a trade-off between applying consistency methods and different forms of intelligent backtracking. Most of these techniques have only been used to solve discrete binary CSPs. The challenge of extending them to non-binary and numeric CSPs still remains.

2.7 Search strategies for solving continuous CSPs

While there has been a lot of research involved in solving discrete CSPs, numeric constraints still pose a lot of difficulties. This is mainly due to the fact that, even if a variable domain is bounded by a lower and an upper limit, there are infinitely many possible values the variable can take on in its domain. An enumeration of the values for one variable not

to speak of the combination of values is computationally prohibitive. In this section, we review known methods used for solving numeric systems with continuous constraints in which the variables are given bounded domains; e.g. variable X_i has domain $D_i = [a, b]$ where $a, b \in \mathbb{R}$ and $a \leq b$. This section is meant to set a comparative framework in which we discuss issues relevant to our research. However, we do not attempt to give an introduction into solving numeric systems.

The problem of solving nonlinear system has been attacked in several fields, namely, optimization in linear and nonlinear programming, stochastic optimization, algebraic as well as numerical analysis, interval arithmetic and constraint satisfaction. We review each of these and discuss their relevance for identifying solution spaces as mentioned in the introduction.

2.7.1 Value inference

Value inference is used when some variables in the constraint network are already assigned. Values for the uninstantiated variables are then inferred by applying the constraints.

An example of inference method is given in [Serrano and Gossard, 1992]. The variables are split into a set of known and unknown variables. Graph-based methods allow a directed graph to be determined from the initial constraint graph. From the directed graph an order can be derived, in which the variables have to be processed in order to infer values for unassigned variables. This method also detects cycles in the graph and collapses the associated variables and constraints into super-nodes that have to be solved by a specialized method. The advantage of this method is that dependency information between variables can be used to find cycles, to identify variables that will never be assigned or to detect over-constrained systems. The disadvantage is that a solution in a cyclic graph can only be found once all collapsed nodes have been solved.

2.7.2 Propagating degrees of freedom

This method identifies parts of the constraint graph that have enough degrees of freedom so that values can be found that satisfy all their constraints. Regions like this and all the constraints that apply to them are removed from the graph. Deletion of the constraints may give another part enough degrees of freedom to be removed and so on. The part of the graph that is left is satisfied by some method and value inference is applied to propagate the values into the parts with a large degree of freedom, successively instantiating variables. Sketchpad [Sutherland, 1963] used propagation of degrees of freedom and if this failed resorted to relaxation methods (see Section 2.7.4 below).

2.7.3 Optimization techniques

Techniques for solving numeric systems developed in numerical analysis rely on global optimization of a given objective function subject to constraints on the variables involved. In general, optimization algorithms are iterative, starting from an initial solution and enhancing it stepwise to find a local optimum. Under the assumption that the constraints

and the objective function are convex, a local extremum found by such a method can be proved to be the optimal one. This is the case in linear programming (LP), where the function as well as the constraints are linear. The simplex algorithm [Danzig, 1965] used together with Gaussian elimination is an example of a feasible point method. Given an initial solution, it moves from one extreme point of the polyhedron defined by the constraints to another along a sequence of feasible descend directions (the edges of the polyhedron). For nonlinear systems, the direction of the optimum is derived from the gradient on the objective function. As long as no further convexity condition is imposed on the objective functions or the constraints, there is no guarantee for the result to be a global optimum [Reiner Horst and Thoai, 1995], [Nash and Sofer, 1996]. Optimization problems that are subject to constraints can be cast back into unconstrained ones by constructing an objective function from the constraints (the Lagrangian). Penalty methods, for example, add a penalty function (error function) to the objective function in order to minimize the error and to move in direction of the local optimum. This penalty function either measures the violation of a constraint or penalizes for reaching the boundary of a constraint (interior point method) [Nash and Sofer, 1996]. Approximation methods also exist, as for example the cutting plane method [Avriel, 1976]. The problem of these algorithms is that they are not guaranteed to converge for any type of constraint system and that convergence may be very slow. Furthermore, if the method converges, only one solution is found, which depends on how the initial values have been chosen. This means that one cannot predict which solution is found in an under-constrained problem. Most of these methods handle equality systems well but are not adapted to inequalities.

Many packages available in constraint programming use the simplex algorithm to solve the linear equations and delay the nonlinear part of the problem in the hope that it will simplify later. Example of such systems are PrologIII [Colmerauer, 1993] and CLP(R) [Heintze et al., 1987]. Some methods restricted to quadratic equations propose a piecewise linear approximation of the initially nonlinear functions and solve then the linear system, as for example in QUAD-CLP(R) [Pesant and Boyer, 1994].

2.7.4 Relaxation and perturbation methods

Both methods are similar in that all variables are assigned a specific value at the beginning. Some values are perturbed and the constraint solver has to adjust variable values such that the constraints are satisfied.

Relaxation algorithms relate to more traditional techniques employed in numerical analysis in order to solve nonlinear systems. In relaxation methods, the variables are instantiated to initially guessed values that do not necessarily satisfy the constraints. The error is measured by some heuristic and the guesses are adjusted accordingly. This is repeated until a solution is found. Typical penalty methods use gradient directions of the error function in order to find new values that minimize the error. The difference between penalty methods and relaxation methods lies in the way the error function is formulated. Relaxation methods use separate penalty terms for each constraint thus measuring the error for each constraint separately. This error evaluation can be executed in isolation

for each constraint and lends itself for a parallelized implementation. It is also easier to implement the treatment of numeric inequalities in such an algorithm. In applications like Sketchpad [Sutherland, 1963] and Juno [Nelson, 1985] relaxation methods were applied to solve coupled sets of constraints.

Perturbation methods have been developed especially in the context of interactive applications. At the beginning, variables have associated specific that satisfy the constraints. The value of one or several variables is perturbed by some outside influence, for example a user request, and the constraint solver has to adjust the values such that the constraints are again satisfied. One way to resatisfy the constraints is to propagate changes throughout the network using the constraints in order to recompute a value from given input values. Such a perturbation model is the basis of algorithms like DeltaBlue and SkyBlue [Sannella et al., 1993], [Sannella, 1993]. They necessite a planning cycle to determine which value is recomputed from which input values in a given constraint thus establishing a directed constraint graph. Value conflicts on a single variable are solved by leaving some constraints unsatisfied. This method is incremental. However, as [Trombettoni and Neveu, 1997] have shown, SkyBlue is exponential in contrast to DeltaBlue, which is polynomial. A further disadvantage is that cycles in the constraint graph are not treated and thus multiple constraints over the same variables are either not allowed or not solved.

2.7.5 Stochastic methods

Stochastic methods apply to global optimization problems and have been introduced in order to overcome the difficulty of getting stuck in a local optimum. These methods have a random aspect that reduces the chance of converging to a local optimum. Search is no longer limited to the search space around the starting point but allows for large search spaces to be traversed. Examples of stochastic methods are genetic algorithms (GA), tabu search (TS) and simulated annealing (SA).

GAs are based on the model of natural reproduction. In nature, inherited characteristics are encoded in the genes of every living being and those entities with the "best" genes survive to reproduce. The variable values are encoded in strings representing the chromosomes and a fitness function (the objective function) is evaluated for each string only selecting part of the population of chromosomes. The next population is created by applying crossover and mutation to the chromosomes and again the fitness function selects only a part of the mutated population. This process of selection and mutation is repeated until a good solution is found [Goldberg, 1989].

TS [Glover, 1989a], [Glover, 1989b] is a heuristic search method that tries to overcome problems in conventional hill-climbing procedures by 1) adding a list of tabu-moves and 2) providing an aspiration function. Tabu moves specify moves that will not enhance the current solution. They provide a kind of short-term memory in order to prevent the procedure of falling back into a local optimum already visited. The aspiration function, on the contrary, provides some strategic forgetting in that tabu moves may become legal again but still retaining the ability to prevent from falling back into local optima.

SA [Metropolis et al., 1953] is an adaptive search technique based on the process of metal annealing. A temperature parameter, which decreases during search, is added to the objective function. When it is sufficiently low, the algorithm gets stuck in a local optimum which is then supposed to be the global one.

According to [Thornton, 1993], who compared genetic algorithms with simulated annealing for solving problems in embodiment design, the critical aspect of heuristic search methods is tuning these heuristic parameters (energy for SA, fitness function and reproduction parameters for GA) correctly. Furthermore, this tuning may change for different numerical systems. The convergence might be as slow as for the traditional optimization methods. For SA it has been proven, that the temperature should decrease in very small steps to guarantee optimality, which implies a slow convergence of the algorithm. According to [Thornton, 1993] the SA algorithm performed better than the GA on embodiment design problems because the SA algorithm works directly on the original constraint system and does not need an encoding mechanism.

2.7.6 Algebraic methods

Algebraic systems are able to solve algebraic constraints at a symbolic level.

An algebraic method for solving polynomial systems over complex numbers has been developed by [Buchberger, 1965], [Buchberger, 1985]. This method reduces a given polynomial system algebraically to a standard form called Groebner bases, which are more easily solvable. One disadvantage of this method is that it is more appropriate for proving that a polynomial system has no solution than for solving it.

For proving quantified existential theorems of first-order theory over reals, Tarski [Hollman and Langemyr, 1993] has developed the so-called cylindrical algebraic decomposition CAD. In this method, polynomials are projected recursively in order to construct sign-invariant regions, called cells. This cylindrical decomposition makes it possible to compute the truth-value for a cell and then to determine the truth-value for the overall formula. The problem is that the algorithm's complexity is doubly exponential in the number of variables [Collins, 1975]. RISC-CLP(Real) [Hong, 1993] is a CLP-prototype using a combination of Groebner bases and CAD to solve non-linear equations. CAL [Sakai and Aiba, 1989] is another system that uses Groebner bases to solve polynomial systems.

2.7.7 Constraint logic programming (CLP)

The CLP framework can be seen as a generalization of the logic programming scheme. Most of the results from logic programming are easily adaptable to CLP by replacing the Herbrand universe with a constraint domain and unification by constraint satisfaction [Jaffar and Maher, 1994]. In CLP languages, the body of clauses may contain one or several constraints that have to be satisfied simultaneously. The underlying Prolog engine sequentially chooses clauses to satisfy and the constraints in their body are added to a so-called constraint store. Finding a solution to the new problem with the body of the current clause parsed, consists of incrementally satisfying the new constraint store [Cohen, 1990]. Most

of the CLP languages are dedicated to solving boolean and discrete constraints CLP(FD). Others solve linear numeric constraints [Heintze et al., 1987] CLP(R) and CLP(intervals) uses interval arithmetic to solve general numeric systems (see 2.7.8). The advantage of CLP systems is that they are based on an incremental constraint satisfaction algorithm and that the Prolog machine allows them to compute all solutions, their disadvantage is that they are restricted to a specific variable domain.

2.7.8 Interval analysis

Interval methods have originally been developed in order to render the resolution process more stable by evaluating the numerical errors committed during computation with floating-point numbers. When applying interval methods in a branch and bound approach to optimization problems, a method for searching a continuous space is created. The idea is to subdivide (branch) the constraint region into a finite number of subregions that have then to be tested for the optimum (bound). In general, search for individual solutions in numeric domains relies on the bisection of the variable labels in a round-robin fashion and is exponential in the number of variables. An interval of a variable label is split into two parts that are recursively searched. This splitting is applied to all variable labels. A search node consists thus simply of a set of intervals, one for each variable. At each search node, a refine operator can be applied to prune the intervals additionally. In this context, the pruning power due to an effective refine operator reduces the number of interval splittings necessary during search for a solution. The optimization function and the constraints are mapped into interval functions and interval arithmetics is applied to evaluate the functions on the interval sets.

CLP(BNR) is a system using pure interval analysis. It may happen that a given initial interval is not narrowed down at all [Older and Vellino, 1993].

Therefore, [Van Hentenryck et al., 1995] has developed a more powerful prototype called Newton and its successor, Numerica, using interval analysis coupled with consistency techniques. The system solves sets of non-linear polynomial equations or optimizes an objective function subject to nonlinear constraints. The system consists of a branch and bound algorithm applying a refine operator called box-consistency. Box-consistency is an approximation of arc-consistency capable of narrowing down the interval values of a variable. This operator uses the projection of a constraint onto one axis, transforms this projection into an interval function and determines the outermost solutions to this function with an interval Newton method.

CIAL [Chiu and Lee, 1994b], [Chiu and Lee, 1994a] enhances interval narrowing techniques for non-linear systems with capabilities of linear interval equality solvers. The linear solver is based on generalized interval analysis and has to be preconditioned by a Gauss-Seidel method.

[Hyvönen, 1992] uses an interval analysis approach called tolerance propagation. For each constraint, solution functions are defined to compute each variable value from the rest of the variables. These functions are extended to interval functions so that the variable values can be directly computed as intervals. The idea of his algorithm is to refine the

solution sets in a top-down manner thus creating a lattice of solution sets. His global search strategy includes an analysis of the search space by breaking down the space into monotone, well-defined continuous parts and then locally applying interval propagation. Cycles in a constraint system are isolated and solved by a conventional solver. The rest of the network can then be made globally consistent by applying local consistency techniques according to [Dechter, 1990].

The advantage of using interval analysis is that there are nice completeness (all solutions are found) and convergence properties derived from interval analysis and that there is also the possibility of guaranteeing the existence of solutions in the returned interval results. Hentenryck and Hyvoenen have implemented a search algorithm that finds one or all solutions of a constraint system. Most of the examples treated in [Benhamou et al., 1994], [Van Hentenryck et al., 1995], [Van Hentenryck, 1997] as well as in [Hyvönen, 1992] are fully-constrained systems with n constraints and n variables. It is however not obvious how this method performs on under-constrained problems. Since box-consistency has less pruning power than arc-consistency, it is not clear how much of the intervals is pruned away by the narrowing operator and how much time of the overall resolution time is spent by splitting intervals.

2.7.9 Consistency techniques

Only in recent years, interest has turned to consistency techniques for constraints defined on continuous domains.

[Davis E., 1987] was the first to report a series of negative results when applying the Waltz algorithm to continuous constraint problems, including the fact that the algorithm does not guarantee a locally consistent labelling and often fails to terminate. Davis represents variable labels as single closed intervals and applies the refine operator for a given constraint and variable by replacing the interval boundaries of all other variables into the constraint thus computing new interval bounds for the given variable. Replacing the interval boundaries into the constraint corresponds to approximating by a box the volume defined in space by the constraint and the propagated interval. The variable value corresponds to the projection of this box onto the respective axis. [Faltings, 1994] has shown that many of his negative results are due only to an overly straightforward formulation of the refine operator. He improves the propagation rule for binary constraints by propagating intervals through sets of constraints defined on the same pair of variables, called total constraints, and by considering not only intersections of interval bounds with the constraint but also local extrema of the region restricted through the total constraint. The refine operator proposed assures arc-consistency over binary constraints. If the constraint system has to form of a tree, it can thus be made globally consistent by applying Faltings' operator.

[Lhomme, 1993] proposes applying arc-consistency and, more generally, k-consistency only to the bounds of intervals (arc-B-consistency and k-B-consistency respectively). If a constrained region splits an interval into several subintervals, the result is a single convex interval encompassing the subintervals. Thus some values in the resulting interval may be

inconsistent. In his algorithm, given constraints are first transformed into a set of basic constraints for which it is possible to compute a minimum and a maximum of the projection of the constraint onto each axis. Then local consistency on bounds is enforced. To guarantee termination and to evaluate the complexity of the algorithms, Lhomme introduces the width ω a parameter specifying the authorized imprecision at the bounds that may occur during computations in the real domain. The worst time complexity of his arc-B-consistency algorithm is $\mathcal{O}(D * m)$ with m the number of constraints and D the domain size taking into account ω.

[Sam-Haroud, 1995], [Sam-Haroud and Faltings, 1996] defines convexity conditions for tractable global consistency in continuous domains. She gives an algorithm for global consistency that relies on a discretized representation of the constraint regions as 2-k-trees. The original constraint system first has to be transformed in a system in which all constraints are ternary. Each constraint is then discretized and represented as an octree. This representation allows her to combine 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, which can be assured in polynomial time, is sufficient for establishing global consistency. (3.2)-relational consistency guarantees that each triplet of constraints having two variables in common has a non-null intersection. In the general case, a constraint problem can be decomposed into subproblems verifying the convexity conditions. The problem here is that the discretized representation of the constraints is space-consuming and the polynomial complexity of the algorithm in $\mathcal{O}(N^5)$ where N is the number of variables is large.

Chapter 3

Local consistency techniques

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth ?"

Sherlock Holmes in *The Sign of Four*, The Strand Magazine (1890)

In this chapter, we discuss local consistency techniques that either approximate or compute exactly 2-consistency for discrete and continuous constraints. A new definition of local consistency is given for both types of constraints, which results in tighter labellings than other known local consistency techniques. The rationale behind this new definition is that search can be greatly enhanced by local consistency techniques with a good pruning power. The author would like to contribute to the present and past efforts in this domain by giving some insight into the difficulties of achieving local consistency especially in the continuous domain. The following topics are discussed:

- Fix-point algorithms for local consistency and their integration into a framework for mixed CSPs.
- Local consistency for binary continuous constraints as an introduction to the subsequent chapter on ternary continuous constraints. A prototype implementation of local consistency for binary continuous constraints is also discussed.
- Local consistency for discrete constraints.
- Local consistency for mixed constraints.

3.1 Fix-point algorithms for local consistency

Local propagation algorithms compute a concise representation of potential solutions for a given constraint system by locally propagating the effects of constraints. Algorithms for local consistency remove values from the domain of a variable for which no compatible value can be found for another variable. Such reduced domains are represented by *labels*. For discrete variables with a finite domain, the labels can be represented explicitly by the

```
procedure propagate
begin
  Q ← {(X_i, X_j, C_{X_i X_j}) | i ≠ j}
  while Q ≠ ∅ do
    remove element (X_i, X_j, C_{X_i X_j}) from Q
    L_new ← L_{X_j} ∩ refine(X_i, X_j, C_{X_i X_j})
    if L_new = ∅ then
      return inconsistent !
    fi
    if L_{X_j} ≠ L_new then
      L_{X_j} ← L_new
      Q ← Q ∪ {(X_j, X_k, C_{X_j X_k}) | k ≠ i, k ≠ j}
    fi
  od
end
```

```
function refine(X, Y, C_XY)
begin
  L ← all values v of L_Y such that
    there exists a value w in L_X and
    C_XY is satisfied by v, w
  return L
end
```

Figure 3.1: *General algorithm for ensuring consistency over pairs of variables.*

set of discrete values whereas the label of a continuous variable consists of a set of intervals. Polynomial-time algorithms exist for computing locally consistent labels for discrete CSPs, for example AC-3 described in [Mackworth and Freuder, 1985]. The AC-3 algorithm, a binary version of the original algorithm of [Waltz, 1975], is shown in Figure 3.1 and will be called **propagate**. It applies iteratively a **refine** operator to revise the label of a variable Y such that it only contains values compatible with the label of variable X and the constraint C_{XY}. If the label for a variable X_i changes, all variable pairs (X_i, X_j) that depend on the changed variable are added to the queue for further propagation. The refinement step is thus iterated over every ordered variable pair until it results in no further change for the labels.

The integration of continuous and discrete constraints into the same framework achieving local consistency has been studied especially in the field of constraint logic programming (CLP). CLP(X) languages are defined over a given constraint domain X with its logic theory and its specific language, e.g. CLP($\mathbb{R}$) stands for a CLP language over the domain of reals. The restriction of CLP to specific variable domains has been found a serious limitation when to solve real-world problems. Hence, the idea has emerged in this field to combine different constraint solvers within one constraint solving framework, which should enable the resolution of so-called *mixed* constraint problems. This "mixing" happens between:

- different types of variable domains such as floating-point intervals and integers representing symbolic domains
- different constraint types such as continuous and discrete constraints.

Generic approaches have been identified to solve mixed constraint problems:

1. A generic constraint management module: different constraint solvers cooperate in a larger constraint management module [Tinelli and Harandi, 1996].
2. operator integration: refine operators defined for each constraint type are directly integrated into a general fixed point algorithm [Benhamou, 1996].

If a CSP consists of different types of constraints that share no variables, each constraint set consisting of constraints of the same type can be solved independently. In that case, the different parts represent two disconnected subgraphs in the according constraint graph. A problem arises when the same variable appears in constraints of different type. Such constraints are often present in design problems:

$$\begin{aligned} V.volume &= V.bottomArea * V.height \\ C(M, V.volume) &:= \{(reactor\ [0,100])(mixer\ [0,1000])(storagetank\ [0,1000])\} \end{aligned}$$

The vessel volume in the mixer example 2.3 is involved in an equality, which specifies its value by the vessel height and the bottom area and also in a discrete constraint defining how the type of mixer influences the vessel volume and vice versa. In the equality, the label of the vessel volume is a floating-point interval whereas it can only take on interval values from a fixed set of integer intervals in the discrete constraint.

For an approach based on a constraint management module, the constraints are rewritten in a separate form and equalities ensure the communication between two solvers, one for the continuous and the other for the discrete constraints. For example,

$$\begin{aligned} &Solver1: && V.volume' = V.bottomArea * V.height \\ &Solver2: && C(M, V.volume) \\ &Communication: && V.volume = V.volume' \end{aligned}$$

This approach is problematic as soon as constraints share variables over several solvers, because it results in finding a fixed point for each solver independently plus a fixed point over the set of solvers. Furthermore, if the mixed CSP contains other continuous constraints not involving X and Y, these may be rechecked by a **propagate** algorithm at each call.[1] Also, the initial data structures like the queue etc. have to be rebuilt each time.

A neat framework has been proposed by [Benhamou, 1996]: He defines a solver as system that associates specific refine operators to each constraint type. Additionally, the refine operators compute values over an *approximate domain.*

Definition 3.1 (Approximate domain, [Benhamou, 1996]) *An approximate domain A over a domain D is a subset of $\mathcal{P}(D)$, with $\mathcal{P}$ the power set, closed under (eventually infinite) intersection and such that $D \in A$*

[1] This disadvantage has been removed in optimal consistency algorithms like AC-6 [Bessière, 1994] over discrete constraints. The author knows of no similar algorithm based on support propagation for continuous propagation algorithms.

The idea is that the result of a refine operator is additionally approximated if necessary in order to comply with different variable domain types. The definition of approximate domain is important if different operators use different domain representations. To quote Benhamou: "If a variable is fixed by the Simplex to a value which is not representable in the floating point representation used, the processing will consist in approximating the infinite precision rational with a floating point interval, apply the propagation algorithm and then intersect back the floating point interval representing the set of possible values for this variable with the initial rational number... ".

If we apply a direct integration of refine operators, a single queue containing different constraint types with their refine-operators associated is created in the fix-point-algorithm **propagate**. The refinement step in line 4 of Figure 3.1 applies the operator according to the constraint type. A change in the label of a shared variable has the effect of adding new elements to the queue. In the first approach, such a change caused within one solver has the effect of adding all constraints over this variable within the second solver. In the second approach, the queue consists of different constraint types and some constraints, which should be added, are already in the queue. Before and after a call to the dedicated refine operator, a transformation step may be necessary, which interfaces the results of the operator taking into account the approximation of domains.

In the light of these remarks, we consider the approach of integrating refine operators for discrete and continuous constraints into the propagate-algorithm and discuss in the following sections refine operators for discrete and mixed constraints as well as the necessary transformation functions between variable domains.

3.2 Local consistency for binary numeric constraints

In the following sections, the general definition of a CSP given in section 2.4 is restricted to variables with continuous domains and numeric constraints. We consider *numeric CSPs* defined on continuous variables, as follows:

Definition 3.2 (Numeric CSP) *A* numeric CSP *is given by*

- *a set of variables* $\{X_i\}$, $i = 1, \ldots, n$ *with labels* L_i *associated each having its domain* D_i *in* $\mathbb{R}$
- *a set of continuous constraints* $C_{\mathbf{X}}$ *on a set of variables* X. *Each constraint* C *is built out of an expression* E^C *and relation* $\bigodot = \{\geq, \leq, =\}$. *We normalize the syntax by allowing only inequalities of the form* $E^C(\mathbf{X}) \geq 0$ *with* $\mathbf{X}$ *a set of variables from* $\{X_i\}$. *An equality* $E^C(\mathbf{X}) = 0$ *can be modeled as a conjunction of inequalities* $E^C(\mathbf{X}) \geq 0$ *AND* $E^C(\mathbf{X}) \leq 0$. *Unary and binary constraints are defined in the same manner with one respectively two variables.*

A solution to the numeric CSP is a set of values $\{X_i = x_{i_j}\}$ *such that* $x_{i_j} \in L_i$ *that satisfies all constraints.*

In the following, $\mathbf{X}$ is restricted to two variables. A solution satisfies the *conjunction* of the constraints in the numeric CSP. Geometrically, a constraint $C_{XY} : E(X, Y) \geq 0$ defined on

two variables X and Y with initial labels L_X and L_Y can be represented in Euclidean space $\mathbb{R}^2$ defined by the axes X and Y. The feasible region defined by the inequality represents an arbitrary shape in this space. The conjunction of numeric constraints defines a *feasible region* in $\mathbb{R}^n$ (where n is the number of variables) in which the solution must be located.

3.2.1 Computing in continuous domains

Nowadays, computers can only store and manipulate a finite amount of information. Solving nonlinear problems, which appear in many fields such as chemistry and structural engineering, however, may produce real-valued solutions. The computer is only able to represent a finite amount of numbers on the real axis called floating point numbers. Since any real solution is approximated by the computer, algorithms over the real domain implemented on the computer are bound to make errors and also to propagate these errors due to finite precision arithmetic.

An interesting example simulating the effects of rounding errors is the Wilkinson problem reported in [Van Hentenryck, 1997] that consists of finding all solutions for X to the equation

$$\prod_{i=1}^{20}(X+i) + pX^{19} = 0$$

in the interval $[-20.6, -9.6]$ with a factor p quantifying a rounding error. When $p = 0$, the equation has 11 solutions $X = -20, -19, \ldots, -10$. When $p = 2^{-23}$ the solutions within the feasible interval are removed. The value of p could typically be a numerical error that drastically changes the solution set of the function.

Computational methods dealing with real numbers should therefore know the proximity of the computed solution to the real solution. This topic has been attacked by interval arithmetic. The key idea in interval arithmetic is the approximation of real numbers by intervals in order to quantify the errors introduced by the finite precision arithmetic ([Alefeld and Herzberger, 1983], [Moore, 1966]). Let $\mathbb{F}$ be the set of floating-point numbers. A real value $r \in \mathbb{R}$ should be enclosed into the interval bounded by r^+ and r^- where $r^+, r^- \in \mathbb{F}$ are the two floating point values nearest to r, e.g. $r \in [r^-, r^+]$. The results of arithmetic operations executed on the computer are "outward-rounded" in order to preserve the correctness of the computations.

In our thesis, we have not focused on these concerns of how to approximate real-numbers on a computer and the results that we present are floating point values as computed with the prototype implementation. For any attempt at implementing our algorithms, it is however inevitable to use interval arithmetic software in order to guarantee the precision of the results. This does not affect the general correctness of our algorithms as proved in this thesis but points out the difficulties encountered when implementing any algorithm computing with real numbers on a machine.

The algorithms we developed for local consistency in continuous domains make use of some local properties of the constraint curves and surfaces like local extrema. Based on the assumption that these local properties can be determined, a theory of how local consistency can be computed is presented and its correctness is proved. Nevertheless, a

short description of how the required local properties might be determined is given at the end of the chapter without claiming to be complete. It would be nice to indicate a very general class of constraints for which these local properties can always be determined correctly. Establishing such a characterization would require a more profound investigation in the topics of functional and topological analysis, which is beyond the scope of this thesis.

3.2.2 Existing refine operators

Few algorithms computing local consistency in the continuous domain exist. All of them result in some crude approximation of the projection of a constraint region onto the axes. It is symptomatic in this field to find very different approximations and it is often difficult to compare them. An explanation for this variety may be that the underlying problems and difficulties have not yet been fully understood. The reason why some researchers are interested in local consistency for continuous constraints is that any search in continuous domains consists of expensive interval splitting and that a search procedure interleaved with efficient local consistency techniques reduces the number of splittings. In the following, we point out the characteristics of existing methods:

- Waltz algorithm [Davis E., 1987]
- 2-B-consistency [Lhomme, 1993]
- box-consistency [Benhamou et al., 1994], [Van Hentenryck et al., 1995]
- simple-propagate [Faltings, 1994]
- tolerance propagation [Hyvönen, 1992]

with respect to soundness, completeness, as well as termination criteria. Soundness is understood in the context of 2-consistency; e.g. soundness examines if the resulting labels still contain values for which no value for a second variable can be found. Completeness of local consistency is very important when a search algorithm is proposed that finds all solutions to a given numeric CSP. It would not be acceptable to use a local consistency propagation step that loses solutions during search. It would also be interesting to compare the tradeoff between efficiency and the pruning obtained by the different operators. However, this aspect is hard to measure because the algorithms proposed do not apply to the same type of examples nor are their implementations comparable. In this section, we would merely like to understand how these algorithms determine feasible constraint regions.

To make the presentation easy, binary numeric constraints are used in the examples and we concentrate on one propagation step executed by **refine**. The underlying idea of **refine** is to approximate the projection of the feasible region defined by the constraints onto a given axis in order to determine the range of feasible values for this axis. In Figure 3.2 a) for example, the feasible region that lies within the label of X, denoted by L_X, results in a convex interval for the label of Y when projected onto the Y axis. To compute this projection, projection function are in general derived, which resolve a constraint expression for each variable. From $C_{XY} : E(X,Y) \leq 0$ result the two projection functions $X = F_1(Y)$

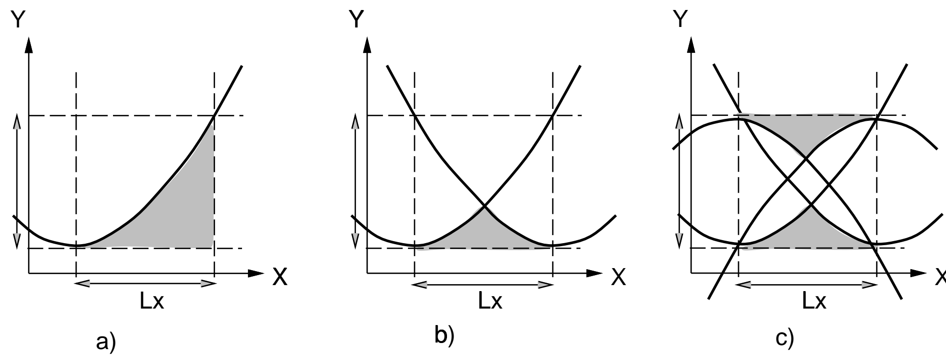


Figure 3.2: *a) The projection onto the Y axis of the constraint region can be computed from the intersection of L_X with the constraint. b) If the projections are computed individually for each constraint, intersections between constraints are neglected and the resulting label for Y is locally unsound. c) like b) with the effect that inconsistency is not detected by an individual propagation of the constraints. Taken from [Faltings, 1994]*

and $Y = F_2(X)$. Some algorithms approximate this projection, i.e. they try to find an enclosure of the projection.

1. **Davis' propagation rule**: The labels resulting from his algorithm are always convex intervals. Propagating the label $L_X = \{[1,4]\}$ through a constraint like $Y = X^2$ results into $L_X = \{[-2,2]\}$ in a convex representation instead of $L_X = \{[-2,-1],[1,2]\}$. Unsound values between sound ones are thus preserved. The algorithm uses the projection functions to compute the range of feasible values for each variable and each constraint independently. Depending on the topology of the constraint region and the interval labels to propagate, this approach applied naively does not always preserve completeness. In example a) in Figure 3.3, such an algorithm is not able to conclude if there exists a region within the propagated interval or not, b) shows an incomplete label L_Y and c) a complete one. Davis also points out cases of non-termination: Let C_1 be $Y = 2 * X$ and C_2: $Y = X$ and $L_X = L_Y = \{[0,100]\}$. From C_1, $L_Y = \{[0,50]\}$ is deduced and from C_2, $L_X = \{[0,50]\}$ is deduced. Then, again from C_1, $L_Y = \{[0,25]\}$ etc. The correct result $X = Y = 0$ is only attained asymptotically.

2. **Tolerance propagation**: The algorithm computes labels that are sets of intervals. Each constraint is propagated individually and the results for each variable of the constraint are computed by the natural interval extension of the projection functions (Appendix C). To preserve completeness, Hyvönen proposes to decompose the initial interval labels such that the projection functions are monotonic and continuous over the decomposed intervals. Such a monotonicity analysis is conducted on the first partial derivatives of each function (Appendix C). The disadvantage of his method is that the number of subintervals to be considered may grow in a combinatorial manner.

3. **2-B-consistency**: Another way to guarantee a complete propagation step is to implement a subset of all possible functions, called basic functions, for which the minimum and maximum value of the projection can be computed directly. A given, arbitrary constraint set is decomposed into these basic functions. Lhomme proposes an algorithm called B-consistency for bound-consistency, which uses basic functions to achieve 2-consistency only for the bounds of intervals. Thus, the resulting labels are always convex intervals. The advantage of his approach is that the computations are fast because they can be hard-coded. A problem is that a the locally consistent solution space computed from basic functions may loose in accuracy with respect to an implementation that considers general constraints directly. To guarantee termination, Lhomme parameterizes 2-B-consistency by an adjustable precision factor on the results.

4. **Box-consistency**: The idea of box-consistency is to approximate 2-consistency guaranteeing completeness and robustness of the results. Box-consistency is achieved by replacing all variables but one by its interval domain and by applying one of the narrowing operators well-known in interval analysis[2] in order to determine a tight approximation of the projection of an individual constraint onto the given axis. The advantage of this approach is that the constraints are considered directly without decomposition into basic constraints and that results from interval analysis also apply. It is for example possible to guarantee the existence of a solution in the resulting intervals or to do some optimization. Furthermore, interval analysis takes into consideration propagation errors. However, it is a well-known result from interval analysis, that the operators used in interval analysis overestimate the projection if a constraint contains the same variable several times as is the case of X in a polynomial like $Y = 1 - 5X + X^3/3$ (Figure 3.4). This can be explained by the fact that the dependency between both occurrences of X is lost as soon as it is replaced by an interval. A precision factor can be used in this algorithm to guarantee fast termination.

5. **Simple-propagate**: Faltings operator is complete and locally sound because it uses topological arguments to ensure the consistency condition defined over pairs of variables; i.e. it implements exactly the **refine** step. Faltings has proved, that in order to achieve 2-consistency also intersections between constraints defined in the same two-dimensional space (defined over the same pair of variables) have to be considered. Considering only one constraints at each propagation step leads, in general, to looser labellings, which include unsound values or even misses inconsistencies as shown in Figure 3.2. The bounds of locally consistent intervals for a label thus originate either from local extrema of constraint curves, intersections between constraints or intersections between constraint curves and interval bounds. The determination

[2] For more complex functions, interval analysis proposes narrowing operators that enclose the projection; i.e. their result is complete but very often not sound, like natural interval extension or Taylor interval extension. The goal of these different extensions is to approximate the projection of the feasible constraint region onto an axis as accurately as possible.

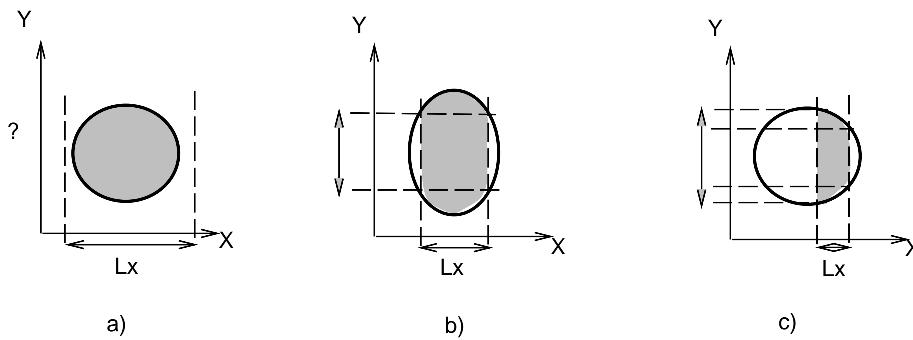


Figure 3.3: *A refine operator based on the intersection of interval bounds of L_X with the constraint boundary may not be able to determine if a locally consistent label exists for Y as shown in a), or find an incomplete label b) or result in a complete and sound label as shown in c).*

of local extrema on constraint curves corresponds to the monotonicity analysis of other approaches because these extrema account for a change in the monotonicity of a function defining the projection of a constraints onto an axis. Furthermore, Faltings eliminates one cause of cycling that occurs inbetween constraints defined on the same pair of variables by considering them jointly in a single propagation step. However, cycling involving several constraints defined over different two-dimensional spaces cannot be prevented and is in fact a well-known phenomenon occurring in algorithms operating locally like **refine**.

In many cases, the refine operators presented in literature result in too loose labellings, i.e. they include unsound values. A first step in the direction of a locally sound operator has been made by Faltings' operator, which takes into account the monotonicity of constraints and considers constraints that are defined over the same pair of variables simultaneously. This refine operator is presented in detail in the subsequent sections.

3.2.3 A refine operator for binary numeric constraints

In this section, we recapitulate results of [Faltings, 1994] who proposes a new refine operator for binary numeric constraints. In the following, references to this paper are omitted. His operator computes tighter labellings than the others presented in the previous section because it takes into account all constraints defined over the same pair of variables. In this section, we present this method, describe an implementation showing the feasibility of this approach and to lie the ground for an extension to ternary constraints.

In general, a locally consistent labelling has to be represented as first order labelling, i.e. a *set of intervals*. Differently from Davis, Faltings therefore defines:

Definition 3.3 (Label) *The label L_i of a variable X_i is a set of disjoint intervals $\{I_1, I_2, \ldots, I_k\}$. An interval I_k is an ordered set of real values $r_{min}, r_{max} \in \mathbb{R}$ written as $[r_{min}, r_{max}]$*

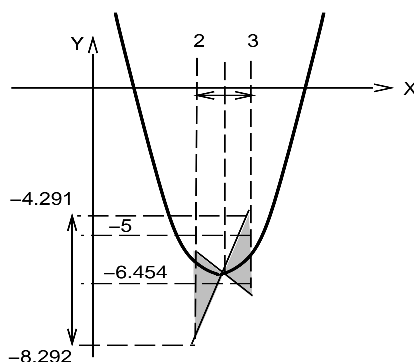


Figure 3.4: *An interval extension based on the first-order Taylor form taken around the center of I_X for the equation $Y = 1 - 5X + X^3/3$ over the interval $I_X = [2, 3]$. The projection of the shaded region onto the Y-axis defines the interval value for the extension, which is $[-4.291, -8.292]$. The consistent values for Y are also shown. Higher order Taylor forms result in still tighter approximations. Courtesy C. Bliek, AI-Lab, EPF.*

We consider algorithms for computing locally consistent labellings according to the following definition:

Definition 3.4 (Local consistency for binary constraints) *A label L_X of a variable X is* locally consistent *with the label L_Y of variable Y if and only if:*

$$(\forall x_0 \in I_X, I_X \in L_X) \quad (\exists y_0 \in I_Y, I_Y \in L_Y) \quad [\forall C^i_{XY} : E^i(x_o, y_o) \geq 0]$$

Labels are assumed to be finite but can contain arbitrarily large intervals. A labelling is called locally consistent if the labels of all ordered pairs of variables are locally consistent. A labelling is complete if it contains all solutions to the constraint satisfaction problem. A propagate-algorithm is complete if it computes a complete labelling and locally sound if it computes a locally consistent labelling.

Note that this definition is an adaptation of arc-consistency given in section 2.5 to continuous domains, in the sense that *all* constraints defined on a pair of variables X, Y have to be satisfied by a value for Y given a value for X. Furthermore, we only consider general algorithms for local consistency without any attempt to achieve global consistency as proposed for example in [Hyvönen, 1992] or [Lhomme, 1993]. The necessity of considering all constraints defined on the same pair of variables simultaneously leads to the definition of a total constraint.

Definition 3.5 (Total constraint) *A total constraint is a set of bounded* feasible regions *$C^t_{XY}(x, y) = \{Q_1, \ldots, Q_k\}$ containing exactly those combinations of values (x, y) for X and Y that are consistent with all constraints $C_{XY}(x, y)$.*

A single region is a point set which is always connected in the sense that it is formed from a set of points such that every point can be connected by a path lying within the set to

```
function refine(X, Y, C_XY)
begin
  I_Y ← {}
  for all I_X ∈ L_X do
    I_Y ← I_Y ∪ simple-propagate(I_X, C_XY)
  od
  IU ← union of all intervals in I_Y where overlapping intervals
    are merged into single convex ones
  return IU
end
```

Figure 3.5: *The operator* **refine** *for numeric CSPs. It applies a propagation rule* **simple-propagate** *to each interval of a label and merges the resulting intervals.*

each other point and which has a boundary $B(R)$ (Appendix B). When constraints define a region, the boundary of this region is the set of points that verifies all constraints and at least one constraint as equality. The regions Q_i in Figure 3.6 form the feasible regions defined by the conjunction of the constraints.

3.2.3.1 An improved propagation rule for a single interval

This section concentrates on a rule **simple-propagate** propagating a single interval through a total constraint. This rule is repeatedly applied to all intervals of the label as shown in algorithm 3.5 in order to compute the new arc-consistent label. It computes an arc-consistent label for Y (with respect to the definition) by propagating a single interval I_X through the total constraint C^t_{XY}. Remember that a total constraint C^t_{XY} is the set of feasible regions Q defined by the *conjunction* of all constraints defined on the pair X, Y. **Simple-propagate(I_X,Q)** computes the intervals of Y-values occuring in the subregions of a feasible region Q with $x \in L_X$.

Definition 3.6 (Restricted regions) *For a feasible region $Q_i \in C^t_{XY}$, the set of* restricted regions $R(Q_i, I_X) = \{R_1, \ldots, R_k\}$ *are those maximally connected subregions of Q that have X-coordinates entirely contained within an interval I_X.*

The total constraint and I_X might define several restricted regions with complex shapes as for example in Figure 3.6. The interval boundaries for a new label are given by extreme points at the boundary of the regions. Each restricted region $R_j \in R(Q_i, I_X)$ defines a single continuous interval of arc-consistent values for Y, which is bounded by a local minimum and local maximum in Y:

$$I_Y = [min\{y|\exists(x, y) \in R_j\}, max\{y|\exists(x, y) \in R_j\}]$$

A *local extremum* in Y of a point set B is defined as follows:

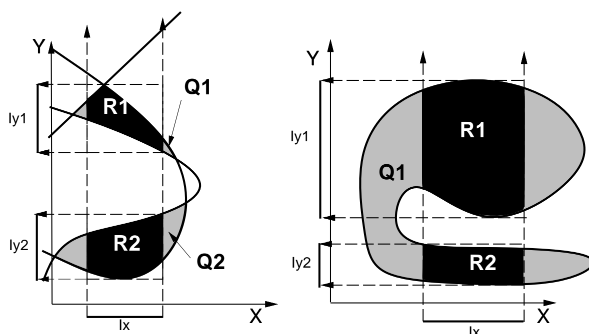


Figure 3.6: *Two examples of total constraints. The constraint on the left consists of the two feasible regions $Q1$ and $Q2$. When propagating from X to Y, the interval Ix generates the restricted regions $R1$ and $R2$, which project into intervals $Iy1$ and $Iy2$. The example on the right shows that multiple restricted regions $R1$ and $R2$ can result from a single feasible region $Q1$.*

Definition 3.7 (Local extremum) *Let $I \subset R$ be a connected set in B, a closed subset of $\mathbb{R}^2$, and two elements $(x_1, y_0), (x_2, y_0) \in B$. I is a* local maximum (minimum) *in Y if there exists a real number $\epsilon > 0$ such that $N_\epsilon = [x_1 - \epsilon, x_2 + \epsilon]$ and $I = \{(x, y_0) | x_1 \leq x \leq x_2\}$ imply that for any $(x', y') \in B$ such that $x' \in N_\epsilon$ and $y' \geq y_0$ ($y' \leq y_0$), (x', y') is part of I. A* local extremum *is either a local minimum or a local maximum.*

An extremum is specified by its *coordinates*, its *type* (minimum or maximum) and the *axis* in which it occurs. In the definition, an extremum is a point when $x_1 = x_2$ or it can be a line when $x_1 < x_2$. Note furthermore, that each of these extrema is a convex set of points, i.e. two extrema that are at the same Y-coordinate will considered as different if they are not connected. Different types of local extrema are allowed by the definition: a maximum might be a maximum of the entire region or only of its boundary; i.e. the feasible region may lie on either side of the boundary. We adopt the following notation:

- $ext_Y(B(R), y_0) \Leftrightarrow B(R)$ has a local extremum (minimum or maximum) in Y at coordinate y_0
- $ext_Y(B(R), y_0)$ and R is convex (concave) in the neighborhood of $y_0 \Leftrightarrow$ the extremum is additionally qualified as convex (concave) local extremum

The additional distinction with respect to the convexity of the region is important in order to recognize unbounded regions. A bounded region will always start with a convex local maximum and end at a convex local minimum (section 3.2.4). It is also important to note that only extrema which satisfy all constraints lie on the boundary of the restricted regions are valid. A criterion for excluding values of Y not arc-consistent with I_X is based on the index of a region $\alpha_Y(R, y)$:

Definition 3.8 *The index $\alpha_Y(R, y)$ is the difference in the number of maxima and the number of minima on the boundary of the region $R \in \mathbb{R}^2$ at Y-coordinates greater than y:*

$$\alpha_Y(R, y) = |\{max_Y(B(R), y_0) \mid y_0 \geq y\}| \quad - \quad |\{min_Y(B(R), y_0) \mid y_0 \geq y\}|$$

Using this definition, Faltings proves that $\alpha_Y(R, y) = 0$ for a region with a closed boundary and for any line segment not intersecting the region R and that $\alpha_Y(R, y)$ is strictly positive if the line segment intersects R.

Lemma 3.1
Let R be a two dimensional region and $B(R)$ its closed boundary.

- *If there is no point $(x^*, y^*) \in R$, then $\alpha_Y(R, y^*) = 0$.*
- *If there is a point $(x^*, y^*) \in R$, then $\alpha_Y(R, y^*) > 0$.*

Proof sketch: The proof is based on the observation that on a two-dimensional closed curve B, local minima and maxima in Y occur in alternating order (Figure 3.7). This is a consequence of the mean-value theorem and the continuity of B. The alternating order implies an equal number of minima and maxima (for a full proof see [Faltings, 1994]) ▲

The index $\alpha_Y(R, y^*)$ of a region R at $Y = y^*$ equals the number of connected intervals of $Y = y^*$ in R.

Lemma 3.2
Given a line $L : Y = y^*$ intersecting a closed curve B and the part I of B between two successive intersections with L. Either all points in I have a Y-coordinate $\geq y^*$ and I has exactly one more local maximum that minimum in Y or all points in I have a Y-coordinate $\leq y^*$ and I has exactly one more local minimum than maximum in Y.

Proof: see [Faltings, 1994] ▲

Lemma 3.3
Let $S = \{(x, y) \mid (x, y) \in R \wedge y = y^*\}$ be the intersection between R and the line $Y = y^*$ and $|S|$ the number of maximally connected regions of S. Then

$$\alpha_Y(R, y^*) = |S|$$

Proof: Suppose, L intersects R n times and we have n parts of $B(R)$ called I_j such that all Y-coordinates are greater or equal than y^*: $|S| = n$. The sum of differences between the number of maxima and the number of minima of each interval is exactly 1 (using the Lemma 3.2): $\sum_{j=1}^{n} \mid \{max_Y(I_j, y) \mid y \geq y^*\} \mid - \mid \{min_Y(I_j, y) \mid y \geq y^*\} \mid = n = \alpha_Y(R, y^*)$ ▲

Lemma 3.1 yields a criterion for deciding if there exists an arc-consistent y-value for Y given I_X: the propagation rule should eliminate a value y^* from the label L_Y if and only if it is in no region, i.e. $\alpha_Y(R_i, y^*) = 0$ for all $R_i \in R$. Since $\alpha_Y(R_i, y^*) \geq 0$ for any R_i,

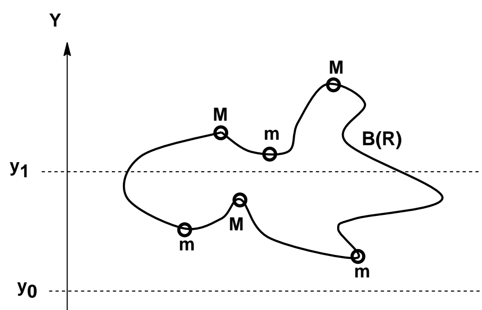


Figure 3.7: *An arbitrary region bounded by the curve $B(R)$. M denote local maxima in Y and m are local minima in Y. In this example, $\alpha_Y(R, y_1) = 1$ and $\alpha_Y(R, y_0) = 0$.*

this can be expressed as $\sum_i \alpha_Y(R_i, y^*) \geq 0$ because a sum of positive numbers is always positive. By rewriting this as:

$$\sum_i \alpha_Y(R_i, y^*) = \sum_i |\{max_Y(B(R_i), y_0) \mid y_0 \geq y^*\}| - \sum_i |\{min_Y(B(R_i), y_0) \mid y_0 \geq y^*\}|$$

we only require the total number of extrema in *all* restricted regions taken together, without any consideration of the regions they belong to. Only if the sum of the indices $\alpha_Y(R_i, y^*)$ of all regions R_i is zero, y^* does not belong to the locally consistent label of Y. The set of illegal y^*, and conversely the set of legal y, can thus be characterized without knowing which extrema belong to which region.

3.2.3.2 The algorithm Simple-Propagate

Under certain reasonable assumptions (specified in section 3.2.4), a candidate set of all extrema verifying Definition 3.7 can be generated by computing all extrema belonging to one of following classes:

1. local extrema on constraint curves
2. intersections between constraints
3. intersections with the boundaries of I_X and constraint curves

Again, the set of extrema can be determined by purely local considerations without knowing which of the restricted regions they belong to. Extrema of class 2 and 3 have to be considered because they define discontinuities and thus extreme points on the boundary of the restricted regions. Even if finding local extrema of classes 1 and 2 can be computationally expensive, it is sufficient to precompute these extrema once and for all for a given constraint system, because they do not change throughout propagation. Extrema of class 3, however, will change as interval boundaries are tightened during propagation. Except from this preprocessing, our algorithm only evaluates the constraints at interval bounds. The amount of constraint manipulation required in each propagation step is the same

as that of other more straightforward propagation rules solely based on the intersections between interval bounds and constraints.

The propagation rule for a single interval **simple-propagate** is shown in Figure 3.8. It can roughly be divided into three steps:

a) **identify-candidates**: finds local extrema in the form of extrema on individual constraint curves, intersections between constraints and intersection between constraints and interval bounds and classifies them as minimum or maximum (convex or concave),
b) **filter-candidates**: filters away those extrema that do not satisfy all constraints in C^t_{XY} or whose X-coordinate is not in I_X, thus pruning extrema that do not lie on the boundary of the restricted regions, and orders the remaining extrema in decreasing value of Y such that for the same value of Y a maximum is always considered before a minimum
c) **compute-intervals**: computes the locally consistent intervals for Y using Lemma 3.1 as shown in the algorithm 3.8 assuming an ordered set of extrema as input

The discussion of how the candidate set of extrema may be computed (function **identify-candidates**) is deferred to section 3.2.4.

3.2.3.3 Completeness and soundness

Faltings has further shown, that a propagation algorithm as presented in 3.8 using the propagation rule **simple-propagate** is *locally sound*. This is true because after reaching quiescence, the labelling is locally consistent and because the propagation rule never removes locally consistent values. It is complete because an arc-consistent labelling contains all solutions to the CSP.

3.2.3.4 Example

As an example, consider the feasible regions described by an ellipsis, a parabola and a line constraint (Figure 3.9).

$$\begin{aligned}
C_1 &:= (0.25 * X + Y - 6.2)^2 + 0.2 * (X - 0.25Y + 1.7)^2 - 5 \geq 0 \\
C_2 &:= -0.7 * X + 0.3 + 0.6 * Y - 1/3 * (0.6 * X - 4.2 + 0.7 * Y)^2 \geq 0 \\
C_3 &:= Y - 9/8 * X - 9 \leq 0
\end{aligned}$$

In a preprocessing step, we compute all local extrema on constraint curves and intersections between pairs of constraints and store them in as shown in the table of Figure 3.9. Local extrema on constraint curves stem from a single constraint (column C_i of the table), whereas intersections are points belonging to two constraints. For a propagation step from X to Y with initial labels $L_X = \{[0.5, 2.2]\}$ and $L_Y = \{[0, 5]\}$, **simple-propagate**$([0, 5], \{C_1, C_2, C_3\})$ executes the following steps according to algorithm 3.8:

```
function simple-propagate(I_X, C^t_XY)
begin
  E ← identify-candidates(I_X, C^t_XY)
  E ← filter-candidates(E, I_X, C^t_XY)
  L_Y ← compute-intervals(E)
  return L_Y
end

function compute-intervals(E)
begin
  I_Y ← {}
  α ← 0
  if unbounded restricted region then
    add extremum to E at ∞ or −∞
  fi
  for each e ∈ E do
    if e is a maximum (convex or concave) then
      α ← α + 1
    elif e is a minimum (convex or concave) then
      α ← α − 1
    fi
    if α has changed from 0 to 1 then
      y_max = y-coordinate(e)
    elif α has changed from 1 to 0 then
      y_min = y-coordinate(e)
      I_Y ← I_Y ∪ {[y_min, y_max]}
    fi
  od
  return I_Y
end
```

Figure 3.8: *Binary propagation rule* **simple-propagate** *for a single interval. How unbounded regions are detected is discussed in the section on filtering relevant extrema.*

a) local extrema in Y are retrieved from the table :

concave Max at $(-1.98, 8.65)$
concave Min at $(1.7, 3.82)$
convex Min at $(1.14, 2.53)$
convex Max at $(-3.35, 5.23)$
convex Max at $(1.46, 10.64)$
convex Min at $(3.11, 7.21)$
convex Max at $(3.62, 4.14)$

b) only those satisfying C_1, C_2, C_3 and such that $x_e \in I_X = [0.5, 2]$ are considered further:

convex Max at $(1.46, 10.64)$
concave Min at $(1.7, 3.82)$
convex Min at $(1.14, 2.53)$

a) intersections with interval boundaries of $I_X = [0.5, 2]$ are computed of which only the points

convex Min at $(2, 7.79)$
convex Max at $(0.5, 3.91)$
convex Max at $(2, 3.83)$

are local extrema in Y. The crosses in Figure 3.9 show points that have been eliminated from consideration because they are not extrema in Y.

c) the resulting extrema are sorted in decreasing order of y-values and the index α is computed for each extremum according to **compute-intervals**:

Extremum (x_e, y_e)	α	y_{max}	y_{min}	I_Y
-	0	-	-	$\{\}$
convex Max $(1.46, 10.64)$	1	10.64	-	$\{\}$
convex Min $(2, 7.79)$	0	-	7.79	$\{[7.79, 10.64]\}$
convex Max $(0.5, 3.91)$	1	3.91	-	$\{[7.79, 10.64]\}$
convex Max $(2, 3.83)$	2	3.83	-	$\{[7.79, 10.64]\}$
concave Min $(1.7, 3.82)$	1	-	-	$\{[7.79, 10.64]\}$
convex Min $(1.14, 2.53)$	0	-	2.53	$\{[2.53, 3.91], [7.79, 10.64]\}$

3.2.4 Implementation

The goal of this section is manyfold: first, to justify our commitment to an anlytical representation of constraints, second, to describe qualitatively what kind of point sets verify the definition of a local extremum 3.7 when regions are defined by constraints and third, to present the current prototype implementation of **identify-candidates** as one of several possible approaches. The reader should not forget that under the assumption that local extrema of restricted regions are found, our algorithm is correct, sound and locally complete. It is not the goal of this thesis to propose new methods for identifying extrema of arbitrary constraint regions. These remarks also remain valid for the extension of the propagation rule to ternary numeric constraints.

Type	Axis	Extremum (x_e, y_e)	C_i
concave Max	X	$(4.59, 5.3)$	C_1
concave Min	X	$(-4.88, 7.17)$	C_1
concave Max	Y	$(-1.98, 8.65)$	C_1
concave Min	Y	$(1.7, 3.82)$	C_1
convex Max	X	$(3.67, 4.7)$	C_2
convex Min	Y	$(1.14, 2.53)$	C_2
convex Min	X	$(-0.43, 8.51)$	C_1, C_3
convex Max	Y	$(-3.35, 5.23)$	C_1, C_3
convex Min	X	$(-4.52, 3.91)$	C_2, C_3
convex Max	Y	$(1.46, 10.64)$	C_2, C_3
convex Max	X	$(3.11, 7.21)$	C_1, C_2
convex Min	Y	$(3.11, 7.21)$	C_1, C_2
convex Max	X	$(3.62, 4.14)$	C_1, C_2
convex Max	Y	$(3.62, 4.14)$	C_1, C_2

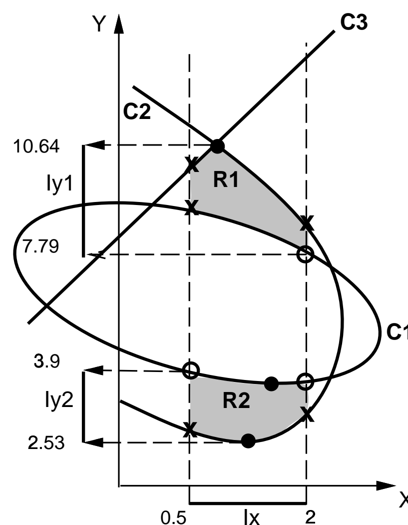


Figure 3.9: *The table on the left side shows local extrema on constraint curves and intersections between constraints for the example represented in the figure. The graphic shows the restricted regions R_1 and R_2 defined by an ellipsis, a parabola and a line constraint. Dots are local extrema and intersections considered by* **simple-propagate**. *While circles indicate intersections of $I_X = [0.5, 2]$ with the constraints resulting in extrema in Y lying on the boundary of the restricted region, crosses show points that are no extrema in Y.*

3.2.4.1 Methods for identifying extremal points of regions

Different methods can be imagined to identify extreme point sets of constraint regions satisfying Definition 3.7. Obviously, they depend on how the feasible constraint region itself is represented. Possible methods are:

- Analytical methods, which determine local extrema directly from the given inequalities.
- Parsing methods, which rely on a discretized representation of the constraint region. First, a constraint region is approximated by 2^k-tree, which is obtained by a carrying out a hierarchical binary decomposition of the solution space ([Sam-Haroud, 1995]). A node of a 2^k-tree is a k-dimensional cubic subregion of the original domain. A simple parsing method can then determine which of the boundary nodes are extremal.

Analytical methods are well-described but can require quite complex computations based on derivatives of the function defining the constraint curve or the constraint surface. Methods based on a discretized constraint representation have the advantage that they are simple and robust. The robustness comes from the fact that only evaluations of the constraint expression are used to determine the discretized representation and that intersections between constraints can be performed logically. The computed extremal nodes, however, depend on the granularity of the discretization and are an approximation of the real points. This is not necessarily a disadvantage because such an approximation also absorbs numerical errors as exposed in the heading section 3.2.1. A more serious problem is the space complexity of

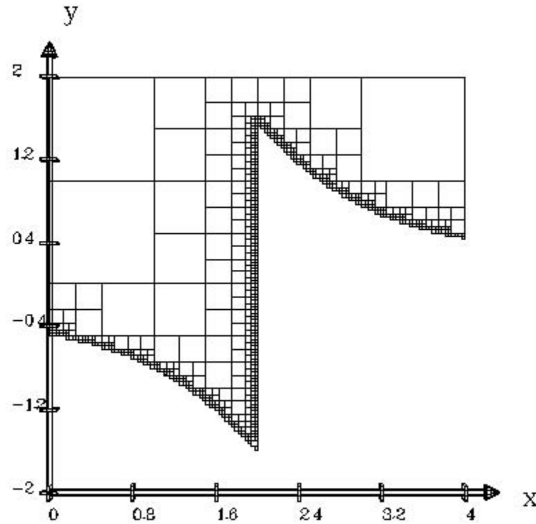


Figure 3.10: *A quadtree representation of the inequality $Y > arctan(1/(X-2)$. Courtesy Claudio Lottaz, AI-Lab, EPFL.*

a discretized constraint representation, which is significantly large in comparison with the analytical approach. In our prototype implementation, we chose the analytical method, first, because many commercial tools exists for analytical computations and second, also in order to investigate the limitations of the analytical representation.

3.2.4.2 Classes of local extrema

Since local extrema appear on the boundary of regions defined by constraints, analytical properties of the constraint curve like continuity and derivability determine the class of extrema. The potential coordinates for an extremum at the boundary of a constraint region $C_{XY} : E(X,Y) \geq 0$ are given by the equation $E(X,Y) = 0$. If possible, $Y = F(X)$ is defined to be the explicit function of $E(X,Y) = 0$ such that $E(X, F(X)) = 0$. For an algebraic curve, which is no function, the first derivative of F is determined for regular points by $F'(X) = -\frac{\partial E(x,y)}{\partial X} / \frac{\partial E(x,y)}{\partial Y}$ (Implicit Function Theorem, Appendix C). The first derivative at the points $(0, \pm 2)$ of a circle constraint $X^2 + Y^2 - 4 = 0$ is $-2X/2Y$ for example. Another property of the constraint region is the gradient at a point (x, y) denoted by $\nabla\ E(x, y)$. It points towards the feasible region defined by $E(X,Y) \geq 0$ (Appendix C).

Analytical conditions ([Douchet and Zwahlen, 1983]) specify where local extrema of a continuous function $Y = F(X)$, F defined over the interval $[a, b]$, occur. The candidate points are:

- at the bounds of the interval $[a, b]$
- at the stationary points of F; e.g. at points where $F'(X) = 0$
- at points within $[a, b]$ where F' is not defined

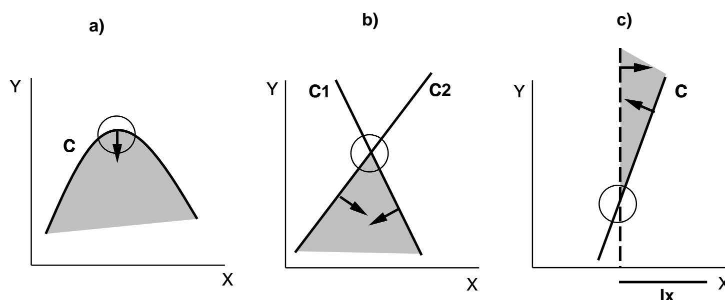


Figure 3.11: *There exist three important classes of local extrema: a) local extremum on constraint curve, b) intersection between two constraints and c) intersection between an interval bound and a constraint.*

Furthermore, if the function itself is not continuous over the interval $[a, b]$, local extrema of constraint regions can occur at a discontinuity. Since a set of constraints may define a constraint region, intersections also belong to the candidates for local extrema because the first derivative at such a point does not exist. Using these analytical properties classes of local extrema can be derived:

1. local extrema on individual constraint curves
2. intersections between constraints
3. intersections between interval bounds and constraints.
4. discontinuities on individual constraint curves
5. singularities on individual constraint curves

Only the first three classes will be considered relevant for our implementation. Although discontinuities and singularities on individual constraint curves can also specify local extrema of constraint regions, they are not considered in the prototype implementation because the analytical computations become exceedingly complex. It is to be emphasized that a discretized representation of continuous constraints would be able to recognize such special local extrema without problem.

Discontinuities on individual constraint curves Discontinuities on individual constraint curves occur at points where either the function value or the limit fails to exist or both exist and are not equal to one another (Appendix C). Examples of functions containing discontinuities are rational functions (the pole in Figure 3.12 b), trigonometric functions (Figure a) or piecewise defined functions (Figure c). The value of the function at a discontinuity has to be determined by computing its right-hand and left-hand limit.

Lemma 3.1 is formulated for restricted regions with a closed boundary. Constraints with a discontinuity in the function defining the constraint boundary specify regions part of which has no border. The constraint $Y - arctan(1/(X - c)) \geq 0$ (Figure 3.13 a), for example has no border along the line $X = c$. Even if this additional constraint defining the missing border would be given, the intersection between $Y - arctan(1/(X - c)) = 0$

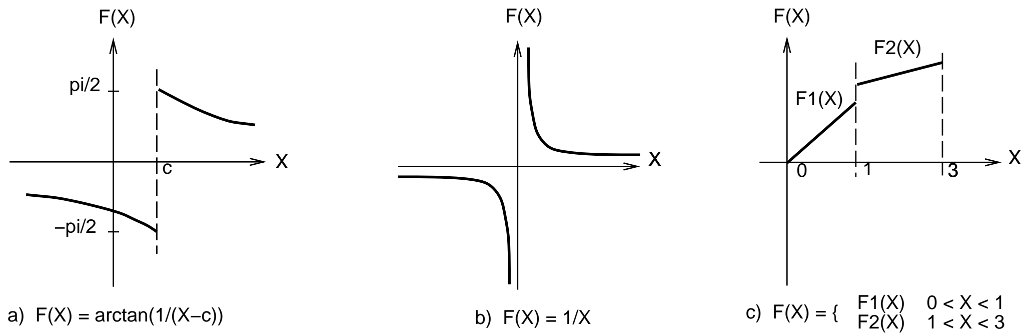


Figure 3.12: *Different types of discontinuities on functions.*

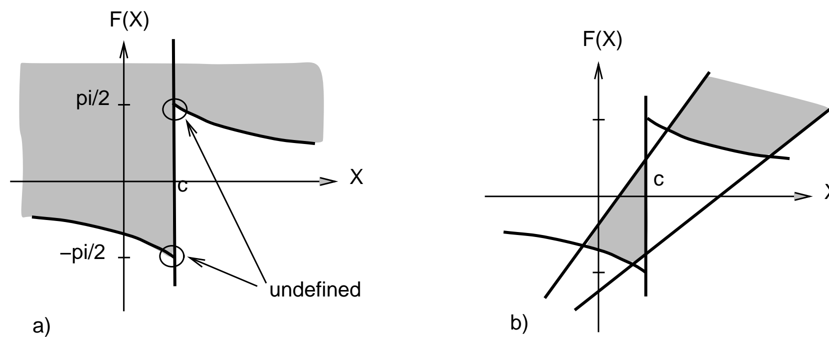


Figure 3.13: *a) A constraint region without continuous boundary b) with a continuous boundary.*

and $X = c$ is still not defined and the function $Y - arctan(1/(X - c)) = 0$ as well as its first derivative have to be evaluated by the left-hand and right-hand limits at $X = c$.

In order to simplify the identification of local extrema, our algorithm is limited to *restricted regions that have a continuous boundary.* As the example of Figure 3.13 b) shows, this restriction does not exclude that a function may be discontinuous over the interval of propagation. Instead it does not compute such discontinuities assuming that they are not part of the restricted regions' border.

It has to be mentioned that a discontinuity can be removed if the right-hand and left-hand limit of the function at the discontinuity have the same value. Such a function can be transformed in a piecewise defined function specifying the function and its value at the discontinuity. An example for such a redefinition is the function $f(X) = sin(X)/X$, which could be defined by

$$F(X) = \begin{cases} sin(X)/X & X \neq 0 \\ 1 & X = 0 \end{cases}$$

Such continuous piecewise defined functions are not considered in the current implementation of the propagation rule **simple-propagate**, they are treated as mixed constraints instead (Section 5.8).

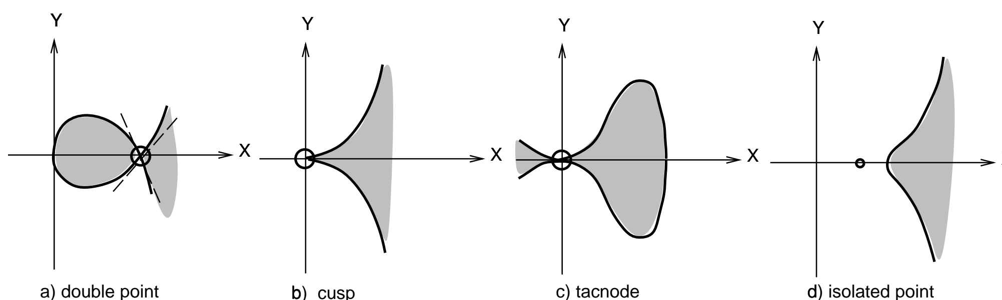


Figure 3.14: *Different types of singularities on constraint curves.*

Singularities on individual constraint curves Another characteristic of constraint curves are *singular points*. They only appear when the constraint boundary $E(X, Y) = 0$ cannot be expressed explicitly as a function at that point but remains an algebraic curve (Implicit Function Theorem, Appendix C). A singular point of a curve is a point at which either several tangents exist (they may be the same) because the curve touches itself or even crosses itself or no tangents exists because the singular point defines an isolated point (for an analytic definition see Appendix C). Examples of singular points are listed in Figure 3.14. Example a) is a Folium of Descartes specified by $X^3 + Y^3 - 3aXY \leq 0$ with $a > 0$ and b) could be the constraint $C_{XY} : X^3 - Y^2 \geq 0$. Only a cusp or an isolated point define local extrema, the other singular points can be neglected anyway. The kind of singular point can be determined by analytical methods however the analysis of the type of local extrema they form requires the analysis of higher degree Taylor developments.

Our algorithm is therefore restricted to *restricted regions in which singular points do not form a local extremum.* Again this restriction does not preclude the existence of singular points over the interval of propagation. Singular points are simply not computed in our implementation. This will lead to errors when a singular point of type cusp or isolated point is part of the boundary of the restricted regions and forms a local extremum there.

As a conclusion, only the first three classes listed at the beginning are accepted as local extrema of constraint regions. Under the assumption that the boundary of the restricted regions do not contain discontinuities nor singular points of type isolated point or cusp, all local extrema can be identified by an analytical method. In an approach based on a discretized constraint representation, extremal points occuring at discontinuities or singularities are not be distinguished from other local extrema. With some care when evaluating constraint expressions at such points, local extrema produced by discontinuities or singularities can be determined more easily by the discretization method.

3.2.4.3 Local extrema on individual constraint curves

The potential candidates for local extrema on an indvidual constraint curve $Y = F(X)$ are determined by the necessary condition:

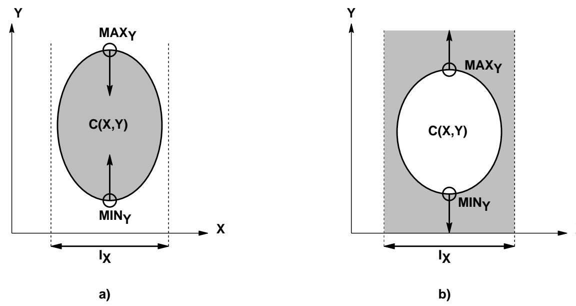


Figure 3.15: *Gradients on the region formed by an ellipsis. In a), the combination of gradient measure and condition indicate a convex maximum and minimum; in b) they indicate a concave maximum and minimum.*

Condition 3.1 (Necessary condition, [Douchet and Zwahlen, 1983]) *If a function $Y = F(X)$, differentiable at x_e, has a local extremum at (x_e, y_e) in Y, $F'(x_e) = 0$.*

Although this condition is not sufficient for the existence of an extremum, it still allows us to derive potential candidates for local extrema. F has an extremum (x_e, y_e) in Y if the following condition is verified:

Condition 3.2 (Sufficient condition, [Douchet and Zwahlen, 1983]) *A sufficient condition for the existence of an extremum at (x_e, y_e) in Y of the function $Y = F(X)$ is that for $I, K \subset \mathbb{R}$, $F : I \rightarrow K$ is at least n times differentiable over I ($n \geq 2$), $F^{(i)}(x_e) = 0, i = 1, \ldots, n-1$, $F^{(i)}$ is continuous in x_e and $F^{(n)}(x_e) \neq 0$ for an even n. Then, $F^{(n)}(x_e) > 0 \Rightarrow (x_e, y_e)$ is a local minimum in Y and $F^{(n)}(x_e) < 0 \Rightarrow (x_e, y_e)$ is a local maximum in Y.*

In order to be able to find local extrema of individual constraint boundaries by analytical methods, we require that the function defining the constraint boundary to be at least twice differentiable over the interval of propagation and the derivatives to be continuous at the local extrema. A more limiting issue would be to tolerate only *smooth* functions; e.g. functions that are continuously differentiable with continuous derivatives over the interval of propagation. This would automatically exlude any discontinuity appearing in derivatives of functions and as a result only discontinuities in first derivatives produced by intersections would be tolerated. This is however more restrictive than necessary because even if discontinuities exist in the derivatives they do not cause problems if they are not on the boundary of a restricted region.

Note that Condition 3.2 only enables us to classify local extrema on constraint curves. Since the gradient points towards the feasible region, the direction of the gradient at a local extremum indicates on which side of the curve the feasible region lies. The condition and gradient direction combined are thus a measure for the convexity of the region in the neighbourhood of the local extremum. Eventually, their combination allows us to classify local extrema as follows:

Condition	Gradient direction	Extremum type
$max_Y(B(R), y_0)$	decreasing Y-values	convex maximum
$max_Y(B(R), y_0)$	increasing Y-values	concave maximum
$min_Y(B(R), y_0)$	decreasing Y-values	concave minimum
$min_Y(B(R), y_0)$	increasing Y-values	convex minimum

In Figure 3.15 a, both extrema are convex whereas in Figure b, they are concave.

3.2.4.4 Intersections between constraints

Intersections between pairs of binary constraints $C^i_{XY} : E^i(X,Y) \geq 0$ and $C^j_{XY} : E^j(X,Y) \geq 0$ create discontinuities in the first derivative of the boundary. Intersections can therefore specify extreme points of a feasible region. Candidate points for the intersection between C^i and C^j are computed by solving the equation system $\{E^i(X,Y) = 0, E^j(X,Y) = 0\}$. The resulting points are each characterized by two distinct tangents and thus also two gradients. The linear combination of the gradients on both constraints at the intersection has to result in a gradient parallel to one of the coordinate axes for a candidate point to be a local extremum. Not all linear combinations are permitted; only those combinations are considered that do not alter the direction of the component gradients (the coefficients of the linear combinations have to be positive or zero). Recall the notation $\nabla\ E(x_e, y_e)$ as gradient of the constraint expression E. N_Y is the Y-component of a vector $\vec{N}$.

Condition 3.3 *An intersection (x_e, y_e) is an extremum in Y if one of the following two cases is satisfied:*

1. *There exists $\beta \geq 0$ and a normal vector $\vec{N}$ parallel to the Y-axis such that*

$$\left\{ \begin{array}{ccc} \nabla\ E^i(x_e,y_e) + \beta * \nabla\ E^j(x_e,y_e) & = & \vec{N} \\ E^i(x_e,y_e) & = & 0 \\ E^j(x_e,y_e) & = & 0 \end{array} \right\}$$

   A special case occurs with $\beta = 0$: $\nabla\ E^i(x_e, y_e)$ is already a normal vector.
   (x_e, y_e) is a convex maximum in Y if $N_Y < 0$ and a convex minimum if $N_Y > 0$.

2. *If $\frac{\partial E^j(x_e,y_e)}{\partial X} = 0$ in the gradient equation above, β does not exist and $\nabla\ E^j(x_e, y_e)$ is already a normal vector.*
   (x_e, y_e) is a convex maximum in Y if $\frac{\partial E^j(x_e,y_e)}{\partial Y} < 0$ and a convex minimum otherwise.

Proof: Consider an intersection $e : (x, y)$ generated by the intersection of two constraints C^i_{XY} and C^j_{XY}. In the neighborhood of (x, y) the feasible region is convex (it can be approximated by the tangents at both constraints in (x, y)). The boundary of the convex region is continuous and its first derivative is continuous in the neighborhood of e but not in e itself. The gradient to the left of e is $\nabla E^i(e)$ and the one to the right is $\nabla E^j(e)$. If

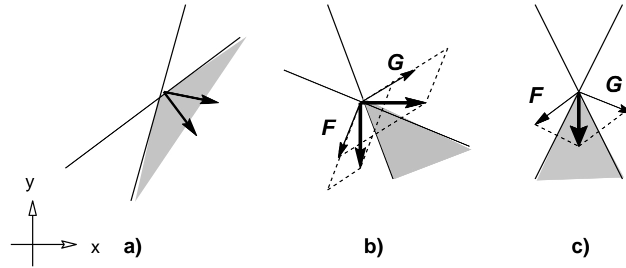


Figure 3.16: *There are three cases of intersections in two dimensions that are candidates for a local extremum in axis Y: a) no extremum, b) a minimum in X and a maximum in Y and c) a maximum in axis Y.*

the gradients $\nabla E^i(e)$ and $\nabla E^j(e)$ enclose a normal, by Lemma C.1 in Appendix C e must be a local extremum. ▲

In the same manner, intersections that are extrema in X are identified. Some examples of constraint intersections are presented in Figure 3.16.

3.2.4.5 Intersections between constraints and interval bounds

Local extrema of a function may also occur on the boundaries of the domain over which the function is defined. This domain corresponds in our case to the interval to be propagated through the constraint. Intersections between a constraint $C_{X_iX_j}$ and an interval bound $I_{k_i} \in L_i$ for variable X_i therefore generate a candidate set for such extrema. They are computed by replacing all possible combinations of interval bounds in the constraint. The type of the resulting extrema is again determined by Condition 3.3.

3.2.4.6 Algorithm for identify-candidates

The propagation of an interval through a total constraint can safely be implemented for a total constraint if the boundary of the restricted regions is continuous, does not contain isolated singular points or cusps and is at least twice differentiable. Under this assumption, the function **identify-candidates** is implemented as shown in Figure 3.17.

3.2.4.7 Filtering relevant extrema

Once the set of local extrema E in an axis (for example Y) has been computed and classified only extrema that satisfy all constraints in C^t_{XY} and whose X-coordinate lies within the interval of propagation I_X, are considered further (function **filter-candidates**). During this filtering process, several special cases have to be treated:

- an extremum composed of a set of points
- redundant extrema
- feasible regions that extend to $+\infty$ or $-\infty$

function identify-candidates(I_X,C^t_{XY})
begin
  $E \leftarrow$ find local extrema on individual constraint curves using the necessary Condition 3.1
    and classify them using Condition 3.2
  $E \leftarrow E \cup$ compute constraint intersections and classify them according to Condition 3.3
  $E \leftarrow E \cup$ compute intersections between constraints and interval bounds and
    classify them according to Condition 3.3
  return E
end

Figure 3.17: *Algorithm for* **identify-candidates**(I_X, C^t_{XY}).

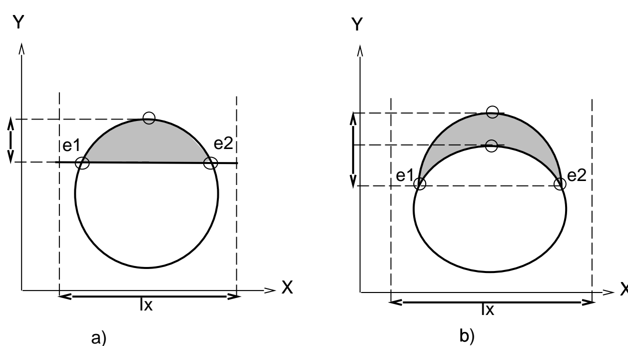


Figure 3.18: *a) Only one minimum has to be considered because both minima come from a constraint that is a line and gives rise to a single minimum. b) Both minima are counted individually because they define different minima.*

- E is empty

Extremum as set of points: An extremum that is a line perpendicular to the axis for which the new label is computed is counted only once (Figure 3.18) because it represents a single extremum according to Definition 3.7.

Redundant extrema: E may contain *redundant* extrema caused by different constraints. By definition, all extrema in a set of redundant extrema share the same coordinates, axis and type but are computed from different constraint sets. Only one extremum of the redundant set must be considered because they all represent the same extremum with respect to the feasible region. The three constraints in Figure 3.19 a) meet at the same point, which is a maximum in Y of the feasible region shaded in grey. If two extrema share the same constraints but differ in their axis or their type, they must both be considered. An equality constraint $E(X,Y) = 0$, for example, is handled as conjunction of two inequalities $E(X,Y) \geq 0, E(X,Y) \leq 0$. Some extremal points of a total constraint involving equalities may at the same time be a local maximum and minimum of the feasible region. In Figure b), $E^1 \leq 0$ and $E^2 \leq 0$ produce a maximum at e_1 and $E^1 \geq 0$ and $E^2 \leq 0$ a minimum in Y at e_2. In Figure c), the outer circle has a convex maximum at e_1 and the inner ellipsis a

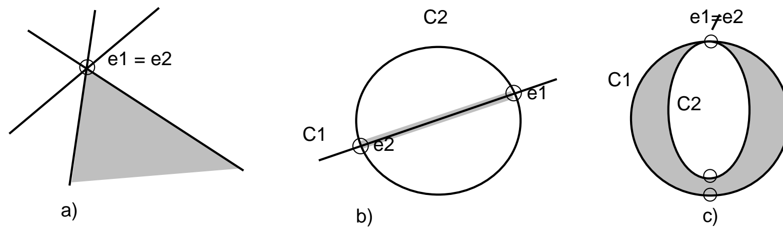


Figure 3.19: *a) Redundant extrema. b) Non-redundant extrema generated by an equalities $C^1_{XY}: E^1 = 0$ and an inequality $C^2_{XY}: E^2 \leq 0$ describing the inner region of a circle. The restricted region is the line segment stretching from e_2 to e_1. c) Non-redundant extrema on the common boundary of two elliptical constraints, C_1 defining the inner part of a circle and C_2 the outer part of an ellipsis. The first extremum determined by C_1 is of type convex maximum and the second determined by C_2 is a concave maximum.*

concave maximum, e_2, at the same point. Both maxima have to be treated separately as they are not of the same type.

Note that the detection of redundant extrema may pose numerical problems due to rounding errors during the computation of the extrema. For a safe implementation, methods from interval analysis or other approximation methods are required to represent the coordinates of local extrema (section 3.2.1).

Unbounded restricted region: In the presence of a restricted region extending to ∞, the set E has to be completed by an extremum at $+\infty$ or $-\infty$ of the according axis. Such a region can be detected by examining the extremum with the largest resp. smallest coordinate. If it is a minimum (resp. a maximum), an extremum at $+\infty$ ($-\infty$) has to be added to E. A similar situation arises when the extremum with the largest coordinate (the minimum with the smallest coordinate) are concave. In Figure 3.15 b) for example, the local extrema are concave indicating an open region, which extends to infinity in both directions.

No extrema exist: Finally, it may happen that no extreme points can be exhibited at all for a given axis, i.e. $E = \emptyset$. This situation occurs when the function forming the boundary of the restricted region has no tangent parallel to the interval to propagate. In Figure 3.20 a), the feasible region is parallel to axis Y and never intersects with I_X. The interval bounds may also intersect the constraint region without producing an extremum, as shown in Figure b). Two cases are possible: either the restricted region extends from $-\infty$ to ∞ or there is no such region at all. This can be tested by intersecting the total constraint with a test-line $T_Y : Y = y$ such that $y \in I_Y$ for a propagation step from X to Y. If there is a feasible point (x, y) satisfying $y \in I_Y$, $(x, y) \in C^t_{XY}$ and $x \in I_X$, the restricted region extends from $-\infty$ to ∞, otherwise, no restricted region exists.

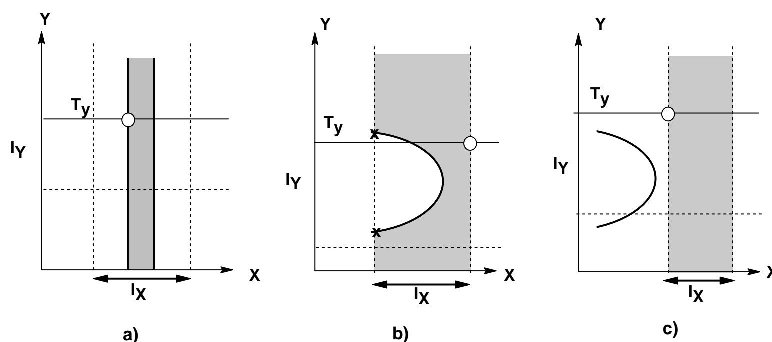


Figure 3.20: *Propagating I_X through the parabola constraint $Y + X^2 \geq 0$ may result in intersections that are no extrema in Y (the intersections are denoted by crosses in Figure b). In this case, it is sufficient to exhibit one point (x, y) such that $x \in I_X$, $y \in I_Y$ and $(x, y) \in C^t_{XY}$ to prove that the restricted region extends from $-\infty$ to ∞. In case a), such a point can be found at the intersection of the constraint with one interval bound of I_Y (T_Y) and in cases b) and c) it is sufficient to test the corners of the box $I_X \times I_Y$.*

3.3 Local consistency for discrete constraints

A refine-operator performing arc-consistency on discrete binary constraints can be found in [Mackworth, 1977a] (**refine-2** in Figure 3.21). Its generalization to k-ary constraints, called NC, was published in the same year [Mackworth, 1977b] (Figure 3.21). This generalization works by applying **refine** to each constraint individually, defining the projection of the tuples onto one axis. Given the number of constraints m and the maximal domain size d, an AC-3 like propagate-algorithm using **refine-2** has a worst time complexity of $\mathcal{O}(m * d^3)$. Subsequent algorithms for arc-consistency in binary CSPs, like AC-4 [Mohr and Henderson, 1986] and AC-6 [Bessière, 1994], are optimal algorithms in $\mathcal{O}(m * d^2)$. [Mohr and Masini, 1988, Bessière and Régin, 1997], generalized AC-4 with a time complexity of $\mathcal{O}(\sum_i |T(C^i)|)$ where $T(C^i)$ is the size of all tuples of a constraint C^i, respectively AC-6 (same time complexity as AC-4 for explicitly given constraints), to k-ary constraints by generating support lists for each variable-value pair and for each constraint individually.

Optimal time complexity has been achieved for discrete constraints by basing arc-consistency on the notion of support. Each variable-value pair has a support list containing at least one variable-value pair that the first pair supports in a constraint. If a value has been deleted from a variable domain, it is queued in order to propagate the effects of deletion. The inconvenience with these optimal time algorithms is that they use additional data-structures (the support-lists) and that the main algorithm loops over a queue containing variable-value pairs whose deletion has to be propagated. Our propagation scheme for continuous variables is rather based on propagating intervals through constraints and not on the propagation of individual values. The support of a discrete value is explicited in the constraint. A support for an interval value in a continuous constraint has to be computed and is not available directly. Furthermore, there exist an infinite number of such supports

```
function refine-2(X, Y, C_XY)
begin
  for each y ∈ L_Y do
    if ∄x ∈ L_X s.t. C_XY(x, y) then
      remove y from L_Y
    fi
  od
  return L_Y
end
```

```
function refine-NC(Y, C_{Z_1...Z_k Y})
begin
  for each y ∈ L_Y do
    if ∄ t ∈ C s.t.∀_i t[Z_i] ∈ L_{Z_i} then
      remove y from L_Y
    fi
  od
  return L_Y
end
```

Figure 3.21: *The refine operators of AC-3 like algorithms for binary constraints (left side) and NC for k-ary constraints(right side).*

in the worst case, because splitting may occur in the support of a value due to a refinement step. In order to keep a uniform queue whose elements consist of variable pairs and the corresponding constraints, we take the AC-3 like propagate-algorithm (in Figure 3.1) as a starting point for discrete propagation and profit from the ideas in continuous propagation when generalizing it to k-ary constraints. We show in the next paragraph, that the existing refine operators do not compute 2-consistency because they miss intersections between constraints sharing at least two variables.

3.3.1 A refine operator for k-ary discrete constraints

A constraint satisfaction problem is formulated as search for a value assignment to all variables that satisfies *all* constraints simultaneously, in other words, the solution must verify the conjunction (the logical AND) of all constraints. We have seen in the previous section, that intersections between binary continuous constraints can form the interval bounds of a refined label. In a discrete binary constraint problem, a binary constraint is usually unique, in the sense that all allowed value pairs are part of the same constraint. If two binary constraints defined over the same pair of variables existed, they could be merged into one by intersecting their tuple sets. Intersections between discrete constraints are of interest as soon as the constraints are of higher arity, because several k-ary constraints with $k > 2$ may involve the same pair of variables but differ in the resting variables.

Let $\mathcal{P}_1$ be a CSP over the variables $\mathcal{V} = \{X, Y, Z, W\}$ each with an initial domain of $\{0, 1, 2\}$ and with the constraints C^1_{XYZ} : ((000) (010) (020) (100) (110) (200)), C^2_{XYW} : ((000) (010) (020) (110) (120) (220)). The projection of these constraints onto X, Y is visualized in Figure 3.22 by binary matrices. It is clear that, given the values $\{0, 1, 2\}$ of Y, 2 cannot be an arc-consistent value for X. No value y can be found for Y such that the value pair $(2, y)$ satisfies both C^1 and C^2. This can only be concluded if all the constraints defined on the same pair of variables are examined simultaneously, i.e. if total constraints

X \ Y	0	1	2
0	1	1	1
1	1	1	0
2	1	0	0

$\cap$

X \ Y	0	1	2
0	1	1	1
1	0	1	1
2	0	0	1

=

X \ Y	0	1	2
0	1	1	1
1	0	1	0
2	0	0	0

Figure 3.22: *The projection of C^1 and C^2 onto X, Y represented as 0-1 matrices. 1 stands for a compatible value pair, 0 for an incompatibility. The intersection of both matrices forms a "triangle" of compatible pairs.*

are considered. Any algorithm generalizing arc-consistency to k-ary constraints by applying arc-consistency to individual constraints will miss intersections between the projections of constraints and therefore result in looser labellings. This remains true for algorithms that first transform a discrete CSP into its binary dual problem where constraints are nodes and links between nodes are variables that must have the same value. The dual problem of $\mathcal{P}_1$ will be $\mathcal{P}_2$ over two variables $\alpha : \{(000)\ (010)\ (020)\ (100)\ (110)\ (200)\}$ and $\beta : \{(000)\ (010)\ (020)\ (110)\ (120)\ (220\}$ and with the constraints $C^3 : X(\alpha) = X(\beta)$ and $C^4 : Y(\alpha) = Y(\beta)$. In this example, there are two different constraints between the same pair of variables that cannot be merged into a unique constraint. Arc-consistency applied on the dual will not detect that value 2 for Y is inconsistent if both constraints C^3 and C^4 are tested individually. This has lead to the definition of interrelational consistency and hyper-k-consistency [Jégou, 1993] in which consistency is defined between constraints and not between variables. We take another approach and introduce total constraints in discrete CSPs and to define local consistency for discrete k-ary CSPs as follows:

Definition 3.9 (Local consistency for k-ary discrete constraints) *A labelling is* locally consistent *if for every X_i, X_j and every value $x_{i_l} \in L_i$, there exists a value $x_{j_l} \in L_j$ such that for every constraint $C_{X_i X_j X_{m_1},\ldots,X_{m_n}}$ involving X_i and X_j and possibly other variables X_{m_i}, there exists a value $x_{m_{il}} \in L_{m_i}, i = 1, \ldots, k-2$ such that $C_{X_i, X_j, X_{m_1},\ldots,X_{m_n}}$ is satisfied for $X_i = x_{i_l}, X_j = x_{j_l}, X_{m_i} = x_{m_{il}}$.*

The motivation behind this definition is that, like in the continuous case, the additional pruning can be obtained easily: Local consistency for a k-ary discrete CSP is achieved by the following steps:

1. collect all constraints involving X_i and X_j (can be precomputed in the total constraint $C^t_{X_i X_j}$)
2. for each constraint project all tuples for which the value of X_{m_i} in the tuple is within its label L_{m_i} onto X_i and X_j
3. remove those values of X_j for which there exists no value of X_i satisfying the projected constraint simultaneously

An algorithm for one propagation step from X_i to X_j is derived in Figure 3.23. The intersection between projections is computed incrementally by verifying that a given value

pair of X_i and X_j is allowed by all constraints of the total constraint (variable *counter* in line 4 to 12). If *counter* equals the number of constraints in the total constraint, there exists a support for the current value of X_j (line 13). Otherwise, the current value of X_j can be removed from its label (line 15). The algorithm finds an arc-consistent label for X in $\mathcal{P}_1$ given the label of Y, $L_Y = \{0, 1, 2\}$ by executing the following steps:

$X = 0$:
  $Y = 0$, $counter = 2$, $support = true$

$X = 1$:
  $Y = 0$, $counter = 1$, $support = false$
  $Y = 1$, $counter = 2$, $support = true$

$X = 2$:
  $Y = 0$, $counter = 1$, $support = false$
  $Y = 1$, $counter = 0$, $support = false$
  $Y = 2$, $counter = 1$, $support = false$
  remove $X = 2$ from L_X

finally: $L_X = \{0, 1\}$

Complexity: The complexity of AC-3 for discrete k-ary constraints is in the order of $\mathcal{O}(d^{k+1}m^2k^2)$ with m the number of constraints, k the arity of the constraints, n the number of variables and d the maximal size of the variable domains.

Proof: The proof is a generalization of the one found in [Mackworth and Freuder, 1985]. We assume that all constraints are k-ary. In the worst case, each variable is connected with each other variable through a constraint, i.e. the hypergraph of the CSP is fully connected. The initial length of the queue is $2 * m * \binom{k}{2}$ because each variable pair is considered. The queue changes its size during propagation as elements are removed and added. We assume further that each element added during propagation is not yet in the queue and that each value removed from a domain is removed in a separate call to **revise** in algorithm 3.1. Entries on the queue are made, if **revise** succeeds. If each variable is connected to $\triangle_i = n - 1$ other variables, a call to **revise** results in adding at most $\triangle_i - 1$ new elements. The number of new elements added to the queue during propagation is $\sum_{i=1}^{n} d * (\triangle_i - 1) = d * n * (n - 2)$. As the number of constraints m is bounded by $\binom{n}{k}$ and assuming that $k \leq n$ we infer that $n * (n - 1) = \frac{m * k!}{(n-2) * \ldots * (n-k+1)} \leq m * k^2$. The number of elements added to the queue is thus bounded by $d * m * k^2$. Furthermore, each call to **revise** takes at most $d^k * |C^t|$ iterations where $|C^t|$ is the number of constraints in the total constraint, which can reach m. The worst time complexity is thus bounded by $d^k * m * (m * k * (k - 1) + d * m * k^2)$ ▲

```
function refine(X_i, X_j, C^t_{X_i X_j})
begin
 1  for each x_{i_l} ∈ L_{X_i} do
 2    support ← false
 3    for each x_{j_l} ∈ L_{X_j} while not(support) do
 4      counter ← 0
 5      for each C ∈ C^t_{X_i X_j} do
 6        found ← false
 7        for each x_{m_{il}} ∈ L_{X_{m_i}} while not(found) do
 8          if C(x_{i_l}, x_{j_l}, x_{m_{1l}}, ..., x_{m_{nl}}) is true then
 9            counter ← counter + 1
10            found ← true
11          fi
12      od od
13      if counter = |C^t| then support ← true fi
14    od
15    if not(support) then remove x_{jl} from L_{X_j} fi
16  od
17  return L_{X_j}
end
```

Figure 3.23: *Algorithm for* **refine**$(X_i, X_j, C^t_{X_i X_j})$ *adapted to k-ary discrete constraints.*

The number $k^2 * d^k * m$ is the optimal time complexity [Cooper, 1989]. Our algorithm has a worst time bound that is $d*m$ larger. In reality, this worst bound of interconnectivity with m as maximal size of the total constraint is seldom reached.

This algorithm can be enhanced if the intersection between constraints is achieved before the actual consistency step. The step achieving local consistency then consists of processing the constraints individually and the optimal time algorithm in [Cooper, 1989] can be used. The preprocessing step is similar to the precomputation of local extrema and intersections for continuous constraints: tuples in a pair of constraints that share two variables can be removed once and for all if their values do not agree. In this case, the complexity of the preprocessing will be in $\mathcal{O}(|T| * |C^t|k^2)$ with $|T| = k * D^k$ under the assumption that the tuples are ordered.

3.4 Local consistency for mixed constraints

In addition to a joint propagation of discrete and continuous constraints, also mixed constraint appear in real-world problems. In section 2.4, a mixed constraint is defined as a relation associating discrete values with real values and vice-versa. Some typical examples

are presented in Figure 3.24. The first two constraints associate intervals or points on the real axis (*Impeller.position*,*V.volume*) with symbolic values (*Impeller*,*Mixer*). The third constraint associates constraint surfaces on the vessel volume with symbolic values and the last one derives the value for an integer variable (*nbPiers*) from two real variables (*typicalSpan* and *length*). In the first two constraints, the real values undergo a discretization in form of intervals that are associated with symbolic values. This indicates that this type of mixed constraint can be transformed into a discrete constraint. The constraint on *nbPiers*, on the other hand, resembles a continuous constraint and can be treated by the continuous refine operator. The constraint on relating the vessel type with its volume is more difficult to handle especially when a continuous variable should be propagated to a discrete one. Several ideas can be exploited:

- Add the continuous constraints during search to the constraint system when the discrete part of the constraint complies with the values of the variables already instantiated
- Apply a kind of consistency by searching within the constraint. Such a refine operator checks for each tuple if the discrete and the continuous part of the constraint added to the constraint system do not result in an inconsistency. If a conflict is detected the entire tuple can be removed from the constraint.

We decided to take into account this type of mixed constraint within the DCSP framework, which is discussed in chapter 5. We thus distinguish three types of mixed constraints:

1. discrete constraints with interval values
2. continuous constraints with discretization operators
3. discrete-continuous relations

3.4.1 Discrete constraints with interval values

Some variables in these constraints have a continuous value domain. In order to be represented as discrete constraint, the domains of such variables have to be discretized. A discretization can be defined by dividing the real axis into regions and landmarks. A landmark is a particular real value. It serves as a precise boundary of a qualitative region (open interval between adjacent landmarks). The landmarks for the vessel volume are indicated in Figure 3.25.

Definition 3.10 (approximate domain for continuous variables) *We define:*

1. I_{base} *as the set of base intervals based on the regions and landmarks.*

2. I_{trans} *as a subset of the power set of* I_{base} *such that two adjacent intervals are merged to form a convex interval.* I_{trans} *forms a lattice. This corresponds to the implementation of an approximate domain as defined by [Benhamou, 1996].*

$\mathbf{C(I, I.position)} :=$
$\{(radialturbine \quad 5)$
$(axialturbine \quad 1.5)$
$(propeller \quad 0.35)$
$(denteddisk \quad [0.15, 0.2])$
$(hellicalribbon \quad [0.15, 0.2])$
$(silversonhighshare \quad [0.15, 0.2])$
$(scaba \quad [0.15, 0.2])$
$(anchorstirrer \quad [0.15, 0.2])\}$

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

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

$\mathbf{C(Bridge.nbPiers, Bridge.length, Bridge.typicalSpan)} :=$
$Bridge.nbPiers = \lceil Bridge.length / Bridge.typicalSpan \rceil$

Figure 3.24: *Three types of mixed constraints.*

3. *A transformation function T_D of a variable domain D and an inverse T_D^{-1} by*

$$\begin{array}{lcl} T_D : I & \rightarrow & I_{trans} \\ T_D^{-1} : I_{trans} & \rightarrow & I \end{array}$$

The transformation function T_I associates a single interval I' from I_{trans} to each convex interval I in $\mathbb{I}$ such that I' is the intersection of all elements in I_{trans} that encompass I.

In Figure 3.25, I is the set of all possible intervals in $[0, 1000]$ and I_{trans} is for example the set

$$\{[0,0], (0,1), [1,1], (1,100), [100,100], (100,150), [150,150], (150,1000), [1000,1000], [0,1), [1,100], \ldots\}$$

Other versions of I_{trans} are possible depending on the kind of base intervals taken from the landmark representation like for example $I_{D_{V.volume}} = \{[0,1], [1,100], [100,150], [150,1000], [0,100], \ldots\}$. Using $T_{D_{V.volume}}$, the constraint **C(M,V.volume)** will then be represented internally by the tuples $((reactor \quad [0,1])(reactor \quad [1,100])(storagetank \quad [0,1])(storagetank \quad [1,100])...)$. The

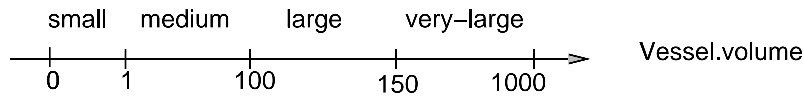


Figure 3.25: *Definition of landmarks for the vessel volume.*

advantage of this transformation is that this type of mixed constraint is representable as discrete constraint and that algorithm 3.23 can be applied directly.

An interval of the label for the vessel volume in the mixed constraint in Figure 3.24, for example, can be transformed according to the function $T_{D_{V.volume}}$:$D_{V.volume} \rightarrow I_{D_{V.volume}}$. We also define the function $T^{-1} = Identity$. Such a transformation function and its inverse are associated to each mixed constraint for each continuous variable. Before propagating a continuous interval through such a constraint, the interval has to be transformed according to T and after the propagation it has to be transformed back into its continuous representation according to T^{-1} because its value might be propagated through continuous constraints. The additional cost of $\mathcal{O}(k * d)$, where k is the number of continuous variables in the constraint and d the maximal number of landmarks minus one, due to the transformation steps before and after a call to the refine operator are negligible with respect to the complexity of **refine**.

The way this transformation function is chosen depends on the definition of the constraints. Either there is a transformation function for each constraint separately or there exists a set of landmarks that are valid for all mixed constraints. In the mixer example, there exists global transformation function: $T_{D_{V.volume}}$, which can be used for the transformation of the label of *V.volume* in all constraints.

3.4.2 Continuous constraints using 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. Continuous consistency algorithms are based on continuous functions with a continuous boundary. These operators, however, define discontinuities as shown in Figure 3.26.

One way to treat such equations is to approximate the equation by continuous constraints according to the definitions given in Table 3.1. The result of one propagation step for the integer variable will then be rounded inwardly to the next integer by the transformation function $T([a, b]) = [\lceil a \rceil, \lfloor b \rfloor]$ with $a \leq b$ and $a, b \in \mathbb{R}$ and $T^{-1}([a, b]) = [a, b]$.

Propagating $typicalSpan = [35, 60]$ through the constraint on $nbPiers$ in Figure 3.24 results in the interval $[2.5, 5.2857]$ for $nbPiers$, which is rounded to $[3, 5]$. Search methods then compute the solutions $\{nbPiers = 3, typicalSpan = (50, 75]\}$,$\{nbPiers = 4, typicalSpan = (37.5, 50]\}$ and $\{nbPiers = 5, typicalSpan = (30, 37.5]\}$.

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	$i = trunc(r/q)$

Table 3.1: *A list of operators that define discontinuous functions. The operand* r *is a real,* i*,*p *and* q *are integers.*

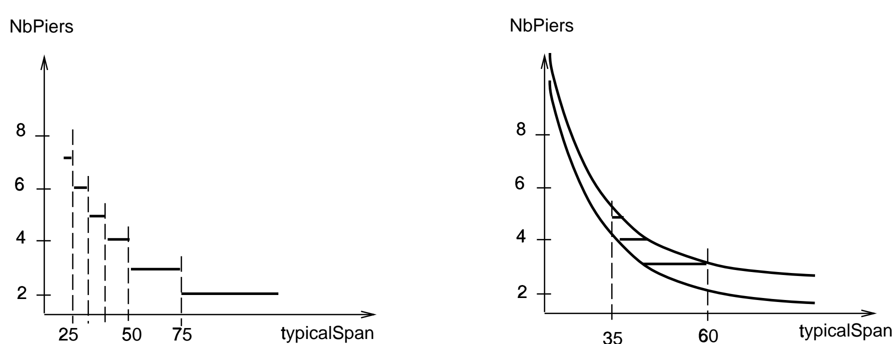


Figure 3.26: *The constraint* $nbPiers = \lceil length/typicalSpan \rceil$ *with* $length = 150m$ *is shown on the left and its approximation by two inequalities on the right side. Within the propagated interval of* $[35, 60]$ *for typical span, several solutions exist:* $nbPiers = 3, 4, 5$.

3.4.3 Completeness and soundness

The completeness of the labellings computed in mixed CSPs with the discrete (Section 3.3.1 and the continuous refine operator (Section 3.2.3) is guaranteed, because each of the operators produces complete results and the communication of results between operators is an approximation of the original results. Thus no solutions can be lost.

It is, however, obvious that the approximation of results communicated between the discrete and continuous refine operator affect the local soundness property of the overall algorithm: approximations of results happen as soon as an open interval computed by the discrete propagation rule has to be propagated within the continuous operator. The open interval is mapped into a closed one by the transformation function because the continuous operator presented can only handle inequalities of type $\leq, \geq$ and thereby equalities. It is not able to handle strict inequalities, which topologically represent the inner part of a region without its boundary. As a consequence, a conflicting, locally unsound value that appears exactly at a landmark remain undetected. Consider again the constraint $\mathbf{C}(\mathbf{M}, \mathbf{V.volume})$ defined this time by $\{(reactor\ [0, 100)\}$ and a continuous constraint *V.volume* ≥ 100 the continuous refine operator will not detect that *V.volume* $= 100$ is not a locally consistent value because the result for *V.volume* computed by the discrete operator is approximated

by $T_{V.Volume}$ with $V.volume = [0, 100]$.

3.5 Summary

Consistency is an important concept in constraint satisfaction. This is especially true in continuous domains where it is impossible to search for solutions using enumerative algorithms. In this chapter, we consider propagate-algorithms [Davis E., 1987] computing local consistency (2-consistency).

The integration of local consistency methods for discrete and continuous constraints into a fix-point algorithm is achieved by specifying refine operators for each constraint type. Additionally, a transformation is applied to labels if a continuous variable takes part in a discrete constraint or a discrete variable in a continuous constraint. The goal of such a transformation is to communicate values between constraints of different type sharing variables. In some cases such a transformation leads to an approximation of the original variable domain, because not all values of the original domain can be represented in a second domain. Since the refine operators are using approximate variable domains, some locally unsound values may remain in the labellings after the application of the refine operator.

Faltings shows in [Faltings, 1994], that some of the very negative results of applying propagation algorithms to continuous domains are due to the way in which the refine operator for the algorithm is formulated. He presents a new operator ensuring 2-consistency for binary constraints over continuous variables. This new operator rule takes into account intersections between constraints defined on the same pair of variables thus providing a tighter pruning than algorithms propagating individual constraints. In the current implementation, we use an analytical representation of constraints and do not compute discontinuities or singularities. In the discussion of the actual implementation, it becomes clear that both, an analytical representation and a discretized representation of constraints have their advantages.

Finally, a discrete refine operator is presented for k-ary constraints, which profits from the ideas of local consistency in continuous domains. More precisely, the discrete operator takes into account intersections in the projection of discrete tuples onto a pair of variables thus achieving 2-consistency for this class of constraints.

The extension to ternary numeric constraints, which we present in the next chapter, is based on the same ideas and also uses topological arguments in order to achieve local consistency in three-dimensional spaces.

Chapter 4

Local consistency for ternary numeric constraints

In this chapter, we extend the binary refine operator to ternary constraints. The chapter is organized as follows:

- We first show why a propagation rule for ternary numeric constraints is more useful in practical applications than its binary counterpart.
- The definitions are then extended to ternary numeric CSPs.
- A refine operator for ternary constraints defined in the same three-dimensional space with an example and a correctness proof.
- Implementation.

4.1 A refine operator for ternary numeric constraints

The propagation rule for ternary numeric constraints, which will be described in the following, profits in many ways from the ideas exhibited in the chapter on local consistency over binary numeric constraints. Definitions like total constraint, local consistency and local extrema are generalized in a straightforward manner to ternary CSPs. Instead of investigating two-dimensional shapes, we move into $\mathbb{R}^3$, in which a constraint C_{XYZ} represents a volume. The axes of the Euclidean space thus defined are X,Y and Z.

4.1.1 Local consistency over k-ary constraints ($k \geq 3$)

Two approaches are possible when devising an algorithm for local consistency over k-ary numeric constraints:

1. The algorithm enforces local consistency directly on k-ary constraints for any k.
2. The algorithm enforces local consistency on a transformed system consisting of constraints whose arity is at most three.

Examples for the first approach are Newton and its successor Numerica ([Benhamou et al., 1994], [Van Hentenryck et al., 1995]) in which a weaker form of local consistency called box-consistency is used directly on the given complex constraints. A constraint is transformed into an interval function and its projection onto a given variable allows for narrowing down the interval of the variable by finding the left-most and right-most zeros of the function. Since the binary propagation rule discussed in chapter 3 has a remarkable pruning effect due to the idea of using total constraints, our first idea was to generalize the rule to at least ternary constraints and to evaluate additional complications of this stepwise generalization. In fact, the ternary propagation rule has its limitations, which are likely to become even more hampering in a further generalization to higher dimensions. Also, identifying local extrema will become increasingly complicated. For this reason, we chose a second approach that relies on a transformation of the initial system into a set of maximal ternary constraints and then achieves local consistency on the transformed set.

Any k-ary constraint $C_{X_1 \dots X_k}$ can be transformed into a set of ternary constraints as follows ([Sam-Haroud, 1995]):

i iteratively replace each subexpression $X_i \bigodot X_j$ with some binary operator $\bigodot = \{+, -, *, \dots\}$ in $C_{X_1 \dots X_k}$ by a new variable X_{n+1}

ii add a new ternary equality constraint $X_{n+1} = X_i \bigodot X_j$.

The process stops when C itself becomes ternary. Its complexity is bounded by $\mathcal{O}(m)$ where m is the number of operators in $C_{X_1 \dots X_k}$. Since the transformation is based on symbolic manipulation, no information is lost in the description of the solution space. It is however possible that a locally consistent labelling produced for a transformed system suffers from larger rounding errors due to the newly introduced equality constraints. It has also been argued in [Benhamou et al., 1994] that the decomposition into basic constraints used in some interval-based algorithms slows down the convergence of the algorithm and result in weaker pruning. The goal of such a transformation is to guarantee certain properties on the constraints using a restricted set of basic operators. This is different from our approach that requires ternary constraints as input without specifying what kind of operator should be used. It is still an open research issue how such a transformation influences the locally consistent labelling.

4.1.2 Extension of definitions to ternary numeric CSPs

It has often been argued in the discrete CSP community that determining a locally consistent labelling for k-ary discrete constraints can be reduced to the problem of finding a labelling of the binary CSP induced by the dual constraint graph. This is in general more difficult for numeric CSPs because in such a transformation the nodes of the dual graph are the constraints and the new constraints establish a relationship between the values of two identical variables located in different nodes. Finding these relationships between values does not seem a trivial task. The approach we take consists of computing locally consistent labellings directly on the region defined by ternary constraints. In addition,

the phenomenon of implicit cycles also appears in ternary constraint systems: if several ternary constraints are defined on the same pair of variables (they might differ in their third variable) and they are considered individually by a propagation algorithm, the effects of propagating a single constraint over one of the two variables induces the reconsideration of all other constraints in the total constraint and so on. Again, such set of constraints should be treated simultaneously and thus belongs to the same total constraint.

Definition 4.1 (Total constraint) *A total constraint is a set of bounded* feasible regions $C^t_{XY} = \{Q_1, \ldots, Q_k\}$ *containing exactly those combinations of values* (x, y) *for* X *and* Y *that are consistent with all constraints* C_{XYZ_i} *for any* i.

A total constraint C^t_{XY} consists of all ternary constraints defined over the variables X, Y resulting in a constraint set $\bigcup_{i=1}^{n}\{C_{XYZ_i}\}$. Note, that the restricted regions of a feasible region Q are now described by $R(Q, \{I_X, I_Z\})$ and are those connected subregions of Q the X-coordinates of which are entirely contained within an interval I_X and the Z-coordinates within an interval of the label L_Z.

Intuitively, we aim at finding a propagation rule that has the effect of producing locally consistent labels on *the binary projection of ternary constraints* that are part of the total constraint. According to this idea, the local consistency definition 3.4 is extended to systems of ternary constraints as follows:

Definition 4.2 (Local consistency for ternary constraint systems) *A labelling is* locally consistent *if for every* X_i, X_j *and every value* $x_i \in L_i$*, there exists a value* $x_j \in L_j$ *such that for every constraint* $C_{X_iX_jX_k}$ *involving* X_i *and* X_j *and possibly another variable* X_k*, there exists a value* $x_k \in L_k$ *such that* C_{X_i,X_j,X_k} *is satisfied for* $X_i = x_i, X_j = x_j, X_k = x_k$*:*

$$
\begin{aligned}
(\forall i, j) \quad & (\forall x_i \in L_i)(\exists x_j \in L_j) \\
& (\forall C_{X_iX_jX_k})(\exists x_k \in L_k) C_{X_iX_jX_k}(x_i, x_j, x_k)
\end{aligned}
$$

Note that this definition is different from others (e.g., [Davis E., 1987]) in that it requires that for each single value x_j there exists at least one value for X_i such that this value pair satisfies *all* constraints over the variables X_i, X_j. This ensures that the locally consistent labelling of the ternary system is at least as tight as the locally consistent labelling of its binary projection.

Similar to the binary propagation rule, we would like to find a means of deciding when an entire three-dimensional region has been considered only by counting its "extreme points". In order to motivate the definition of extreme points in a three-dimensional space, we outline the underlying mechanism on which the proof for a ternary propagation rule is based. In the binary case, we are able to determine the number of connected intervals in the intersection of a given value $X_j = x_j$ and the feasible regions defined by the binary constraints. Letting a line $L : X_j = x_j$ sweep over the given intervals for X_j, we detect if

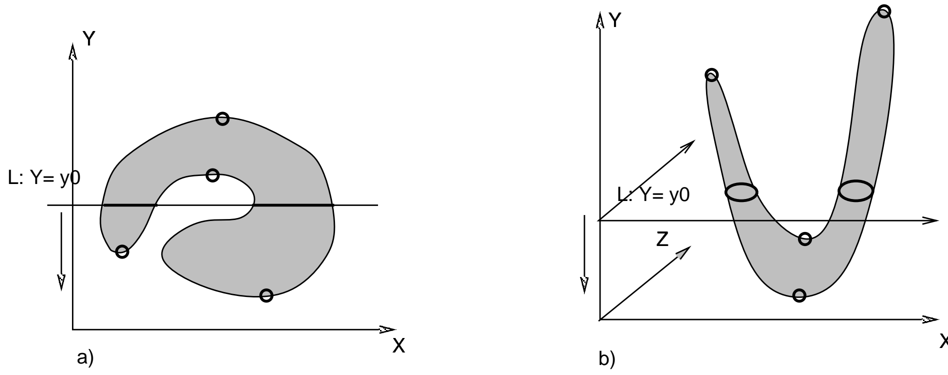


Figure 4.1: *a) In 2 dimensions, $\alpha_Y(R, y_0)$ represents the number of connected intervals in R. In 3 dimensions, we would like to determine the number of connected regions in the XZ-plane for a given y_0 in a similar manner. The circles mark points at which this number changes.*

there is at least one value (or one interval) for X_i thus verifying the consistency condition for binary numeric constraints. This method can be directly adapted to regions defined by ternary constraints. L now represents a sweep plane and the intersection between L and the feasible regions are now at least binary connected regions and no longer intervals. In Figure 4.1 a), there are two connected intervals and in Figure b) $R \in \mathbb{R}^3$ is a three-dimensional region in space X, Y, Z and $L : Y = y$ represents a sweep plane parallel to X, Z moving from larger Y-coordinates to smaller ones. It can be observed that the number of connected regions appearing in the sweep plane will only change at points on the surface of the three-dimensional region that have a tangent plane parallel to the sweep plane. The tangent plane of a point (x_e, y_e, z_e) on the surface $B(R)$ is defined by the tangents situated at the intersection of $B(R)$ with the planes defined by two coordinates of the point, for example $X = x_e$ and $Z = z_e$ (Figure 4.2). At a point that has a tangent plane parallel to the sweep plane, the value of the function defining the surface does not change in the neighborhood of that point. It is at such a point that a new region appears or disappears in the sweep plane and existing regions are merged or disconnected. Points whose tangent plane is parallel to the sweep plane, i.e. which have a normal parallel to one of the coordinate axes, are called *stationary points.*

Definition 4.3 (Stationary point) *Let I be a connected subset of a closed set $R \subseteq \mathbb{R}^3$. I is stationary in Y of R if and only if for any point $e_j \in I$, both sets $S^j_{XY} = \{(x, y) \mid (x, y, z_{e_j}) \in R\}$ and $S^j_{ZY} = \{(z, y) \mid (x_{e_j}, y, z) \in R\}$ have a local extremum in $Y = y_{e_j}$.*

Stationary points not only include local minima and maxima but also saddle points like the one shown in Figure 4.2. Each z_{e_j} and x_{e_j} of a stationary point e_j defines a "cut" through the space defined by R, called a *slice*: S^j_{XY} involving X and Y and S^j_{ZY} involving Z and Y. The definition of stationary points also includes sets of points that are all identical with respect to the extremum type in both slices. Typically, a curve or even a surface may form a stationary point set. A stationary point e_j of a region is classified according to the types

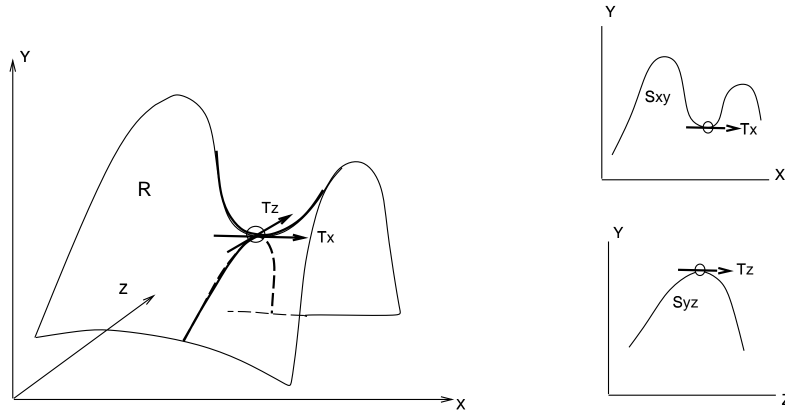


Figure 4.2: *Cutting through a region along $X = x_e$ and $Z = z_e$ with e being a stationary point results in two sets S_{XY} and S_{ZY}. S_{XY} has a horizontal tangent T_X in (x_e, y_e) and S_{ZY} has a horizontal tangent T_Z in (z_e, y_e). The local extremum in S_{XY} is a minimum in Y and the second in S_{ZY} a maximum in Y.*

of local extrema in both slices S^j_{XY} and S^j_{ZY}. To compute the number of connected regions in the intersection with the sweep plane, all stationary points must be found and classified[1]. These points again do not only appear on individual constraint surfaces and intersections between surfaces but they may also form in the intersection plane of an interval boundary.

4.1.2.1 Classification of stationary points

Let R be a three-dimensional closed region and e_j a stationary point to which the slices S^j_{XY} and S^j_{ZY} are associated. When propagating an interval in X to the Y-axis, the S^j_{XY} is called main slice because it is parallel to both axes involved in the propagation. In order to classify a stationary point e_j, only local information obtained from the slices S^j_{XY} and S^j_{ZY} is used. By definition the stationary point has a local extremum at the boundary of both slices denoted by $ext_Y(B(S^j_{XY}), y_{e_j})$ respectively $ext_Y(B(S^j_{ZY}), y_{e_j})$. The type of extremum in both slices determines the type of the stationary point. Altogether, there exist $2 * 2 * 2 = 8$ combinations (maximum or minimum in both directions combined with either side of the boundary being the feasible region) as reported in Figure 4.3. In this Figure, the main slice $S_M = S^j_{XY}$ is indicated by a solid and $S = S^j_{ZY}$ by a dotted line. Two of these eight combinations are simple rotations of another combination and are therefore discarded. There remain six different types of stationary points to be distinguished in order to interpret ternary regions correctly. This classification is justified by the fact that all the six types will have a different effect on the intersection between the feasible regions and the sweep plane (see Section 4.2.3 on the correctness of the propagation rule). The names to distinguish these types have been chosen according to the following convention: A *max* is a maximum in both slices, a *min* is a minimum in both slices and a *saddle* has

[1] Similar theories exist in topology, in which an integral or a topological index over a whole surface is computed [Milnor, 1963], solely based on information at critical points of the surface [Fulton, 1995]

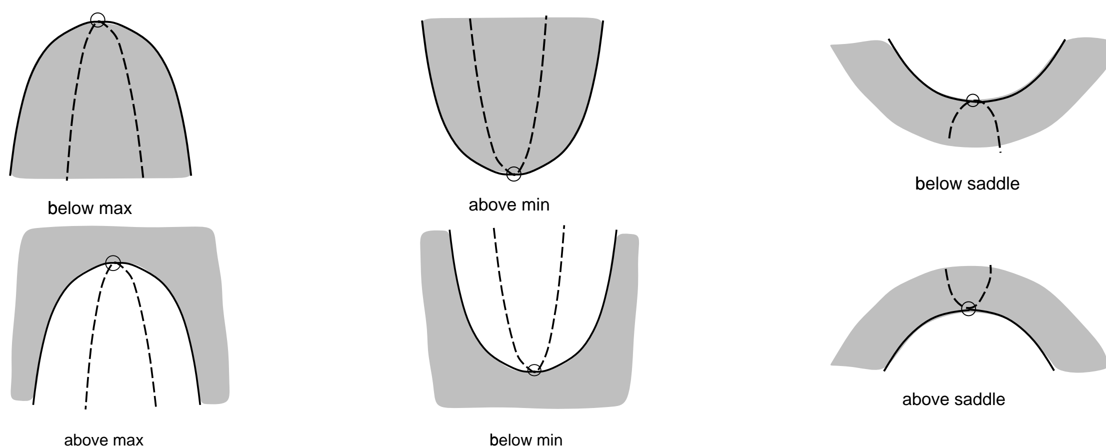


Figure 4.3: *Classification of stationary points.*

different extrema in both slices. If the feasible region (shaded in Figure 4.3) extends above the stationary point it is termed *above* otherwise *below*.

Case 1: below max e_j is a maximum in both slices: $max_Y(B(S_M), y_{e_j})$ and $max_Y(B(S), y_{e_j})$ and the feasible region lies below e_j.

Case 2: above min e_j is a minimum in both slices: $min_Y(B(S_M), y_{e_j})$ and $min_Y(B(S), y_{e_j})$ and the feasible region lies above e_j.

Case 3: below saddle e_j is a maximum in the main slice XY and a minimum in the secondary slice ZY or a minimum in the main slice and a maximum in the secondary slice: $ext_Y(B(S_M), y_{e_j}) \neq ext_Y(B(S), y_{e_j})$ and the feasible region lies below e_j.

Case 4: above saddle e_j is a maximum in the main slice XY and a minimum in the secondary slice ZY or a minimum in the main slice and a maximum in the secondary slice: $ext_Y(B(S_M), y_{e_j}) \neq ext_Y(B(S), y_{e_j})$ and the feasible region lies above e_j.

Case 5: above max e_j is a maximum in the main slice XY as well as a maximum in the other slice ZY: $max_Y(B(S_M), y_{e_j})$ and $max_Y(B(S), e_j)$ and the feasible region lies above e_j.

Case 6: below min e_j is a minimum in the main slice XY, as well as a minimum in the other slice ZY: $min_Y(B(S_M), y_{e_j})$ and $min_Y(B(S), y_{e_j})$ and the feasible region lies below e_j.

4.1.2.2 Classification of three-dimensional regions

Since the form of feasible regions that appear in three dimensions becomes more complex, we have to distinguish between different types of regions with respect to topological

considerations (Appendix B).

Remember that a region in our case is the set of all points connected to a given point; i.e. it is always path-connected. If a feasible space consists of several regions, those are disconnected by definition. Path-connectedness does not exclude the existence of holes in a region. A *hole* defines the complement of a closed bounded region as feasible region (the outside of that region). Different types of holes exist: *Cavities* are holes that are completely surrounded by a feasible region, whereas channels pierce a feasible region. We need to distinguish between two types of regions:

1. *simple regions* are regions containing no holes
2. (path-connected) regions are regions that can contain one or more holes

The path-connected regions may contain holes in form of cavities or channels. An example for such a region is a "swiss cheese" or a "sieve". If we do not precise that a region is simple, it may contain holes.

Furthermore, as for the binary propagation rule, we consider connected regions with a boundary that has kinks and corners only at intersections of different constraints forming the region but has no other irregularity (discontinuity or singularity) on its boundary.

4.2 Refine operator for a single third variable

We first consider the case of propagating the label of X to Y through a set of constraints $C^t_{XY} = \bigcup_i C^i_{XYZ}$, a total constraint involving a single third variable Z. The consistency condition of Definition 4.2 for the label L_Y is now:

$$(\forall y \in L_Y)(\exists x \in L_X)(\exists z \in L_Z)C_{XYZ}(x, y, z)$$

Similar to the binary case, we would like to find locally consistent labellings for a three-dimensional region only by taking into account its stationary points. In the following, a refine operator is devised that propagates a single interval I_X through the total constraintC^t_{XY} in order to find a locally consistent label for Y. For the moment, we assume that the total constraint forms a set of simple regions.

4.2.1 Propagating a single interval through simple regions

In order to know when a feasible region has been completely considered, only the stationary points of the restricted regions need to be determined, classified and counted similar to the binary propagation rule. The procedure **simple-propagate**$(I_X,C^t_{XY},\{I_Z\})$ computes the intervals for Y that are in the restricted regions where $x \in I_x$ and $z \in I_Z$. Each of the restricted regions $R_i \in R(C^t_{XY}, \{I_X, I_Z\})$ projected on the Y-axis defines a single continuous interval of locally consistent values for Y given as

$$I^i_Y = [abovemin\{y|\exists(x, y) \in R_i\} \dots belowmax\{y|\exists(x, y) \in R_i\}]$$

From all types of stationary points these are the only ones at which a region completely vanishes (above min) or is being created (below max). This observation does however not imply that only stationary points of type above min and below max have to be considered. The other types rather represent intermediate points allowing for a correct interpretation of the restricted regions. The region represented in Figure 4.1 b, for example, has two below max but only one above min. It is the saddle point that compensates for the missing second minimum. In the example of Figure 4.5, the set $R(C^t_{XY}, \{I_X, I_Z\})$ has two regions R_1 and R_2 that define two intervals $I_{Y_1} = I_{Y_2}$ with values for Y that are locally consistent with $x \in I_X$ and for which there exists a value $z \in I_Z$. Provided that the boundary of restricted regions does not contain discontinuities nor singularities (similar assumptions as for binary constraints), the stationary points in the boundaries of the restricted regions $R_i \in R(C^t_{XY}, \{I_X, I_Z\})$ fall at:

1. Local extrema and saddle points of an individual constraint surface lying within I_X and I_Z.
2. Intersections between two respectively three constraint surfaces lying within I_X and I_Z.
3. Local extrema and intersections in the boundaries of I_X and I_Z.

The saddle point in Figure 4.2 is an example of the first category. In the same graph, there are two local maxima of the constraint surface in the interval bounds of X. Intersections between constraint surfaces that form stationary points are shown in Figure 4.5.

The propagation rule **simple-propagate** for ternary constraints is applied to each combination of intervals in L_X and L_Z (Algorithm 4.4, lines 2-4). It executes exactly the same steps as the one for binary constraints presented in Algorithm 3.8:

a) finding the candidates in form of stationary points and classifying them (function **identify-candidates**),

b) filtering relevant points that satisfy all constraints of C^t_{XY}, whose X-coordinate lies within I_X and the Z-coordinate within I_Z (function **filter-candidates**) and ordering the remaining points with decreasing Y-values,

c) computing the locally consistent intervals for L_Y (function **compute-intervals**) on the ordered set.

The only difference lies in the fact that there are more distinct types of stationary points to be expected (not only minima and maxima as in the binary propagation rule) and α is incremented at stationary point of type below max, below min, above saddle and decremented at those of type above max, above min, below saddle (lines 16,18 of the algorithm). As we will prove later, **simple-propagate** is correct for a set of simple regions; i.e. regions not containing holes.

```
function refine(X, Y, C_XY)
begin
 1  I_Y ← {}
 2  for all combinations (I_X, I_Z) with I_X ∈ L_X and I_Z ∈ L_Z do
 3    I_Y ← I_Y ∪ simple-propagate(I_X, C_XY, {I_Z})
 4  od
 5  IU ← union of all intervals in I_Y where overlapping intervals
    are merged into single convex intervals
 6  return IU
end

function simple-propagate(I_X, C^t_XY, {I_Z})
begin
 7  E ← identify-candidates(I_X, C^t_XY, {I_Z})
 8  E ← filter-candidates(E, I_X, C^t_XY, {I_Z})
 9  L_Y ← compute-intervals(E)
10  return L_Y
end

function compute-intervals(E)
begin
11  I_Y ← {}
12  α ← 0
13  for each e ∈ E do
14    if e is of type below max, below min, above saddle then
15      α ← α + 1
16    elif e is of type above max above min, below saddle then
17      α ← α − 1
18    fi
19    if α has changed from 0 to 1 then
20      y_max = Y-coordinate(e)
21    elif α has changed from 1 to 0 then
22      y_min = Y-coordinate(e)
23      I_Y ← I_Y ∪ {[y_min, y_max]}
24    fi
25  od
26  return I_Y
end
```

Figure 4.4: *A propagation rule for a single third variable.*

4.2.2 Example

In order to illustrate the algorithm, a simple example of the intersection between a parabolic surface rising in X_3 and a sphere is given (Figure 4.6):

$$C_1 \quad := \quad X_1^2 + 1/2 * X_2 + 2 * (X_3 - 6) \geq 0$$

$$C_2 \quad := \quad X_1^2 + X_2^2 + X_3^2 - 25 \leq 0$$

In the following all coordinate values are rounded to 2 decimals. The sphere has a local maximum in X_3 at $(0, 0, 5)$ and a local minimum at $(0, 0, -5)$. There are four intersections that are also local extrema in X_3: below max $(1.54, 1/4, 19/4)$, below max $(-1.54, 1/4, 19/4)$, above min $(4.17, 1/4, -11/4)$, above min $(-4.17, 1/4, -11/4)$. Only the four intersections are part of the feasible region. To propagate X_1 to X_3 with $L_{X_i} = \{[-10..10]\}, i = 1, 2, 3$, we have to consider the restriction of the feasible region $Q = R(C^t_{XY}, \{I_{X_1} = [-10, 10], I_{X_2} = [-10, 10]\})$. This restriction does not produce any new stationary points. The four intersections are sorted in decreasing order and the propagation rule is applied. The list of stationary points is given in Figure 4.5 with the index $\alpha(Q, x_{3e})$ for each point e together with an illustration of the slice at $X_2 = 1/4$. From this list the resulting interval for X_3 is computed: $I_{X_3} = \{[-2.75, 4.75]\}$. A comparison between our propagation rule and a refine operator similar to Davis' propagating individual constraints[2] results in:

label	*local consistency*	*Davis*
L_{X_1}	$\{[-5, -1.65][1.65, 5]\}$	$\{[-5, 5]\}$
L_{X_2}	$\{[-3.5, 4]\}$	$\{[-5, 5]\}$
L_{X_3}	$\{[-2.75, 4.75]\}$	$\{[-5, 5]\}$

Using Davis' operator, which propagates the constraints individually and which does not take into account intersections between constraints, the intervals cannot be reduced beyond the projection of an individual constraint onto each axis. Furthermore, the necessary splitting of initial labels into monotonic and continuous subparts results into 2^3 applications of the propagation rule instead of a single one.

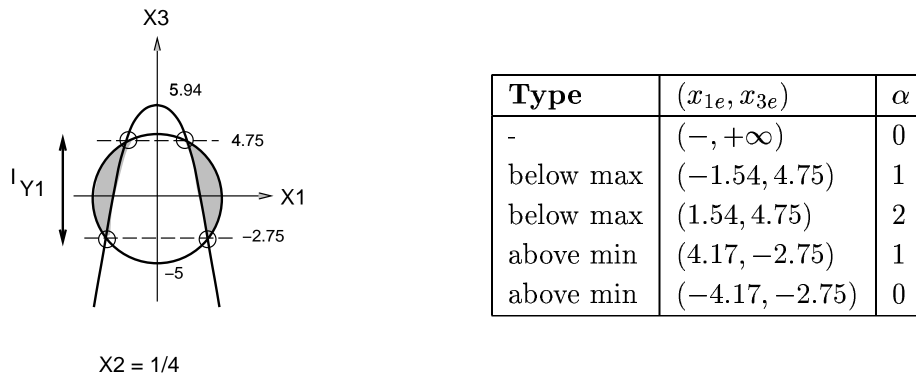
Imagine now that the constraint of the parabolic surface is slightly changed into

$$C_1 := X_1^2 + 1/2 * X_2 + 2 * (X_3 - 3) \geq 0$$

shown in Figure 4.6 such that the local maximum in X_3 of the sphere at $(0, 0, 5)$ becomes part of the feasible region and only three intersections remain: above min $(-3.58, 1/4, -3.48)$, above min $(3.58, 1/4, -3.48)$ and a saddle point above saddle $(0, 4.65, 1.84)$ shown in Figure 4.7. Again the intersections with the bounds of the label of X_2 do not result in further stationary points. After sorting the points in decreasing Y-values, the locally consistent interval for the propagation step of $I_{X_1} = [-10, 10]$ to X_3 is $I_{X_3} = [-3.48, 5]$. A comparison with a Davis-like operator gives:

label	*local consistency*	*Davis*
L_{X_1}	$\{[-5, 5]\}$	$\{[-5, 5]\}$
L_{X_2}	$\{[-4.23, 4.73]\}$	$\{[-5, 5]\}$
L_{X_3}	$\{[-3.48, 5]\}$	$\{[-5, 5]\}$

[2] We assume here that the intervals to be propagated have been splitted in monotonic and continuous parts as discussed in Section 3.1. In this example, $L_{X_i}, i = 1..3$ would have to be splitted into two parts: $[-10, 0)$ and $(0, 10]$ and each combination of monotonic subparts of the labels would be propagated separately.



Type	(x_{1e}, x_{3e})	α
-	$(-, +\infty)$	0
below max	$(-1.54, 4.75)$	1
below max	$(1.54, 4.75)$	2
above min	$(4.17, -2.75)$	1
above min	$(-4.17, -2.75)$	0

Figure 4.5: *The slice containing all four intersections at $X_2 = 1/4$ is shown on the left side. The table on the right side indicates all stationary points in X_3 on the boundary of the restricted region with their index α.*

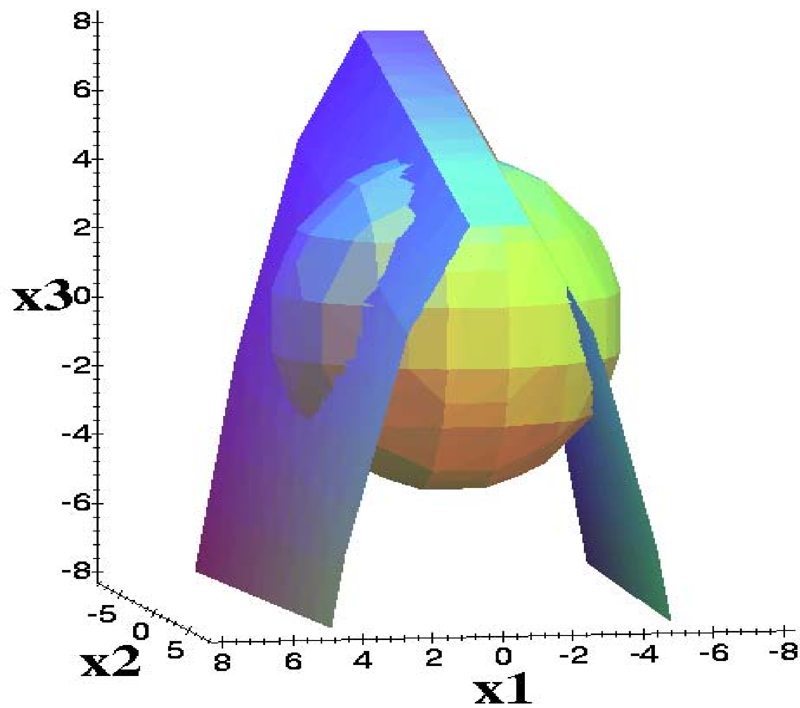
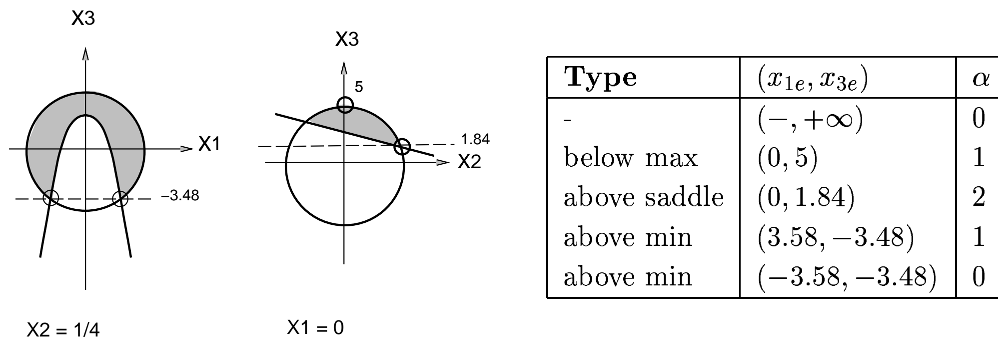


Figure 4.6: A three dimensional plot of the total constraint formed by $X_1^2 + 1/2 * X_2 + 2 * (X_3 - 6) \geq 0$ and $X_1^2 + X_2^2 + X_3^2 - 25 \leq 0$.



Type	(x_{1e}, x_{3e})	α
-	$(-, +\infty)$	0
below max	$(0, 5)$	1
above saddle	$(0, 1.84)$	2
above min	$(3.58, -3.48)$	1
above min	$(-3.58, -3.48)$	0

Figure 4.7: *The slice containing the two intersections at $X_2 = 1/4$ is shown on the left side. The second slice shows the intersection at $X_1 = 0$. The table on the right side indicates all stationary points in X_3 on the boundary of the restricted region with their index α.*

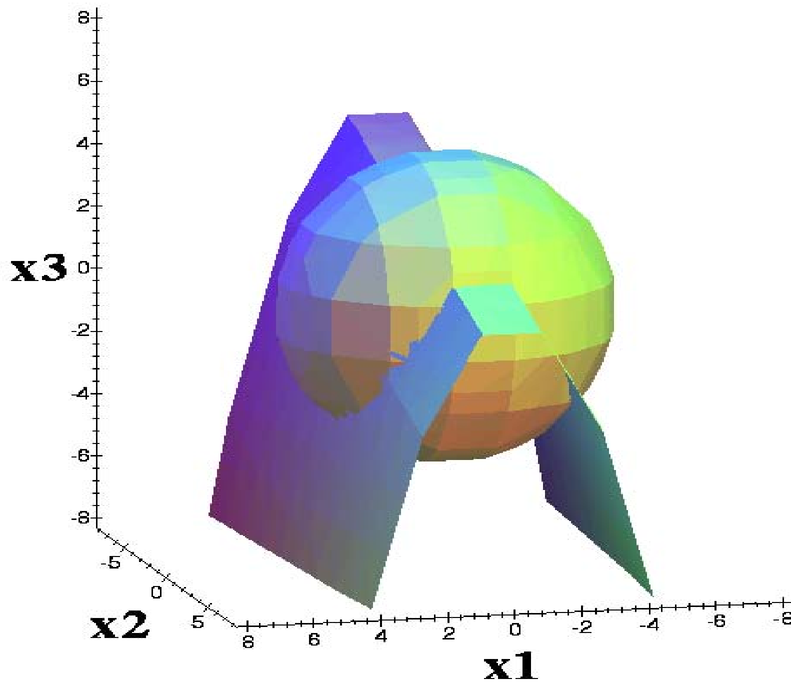


Figure 4.8: *A three dimensional plot of the total constraint formed by $X_1^2 + 1/2 * X_2 + 2 * (X_3 - 3) \geq 0$ and $X_1^2 + X_2^2 + X_3^2 - 25 \leq 0$.*

4.2.3 Correctness of the propagation rule

The purpose of this section is to prove the correctness of the propagation rule we described in Algorithm 4.4 in the preceding section. Given two main categories of stationary points:

$$T1 = \{\text{below max, below min, above saddle}\}$$
$$T2 = \{\text{above max, above min, below saddle}\}$$

we define the index $\alpha_Y(R, y)$ of a region $R \in \mathbb{R}^3$ as follows:

Definition 4.4 *The index of a region $R \in \mathbb{R}^3$ is the difference between the number of stationary points of type $T1$ and the number of stationary points of type $T2$ on its boundary $B(R)$ above $Y = y$:*

$$\alpha_Y(R, y) = |\{T1_Y(B(R), y_0) \mid y_0 \geq y\}| \quad - \\ |\{T2_Y(B(R), y_0) \mid y_0 \geq y\}|$$

with $T1_Y(B(R), y)$ being true if there exists a stationary point in $B(R)$ at coordinate y of type $T1$ and $T2_Y(B(R), y)$ true if there exists a stationary point in $B(R)$ at coordinate y of type $T2$.

First, we prove that the number of stationary points of type $T1$ equals the number of stationary points of type $T2$ for any closed region.

Lemma 4.1 *For a given closed region $R \in \mathbb{R}^3$ and its surface $B(R)$*

$$|\{T1_Y(B(R), y_0)\}| = |\{T2_Y(B(R), y_0)\}|$$

Proof: We prove the lemma by induction. The simplest type of region has the form of a sphere with one maximum in Y and one minimum in Y such that the lemma is verified. We prove now that a topological transformation[3] does not change this relation. Two topological transformations are possible:

1. a new local maximum is added (stationary point of type $T1$): this also implies the creation of a saddle point of type *below saddle* (type $T2$) and vice versa
2. a new local minimum is added (type $T2$): this implies the creation of a saddle of type *above saddle* (type $T1$) and vice versa

In both cases, the relation is verified. Any simple region can so be obtained by a series of topological transformations always keeping the number of stationary points of type $T1$ equal to those of type $T2$.

The same is true for path-connected regions. In fact, holes are defined by the outside of simple regions. This implies that a local maximum of the simple region defining the interior of the hole becomes a point of type *above max*, a minimum a point of type *below min*, a *below saddle* an *above saddle* and vice versa. In other words, the number of points of type $T1$ again equals the number of points of type $T2$ for a hole. Eventually, two possibilities of combining holes and regions exist:

[3] A topological transformation consists of deforming the surface in a smooth manner as if one would treat a piece of clay.

1. the hole is a cavity
2. the hole is a channel that pierces the region and one or both ends

Let S be the outer surface of R and H be the inner surface of the region enclosing the holes. For the first case, we have

$$|\{T1_Y(S, y_0)\}| = |\{T2_Y(S, y_0)\}|$$
$$|\{T1_Y(H, y_0)\}| = |\{T2_Y(H, y_0)\}|$$

and thus the lemma is verified. In the second case, the *above max* and/or *below min* of the hole is replaced by a saddle point of type *below saddle* respectively *above saddle* and the lemma remains true ▲

Prior to explaining how the index α behaves in the presence of stationary points, we need to show that the process of merging or separating regions with a smooth boundary in $\mathbb{R}^2$ never involves more than two regions.

Lemma 4.2 *Let $A_i \subset \mathbb{R}^2$, $i = 1, \ldots, n$ a set of non-empty regions with a smooth boundary. Let P be a point of $\mathbb{R}^2$ such that each A_k touches each other A_j, $k \neq j$ only at point P. Then n, the number of regions, is 2.*

Proof: The condition $\cap_{i=1}^{n} A_i = \{P\}$ implies that the A_i all share the same tangent at P. Assume $n > 2$. Consider a sphere $S_i(C_i, r_i)$, with radius $r_i > 0$ for each A_i such that each sphere is the sphere with the largest possible radius completely within the region of A_i going through P; i.e. the spheres only intersect in P and therefore share the tangent in P. This sphere is unique for each region. The centers C_i must all lie on a line perpendicular to the tangent going through P. It follows that only two spheres can be placed on either side of the tangent with their center on the line without intersecting elsewhere than in P. Thus $n = 2$ ▲

Note that Lemma 4.2 is no longer true if we assume kinks and corners in the boundaries of the regions A_i; e.g. discontinuities at the first derivatives of the function defining their boundary (Figure 4.9). Here, the sphere placed in the interior of a region degenerates into the point P (see Section 4.3.2 for further discussion on degenerate stationary points).

The two main lemmas relate the index $\alpha_Y(R, y_0)$ of a region with the number of connected regions that appear in the intersection between R and $L : Y = y_0$:

Lemma 4.3 *For a given closed region $R \in \mathbb{R}^3$ with a continuous boundary and for any Y-coordinate y_0 for which there is no point $(x, y_0, z) \in R$, $\alpha_Y(R, y_0) = 0$.*

Proof: Given a y_0 with no point (x, y_0, z) in R, R is either completely above $L : Y = y_0$ or completely below L. If R is completely below L, there is no stationary point and $\alpha(R, y_0) = 0$. If R is completely above L, all stationary points of R have been considered and the number of stationary points of type $T1$ equals the number of points of type $T2$ by Lemma 4.1. Again, $\alpha(R, y_0) = 0$ ▲

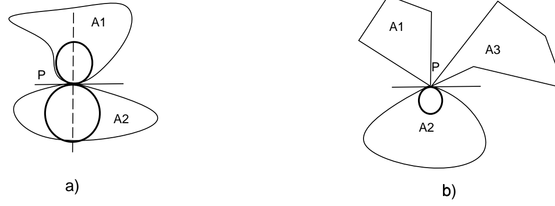


Figure 4.9: *a) Two regions with a smooth boundary touching at point P. b) More than two regions can touch at a single point if they have corners in the boundary.*

Lemma 4.4 *Let R be a single connected region in $\mathbb{R}^3$ with a continuous boundary. Consider the plane $L : Y = y_0$ intersecting R. The following relation holds:*

$$\alpha_Y(R, y_0) = c - h$$

where c is the number of connected regions in $L \cap R$ and h is the number of holes in these regions.

Proof: We present a proof by construction, letting the plane L sweep from $Y = +\infty$ to $Y = -\infty$ updating α each time L touches a stationary point e, i.e. $L : Y = y_e$ with y_e the Y-coordinate of e. Changes in the topology of the intersection between the sweep plane and the region are observed at each of the stationary points. According to its definition, α is incremented for points of type $T1$ and decremented for those of type $T2$:

- If the sweep plane $L : Y = y_e$ touches a stationary point e qualified as *below max*, a new two-dimensional connected region appears in the plane L. At a stationary point that is a *below max*, the index α increases by 1. If L reaches an *above min*, a connected region disappears in plane L and α decreases by 1 (Figure 4.10.1).
- A stationary point of type *below saddle* merges two disjoint regions and α decreases by 1, whereas a point of type *above saddle* separates a region into two disjoint ones and α increases by 1 (Figure 4.10.2).
- A stationary point of type *above max* creates a hole in a former connected region of the plane L; α decreases by 1. This is a special case of two connected regions being merged, as a single connected region is connected to itself. A point of type *below min* makes such a hole disappear, and α increases by 1 (Figure 4.10.3).

According to lemma 4.2, if the three-dimensional region has a smooth boundary there can never be more than two connected regions touching simultaneously at a stationary point. Therefore, α is only decremented by 1 each time two regions merge and incremented by 1 if two regions separate. Finally, each time a new connected region appears α is incremented by 1 and when two connected regions are merged into one α is decremented by 1. This implies that α counts the number of connected regions in L minus the number of holes appearing in these regions ▲

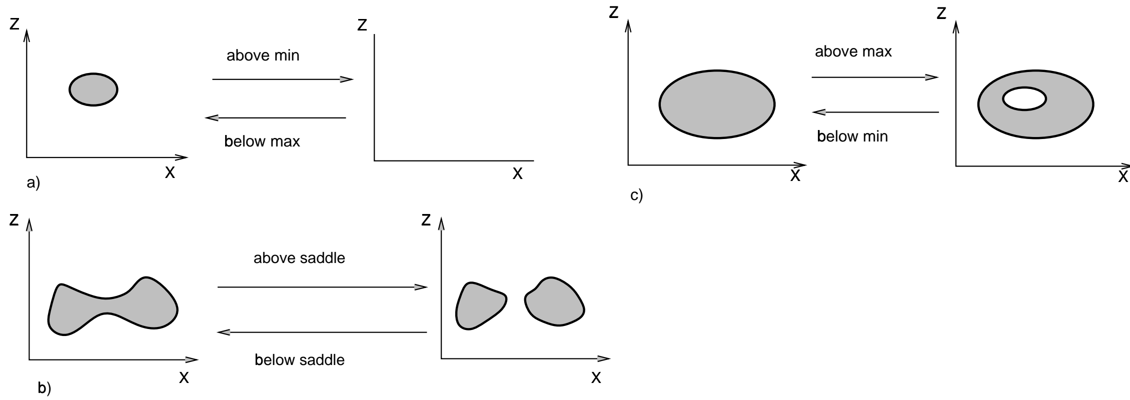


Figure 4.10: *The three figures show how the intersection between the sweep plane L and R and changes when L passes through a stationary point of type $T1$ or type $T2$.*

If the plane $L : Y = y_0$ does not intersect the region R, $\alpha_Y(R, y_0) = 0$. In other words, if L lies completely above or below the region, the number of regions as well as the number of holes in L must be 0. However, contrary to the binary case, $\alpha(R, y_0)$ might also be zero or even negative although L intersects the region R. This phenomenon is due to holes that appear in the plane and are subtracted from the connected regions.

We will first consider the case where $R \in \mathbb{R}^3$ is simple and deduce some useful propositions:

Lemma 4.5 *For any Y-coordinate y_0 for which there is a point $(x, y_0, z) \in R$, $\alpha_Y(R, y_0) > 0$.*

Proof: Consider the plane $L : Y = y_0$ and a simple region R. Since (x, y_0, z) is in R, there must be at least one connected region in $R \cap L$. By lemma 4.4, the number of connected regions in L must be positive; i.e. $c > 0$. Furthermore, R contains no holes, which implies $h = 0$. It follows that $\alpha_Y(R, y_0) > 0$ ▲

Lemma 4.6 *For any simple region R and any value y_0, $\alpha_Y(R, y) \geq 0$.*

Proof: Either there exists a point $(x, y_0, z) \in R$ and $\alpha_Y(R, y_0) > 0$ by lemma 4.5 or there exists no such point and $\alpha_Y(R, y_0) = 0$ by lemma 4.3 ▲

A set of simple regions with a smooth boundary in $\mathbb{R}^3$ can be treated accordingly to the binary case:

Theorem 4.1 *For a Y-coordinate y_0 and a feasible region Q, there exists a region $R \in R(Q, \{I_X, I_Z\})$ containing a point (x_0, y_0, z) with $x_0 \in I_X$ and $z \in I_Z$ if and only if $\sum_{R_i \in R(Q,\{I_X,I_Z\})} \alpha_Y(R_i, y_0) > 0$. Furthermore, there exists a point $(x_0, y_0, z) \in C^t_{XYZ}$ with $x_0 \in I_X$ and $z \in I_Z$ if and only if $\sum_{Q \in C^t_{XYZ}} \sum_{R_i \in R(Q,\{I_X,I_Z\})} \alpha_Y(R_i, y_0) > 0$.*

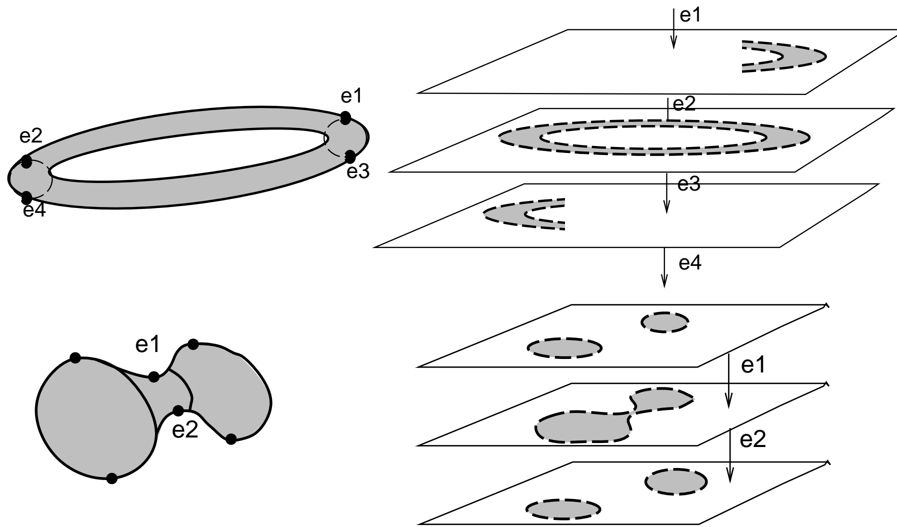


Figure 4.11: *Saddle points e_2 and e_3 on a region with a hole, a torus (upper image) and saddle points e_1, e_2 on a simple region, a cudgel (lower image). The left sides show what happens in the intersection of L with the region.*

Proof: Suppose there was no restricted region R containing a point (x_0, y_0, z). This implies that for all regions R_i, by lemma 4.3 $\alpha_Y(R_i, y_0) = 0$ and the sum of all indices is equally zero. Conversely, if there is a region R containing the point (x_0, y_0, z), by lemma 4.5 $\alpha_Y(R, y_0) > 0$ for at least one region R. Thus, the sum of all indices must be positive by lemma 4.6. The same argument extends to all restricted regions Q for a total constraint ▲

4.2.4 Propagating a single interval through regions containing holes

In this section, we discuss the extension of the propagation rule 4.4 to regions containing holes.

The only types of stationary points at which a hole starts to appear in a region are *below saddle* and *above max*. The reader can verify this easily by looking at Figure 4.10. The difference is that a point of type *above max* always introduces a hole whereas a point of type *below saddle* connects two different parts of the same three-dimensional region. If these parts are already connected, the stationary point *below saddle* adds a second connection such that the region forms a hole. The ambiguity that lies in the interpretation of a stationary point of type *above saddle* is shown in Figure 4.11.

Remember that α is decreased at the start of a hole due to a stationary point either either of type *above max* or *below saddle*. When a region contains one or several holes, the index α may drop below 1 without the region being completed because the existence of a hole implies the existence of a region around the hole. In Figure 4.12, the index α is shown for a feasible region formed by a torus. Although α falls below 1 at the saddle point e_2, we know that the region as not yet been completely considered because a new region can only start at a stationary point of type *below max* and end at one of type *above min*

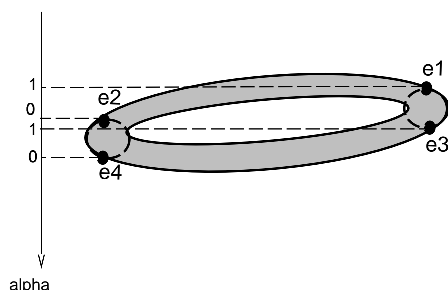


Figure 4.12: *A torus is treated correctly by the propagation rule because no new region can start or disappear at a saddle point.*

and this is not the case for the torus. Our rule can treat a torus correctly. Let us try to imagine what happens if additional local minima and maxima are added to a region hiding the hole. Imagine a mushroom with a wormhole like the one shown in Figure 4.13. The appearance of a hole in e_3 is masked by a branch of the region appearing to the right of the stem, which ends in the local minimum e_4. Additionally the vanishing of the hole in e_6 is hidden by the appearance of yet another branch at the left of the stem starting at e_5. It thus seems natural that a new region starts at e_5, which seems to be without connection to the region ending at e_4. In other words, if the operator does not know about the hole that has been created in e_3, the counting of stationary points leads to a false gap. Instead of the interval from e_8 to e_1 the result would be $e_8..e_5$ and $e_4..e_1$.

The existence of such figures like the mushroom with a wormhole may be hypothetical. In most cases, the additional rule that if α drops to 0 at a stationary point different from *above min* or if α goes to 1 at a point different from *below max*, no gap is created, is sufficient to consider single regions with holes correctly. For the general case however, a more profound analysis of the saddle points *below saddle* and *above saddle* is necessary in order to determine if such points are situated at holes. Those two types of stationary points would have an additional label indicating the existence of a hole. With this additional information, the refine operator can correctly treat any set of regions containing holes.

4.2.5 Extension of the refine operator

The original operator 4.4 for simple regions has to be modified in order to take into account holes in regions. We assume here that additional information is available at saddle points indicating if they are situated at the boundary of a hole. The refine operator treating simple regions is extended by an additional counter H counting the number of holes at each stationary point. According to lemma 4.4, it is clear that a region has been completely considered if and only if H and α are zero (Algorithm 4.14).

4.2.6 Ternary constraints with different third variables

Until now, the propagation rule computes locally consistent labellings for a restricted class of regions represented by constraints defined in the same ternary space. Consider the

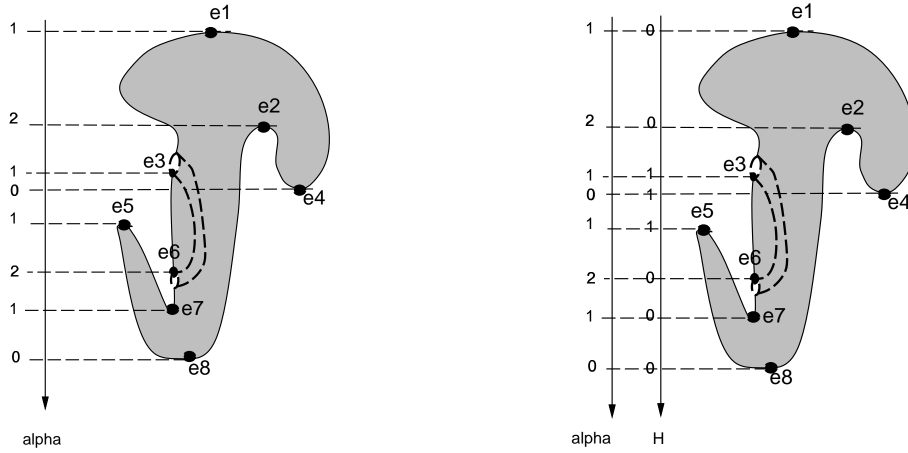


Figure 4.13: *False gaps may appear if the existence of a hole at saddle points is not detected. The refine operator based solely on the identification of stationary points fails if the existence of a hole at a saddle point is hidden by local maxima and minima (mushroom with wormhole). An extended refine operator with the additional information about holes can of course treat the region correctly.*

example of the parabolic surface and the sphere modified such that the sphere is now defined in space X_1, X_3, X_4:

$$\begin{aligned} C_1 &:= X_1^2 + 1/2 * X_2 + 2 * (X_3 - 6) \geq 0 \\ C_2 &:= X_1^2 + X_3^2 + X_4^2 - 25 \leq 0 \end{aligned}$$

If each constraint is treated individually by the propagation rule, intersections that are stationary points in X_3 get lost just as if in the binary case intersections between binary constraints would not be considered. It follows that when there are several third variables $Z_1, Z_2, ..., Z_k$ over which the constraints C_{XYZ_i} are defined, the total constraint is $C^t_{XY} = \{C_{XYZ_i} \mid i = 1 \dots k\}$. Two questions arise: first, which of the stationary points of $\{C_{XYZ_i}\}$ are part of the regions formed by the total constraint (filtering) and second, how are the stationary points of the total constraint classified. In order to understand how a total constraint over several third variables should be treated, let us first consider constraints defined on the same third variable Z, $C^t_{XY} = \{C^1_{XYZ}, \dots C^m_{XYZ}\}$. The total constraint defines a conjunction of constraints; i.e. intervals must be propagated through their intersection. The intersection is computed implicitly by pruning stationary points that do not satisfy all constraints of the total constraint. The remaining stationary points are part of the boundary of the intersection and the new labels are computed based on them. When the third variables are different, an interval must also be propagated through their *intersection*. Each of the constraint sets $\{C_{XYZ_i}\}$ defines a set of feasible regions $R^i = \{Q^i_1, \dots, Q^i_{m_i}\}$ in the space XYZ_i. The aim is to compute the index of the intersection of these regions, denoted $R^\cap = \cap_{i=1}^k R^i$, without explicitly constructing this intersection. From the preceding sections we know how to compute and filter the stationary points for each R^i.

function **compute-intervals**(E)
begin
1 $I_Y \leftarrow \{\}$
2 $\alpha \leftarrow 0$
3 $H \leftarrow 0$
4 **for** each $e \in E$ **do**
5 **if** e is of type below max, below min, above saddle **then**
6 $\alpha \leftarrow \alpha + 1$
7 **elif** e is of type above max above min, below saddle **then**
8 $\alpha \leftarrow \alpha - 1$
9 **fi**
10 **if** e is of type above max or below saddle at a hole **then**
11 $H \leftarrow H + 1$
12 **elif** e is of type below min or above saddle at a hole **then**
13 $H \leftarrow H - 1$
14 **fi**
15 **if** $\alpha + H$ has changed from 0 to 1 **then**
16 $y_{max} = Y$-coordinate(e)
17 **elif** $\alpha + H$ has changed from 1 to 0 **then**
18 $y_{min} = Y$-coordinate(e)
19 $I_Y \leftarrow I_Y \cup \{[y_{min}, y_{max}]\}$
20 **fi**
21 **od**
22 **return** I_Y
end

Figure 4.14: *Revised version of* **compute-intervals** *computing a locally consistent label for Y given the stationary points E of which all saddle points are labeled with additional information about the existence of holes.*

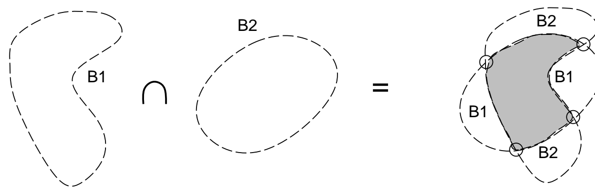


Figure 4.15: *The intersection of the projections of two regions onto their common axes results in a region sharing the boundaries of B_1 and B_2 as well as intersections between the boundaries.*

Since the region $R^{\cap}$ represents a n-ary region with $n \geq 3$ and operations in this space are difficult to visualize, our original idea was to work on the projection of the ternary constraints onto the common variables and to derive a locally consistent labelling from this projection. Two difficulties arise:

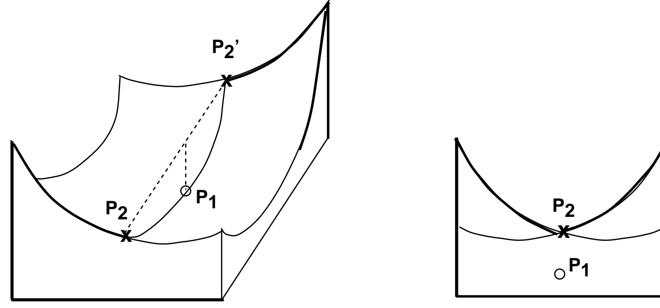


Figure 4.16: *Concave regions produce stationary points that can be hidden in the projection of the region. P_1, a* below min *of the ternary region, is hidden in the projection of the figure. In addition a new local extremum P_2 appears in the projection, which does not even exist as single point in the ternary region.*

1. The filtering step in order to compute the intersections between different projections is likely to become inefficient because a stationary point $e = (x_e, y_e, z_e)$ of constraint $C^1_{XYZ_i} : E^1(X, Y, Z_i) \geq 0$ must satisfy any other constraint $C^2_{XYZ_j} : E^2(X, Y, Z_i) \geq 0$ with $j \neq i$. In other words, there must exist a value z for Z_j such that $E^2(x_e, y_e, z) \geq 0$.
2. There is not always a one-to-one relationship between the stationary points of a ternary region and the local extrema of its projection (Figure 4.16).

We can approximate the projection of ternary regions onto a pair of variables by taking all slices at the third coordinates of stationary points. This construction guarantees that all stationary points of the ternary region are *within* the projection. However, not all of them are on the boundary of the projection as shown in Figure 4.17.

In the light of these difficulties, we decided to restrict the refine operator to constraints that are defined in the same ternary space as presented in Figures 4.4 and Figure 4.14.

4.3 Implementation

As for the binary refine operator, we will show how stationary points of ternary constraints can be identified. As already elaborated in the case of binary constraints, we assume that *no discontinuities and no singularities appear on the boundary of the restricted regions.*

4.3.1 Identifying stationary points

We will first show how to identify stationary points of total constraints with a single third variable. Consider a propagation step from X to Y, propagating the interval I_X through the total constraint $\{C^1_{XYZ}, C^m_{XYZ}\}$. Under the assumption that the boundary of the restricted regions contains no discontinuities nor singularities, stationary points are found among the following sets:

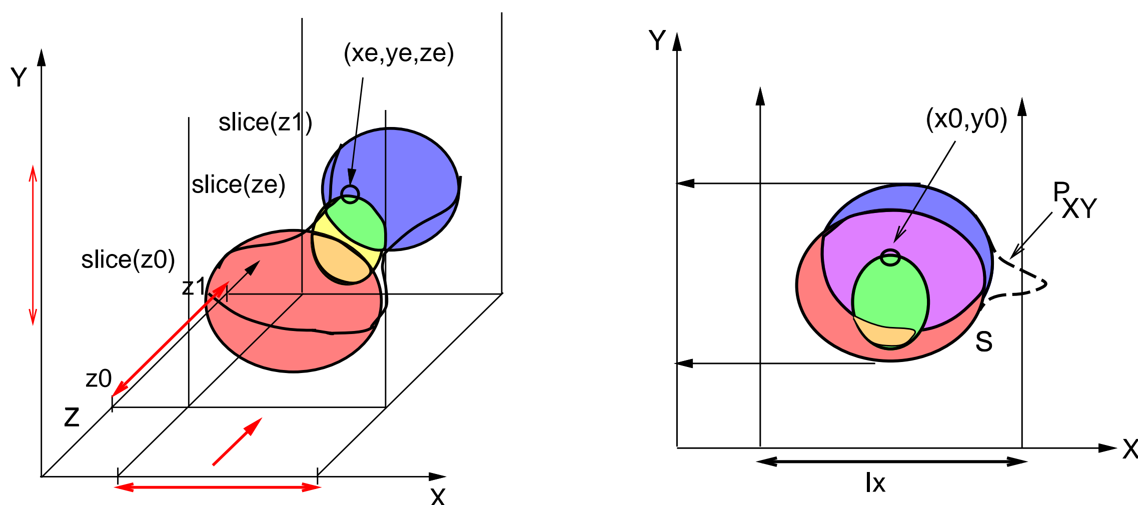


Figure 4.17: *Propagating intervals through a set of ternary constraints with the same third variable representing a hyperboloid. The point (x_e, y_e, z_e) is a saddle point in Y of the ternary constraint. On the right side, the volume has been projected onto X, Y. The three slices at z_0, z_1, z_e generate the set S that approximates P_{XY}. In this case S covers P_{XY} exactly. If we added a kind of "beak" that possesses no stationary point in Y somewhere inbetween the slides, S would not cover P_{XY} shown by dashed lines exactly. The corresponding volume would resemble a crouched chick.*

1. Local extrema or saddle points of an individual constraint surface lying within I_X and I_Z.
2. Intersections between two or three constraint surfaces lying within I_X and I_Z.
3. Local extrema and intersections at the interval bounds of I_X and I_Z.

Again, when the surface of an individual constraint $C_{XYZ_i} : E(X,Y,Z_i) \geq 0$ is regular at a point, the function of the surface $E(X,Y,Z_i) = 0$ can be expressed as $Y = F(X,Z_i)$ such that $E(X,F(X,Z_i),Z_i) = 0$ (Implicit Function Theorem, Appendix C). The partial derivatives of the function F are defined by $\frac{\partial F}{\partial X} = -\frac{\partial E}{\partial X} / \frac{\partial E}{\partial Y}$ and $\frac{\partial F}{\partial Z} = -\frac{\partial E}{\partial Z} / \frac{\partial E}{\partial Y}$.

4.3.1.1 Local extrema and saddle points of an individual constraint surface

The first set of stationary points occuring on individual constraint surfaces are *local extrema* and *saddle points*. Consider a single ternary constraint $C_{XYZ} : E(X,Y,Z) \geq 0$ and the function defining the surface $Y = F(X,Z)$. The potential candidates for local extrema and saddle points are determined by the necessary condition:

Condition 4.1 *If $e = (x_e, y_e, z_e)$ is a local extremum or a saddle point in Y of $Y = F(X,Z)$ and its partial derivatives exist at that point, e is a stationary point if the following conditions are verified:* $\frac{\partial F(x_e,z_e)}{\partial X} = 0$ *and* $\frac{\partial F(x_e,z_e)}{\partial Z} = 0$

This necessary condition allows us to derive candidates, but, not all of the points satisfying the condition are local extrema or saddle points. F has a local extremum or a saddle point

if the following sufficient condition is verified:

Condition 4.2 *Assume that $F : I \to \mathbb{R}$, $I \subseteq \mathbb{R}^2$ is two times continuously differentiable in $e = (x_e, y_e, z_e)$, e is a stationary point and that*

$$r = \frac{\partial^2 F(x_e, z_e)}{\partial X^2}, t = \frac{\partial^2 F(x_e, z_e)}{\partial Z^2}, s = \frac{\partial^2 F(x_e, z_e)}{\partial X \partial Z}.$$

- *If $s^2 - rt < 0$ and $r > 0$, F has a local minimum in e.*
- *If $s^2 - rt < 0$ and $r < 0$, F has a local maximum in e.*
- *f $s^2 - rt > 0$ and $r * t < 0$, F has a saddle point in e.*

No direct conclusion whether F has a local extremum in e can be drawn if $s^2 - rt = 0$ [Douchet and Zwahlen, 1986]. In this case, the slices have to be analyzed in detail in order to find if they both possess a local extremum. According to the gradient direction in e, e is then classified either as a below or an above minimum, maximum or saddle exactly in the same manner as described for constraints in $\mathbb{R}^2$ (Section 3.2.4.3). It is sufficient to determine the type of local extremum in one of the slices (main or secondary) in order to classify e. Finally, only those extrema and saddle points are considered that lie within the intervals of I_X and I_Z.

4.3.1.2 Intersections between constraint surfaces

Intersections that are extrema in Y appear between pairs or triplets of constraints defined in the same space. We need to find points where the intersection curve between constraints reaches an extremum. Remember the notation $\nabla\ E(x_e, y_e)$ as gradient of the constraint E. N_Y is the Y-component of a vector $\vec{N}$. The condition similar to the one for binary constraints is:

Condition 4.3 *Let C^1_{XYZ}, C^2_{XYZ} and C^3_{XYZ} be ternary constraints. $e = (x_e, y_e, z_e)$ is an extremum in axis Y in the intersection of the three constraints if there exists a vector $\vec{N}$ parallel to axis Y and $\beta_1, \beta_2 \geq 0$ such that the following condition is satisfied:*

$$\left\{ \begin{array}{rcl} \nabla\ E^1(x_e, y_e, z_e) + \beta_1 * \nabla\ E^2(x_e, y_e, z_e) + \beta_2 * \nabla\ E^3(x_e, y_e, z_e) & = & \vec{N} \\ E^1(x_e, y_e, z_e) & = & 0 \\ E^2(x_e, y_e, z_e) & = & 0 \\ E^3(x_e, y_e, z_e) & = & 0 \end{array} \right\}$$

The stationary point e is of type below max *if $N_Y < 0$ and* above min *if $N_Y > 0$. A special case of this condition is: If either $\beta_1 = 0$ or $\beta_2 = 0$, the gradient vectors on two constraints lie in a plane parallel to the Y-axis. In this case the intersection forms a curve (and not only a point) and e can be of types T_1 or T_2 determined by the two slices.*

Proof: The projection of pairs of gradients onto X, Y or onto Z, Y satisfies Condition 3.3 because e is an intersection and remains an intersection in the projection. The combination of the projected gradients from both slides results in the gradient equation of the condition ▲

Only intersections that lie within the intervals of I_X and I_Z are considered.

4.3.1.3 Local extrema at interval bounds

Additional candidates can also appear at the intersection between interval bounds of I_X and/or I_{Z_i} and ternary constraints in the form of individual extrema on constraint curves and intersections between binary constraints. Finding these candidates involves two steps:

1. Replace one interval bound of I_X or I_{Z_i} with the ternary constraint. An example is the substitution of $X = max(I_X)$ by C^t_{XY}.
2. compute the local extrema of the resulting binary constraints according to section 3.2.4 and classify the resulting local extrema as points in $\mathbb{R}^3$ by looking at the secondary slice.

4.3.1.4 Algorithms for filtering and identifying candidates

The algorithms for identifying and filtering candidates are shown in Figure 4.18. In **identify-candidates**, those stationary points are identified that occur at intersections of constraints or at individual constraint surfaces. They are exactly those stationary points that can be precomputed. Furthermore, local extrema at interval bounds are identified. The algorithm **filter-candidates** tests for each stationary point obtained if it satisfies all other constraints of the total constraint and if it lies within the intervals I_X and I_Z.

4.3.2 Special cases of stationary points

In some cases, stationary points consist of point sets; they may describe curves or even surfaces. The tori in Figure 4.19, illustrate the creation of such curves. A torus lying diagonally in space is tilted until it is in a horizontal position. The horizontal planes to the right of the tori show the intersection of a plane L with the tori at a given point on the vertical axis. In the first and second picture the stationary points are single. For the second torus a hole is created in L due to the saddle point e_2. The third torus is marked by two stationary points that are ellipses: e_1 and e_2. Both have been created because two stationary points are collapsed into one: e_1 is the sum of the local maximum and the first saddle point and e_2 contains the second saddle point and the local minimum of the second torus. A stationary point that is a closed curve thus either symbolizes the creation of a region and a hole at the same time or the vanishing of a region and a hole and the index α stays 0. We call these stationary points *degenerate*. In Figure 4.20, the region presented has the form of a Mexican hat. At the first stationary point e_1 a region appears ($\alpha = 1$), at the second, e_2 this region is surrounded by a circle ($\alpha = 1+0$), and eventually the circle and the region collapse into a second filled circle ($\alpha = 1 + 1 - 1$) vanishing in e_4. These examples show that stationary points that are curves are no longer local since they may integrate more than one type of stationary point. As soon as the symmetry is destroyed, for example by a rotation around an axis not parallel to the axes of the coordinate system, all stationary points are again computable in isolation.

function identify-candidates(I_X,C^t_{XY}, $\{I_Z\}$)
begin
1 $E \leftarrow$ stationary points in Y belonging to the following:
local extrema, saddle points (Condition 4.2) and intersections (Condition 4.3)
between pairs and triplets of constraints
2 $\mathcal{L} \leftarrow$ coordinates z_e of bounds of I_Z and x_e of bounds of I_X
3 $S \leftarrow \bigcup_{i=1}^{|\mathcal{L}|} S^i_{XY}(C^t_{XY})$
4 **for** each $s \in S$ **do**
5 $i \leftarrow X$ or Z depending on s
6 $E' \leftarrow$ **identify-candidates**(I_i, s)
7 $E' \leftarrow$ only keep those candidates of E' that are also local extrema in
the secondary slice and label them either $T1$ or $T2$
8 $E \leftarrow E \cup E'$
9 **od**
10 **return** E
end

function filter-candidates(E,I_X,C^t_{XY}, $\{I_Z\}$)
begin
11 $E \leftarrow$ those points in E that satisfy all other constraints defined
in the SAME space X, Y, Z and that also lie within I_X and I_Z
12 **return** E
end

Figure 4.18: *Detailed algorithms of how to find candidates and of the filtering step for a total constraint.*

Degenerate stationary points also appear at local extrema or saddle points of a single constraint surface. Such a degenerate point has a singular Hessian form (see [Milnor, 1963]) at that point. In such a case, it is sufficient to use a rotation in order to identify the point correctly.

Regions with degenerate stationary points are treated correctly by our propagation rule as long as the the stationary points can be identified. With a discretized representation of the feasible space (Section 3.2.4.1), it might be easier to classify degenerate stationary points than by analytical methods.

The same is true for a single constraint with ends meeting at a stationary point (Figure 4.21). In this case, the boundary of at least one of the branches must end in a corner at the stationary point according to Lemma 4.2. Recognizing that several branches part from a stationary point and thus several regions are created or disappear in the intersection with L simply requires more powerful methods for identification. Discontinuities in the partial first-order derivatives are responsible for this type of kinks and corners. The local analysis of the stationary point we propose is in this case not sufficient and a deeper analysis of the branches is necessary.

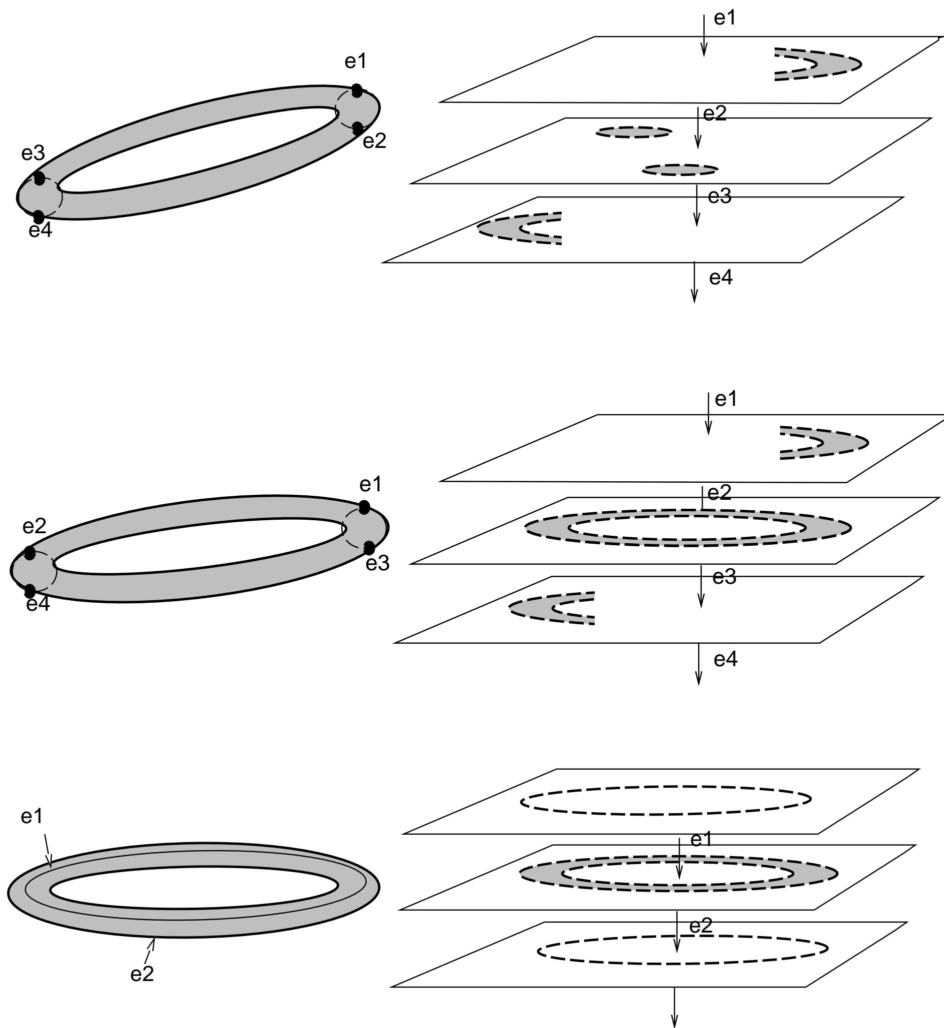


Figure 4.19: *Tori in different positions. The planes taken at points between two stationary points show the change in the intersection of L with the torus.*

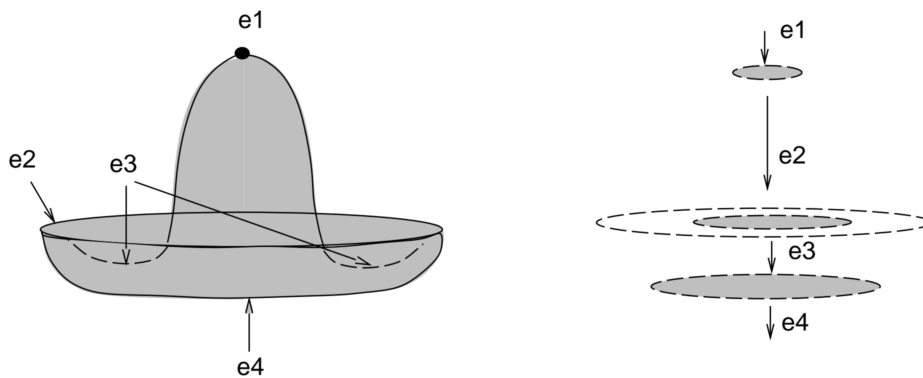


Figure 4.20: *The region defined by the inner part of a "mexican hat". Since this figure can be created by rotation, it is symmetric.*

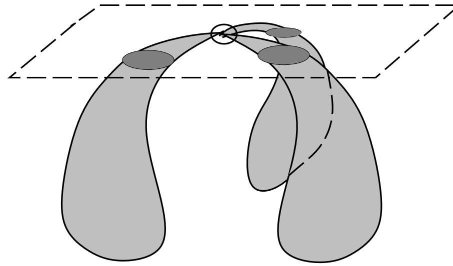


Figure 4.21: *Three branches meeting at a single stationary point. At least one of them must have a corner at the stationary point.*

4.4 Summary

In practice, binary constraint systems are very rare. And, in contrast to discrete CSPs, there exists no generic method that transforms k-ary numeric CSPs into binary CSPs. This implies that a refine operator rule achieving local consistency for numeric general CSPs has to work directly on k-ary constraints or on their projection onto pairs of variables. We show that any CSP containing k-ary continuous constraints with $k \geq 3$ can be transformed into a set of ternary constraints. One of the contributions of this thesis is the generalization of the binary refine operator described in [Faltings, 1994] to ternary constraints over continuous variables. The generalization of the refine operator to ternary constraints again makes use of the property that only the complete set of local extrema has to be determined for the feasible regions without knowing to which region the extrema belong to. Also, the extrema can be determined locally by inspecting maximally triplets of constraints. Intersections between ternary constraints are considered only for constraints defined in the same space, i.e. which have the same third variable. An extension to constraints defined over the same pair of variables but differing in their third variables would have required to consider k-ary feasible regions ($k \geq 3$), in which the determination of local extrema becomes extremely difficult.

A fundamental limitation of the ternary refine operator is discovered that is due to the increasingly complex regions that ternary constraints can form. In the case of feasible regions containing holes, our refine operator requires further information in order to achieve local consistency. In this case, those local extrema lying on the boundary of a hole have to be identified additionally. In practical examples, this information might be given explicitly because the constraint region is known. The examples in the chapter on results suggest that the majority of the constraints define simple volumes without holes, which our refine operator can treat without problem.

With respect to other algorithms achieving some form of local consistency, our propagation rule also takes into account intersections between constraints defined in the same three-dimensional space over the same pair of variables and thus results into tighter labellings. However, our operator is sensible to holes within three-dimensional regions, which is not the case of operators computing an approximation of the feasible region.

The restriction of our refine operator to ternary constraints defined in the same three-

dimensional spaces as well as the difficulty of detecting holes indicate a certain increase in topological complexity with respect to the binary refine operator. Nevertheless, the fact that we do not restrict ourselves to specific types of constraints, like for example polynomials in Numerica ([Van Hentenryck et al., 1995]), the topological concepts on which the correctness proof is based allows us to discuss local consistency for continuous constraints in more generality than has been done in literature until now.

Chapter 5

Systematic generation of problem spaces

In science and engineering the study of "systems" is an increasingly popular activity. Its popularity is more a response to a pressing need for synthesizing and analyzing complexity than it is to any large development of a body of knowledge and technique for dealing with complexity.

[Simon, 1981]

5.1 Introduction

As discussed in the introduction, the inputs of a design task are: a model that describes the components that can be included in the design and a set of constraints that define how components can be combined to form a working product, and customer requirements that specify properties of an individual product. The output is a description of a product to be manufactured, which comprises not only the set of components but also their arrangement in the product structure, i.e. a model of how components relate to each other. The resulting design has to satisfy all the constraints given in the model and the customer requirements. Fundamental forms of knowledge used to describe a design task include that a *choice* must be made, *optional components* may be added to the design, some components are *incompatible* and that some component may *require* another component to function correctly (functional dependency) [Soininen and Niemelä, 1998]. In principle, the CSP paradigm offers an adequate framework for such tasks because it provides a simple, declarative model for representing a design task and also powerful methods for solving it (see introductory sections on search in CSPs 2.6, 2.7). A conventional (or static) CSP encodes choices between different values for a variable and also incompatibilities as those tuples that are not allowed in a constraint. However, when using the static CSP model to solve design tasks, some important shortcomings can be identified:

- It requires all variables and constraints to be known explicitly before problem resolution. This implies that it cannot handle an unknown number of components.

- It does not capture the internal structure of components; i.e. a given hierarchical product structure is flattened into a set of variables of equal importance.
- Functional dependencies between components and thus optional components are difficult to encode in a static CSP. The addition of NULL values to constraints in order to indicate in which context components do not appear in a solution becomes especially difficult in the presence of continuous constraints.
- According to [Mittal and Falkenhainer, 1990], efficiency problems arise whenever the existence of variables depends on specific value assignments. In that case the algorithm has to switch frequently between variable creation and value assignment procedures.

[Mittal and Falkenhainer, 1990] argue that, in synthesis tasks like configuration or design, the set of variables that must be assigned a value may change dynamically in response to choices made in the course of problem solving. The solutions to such a task then differ in the number of variables assigned. When configuring a mixer, for example, a condenser is only necessary if the vessel volume is large and chemical reactions might occur during the mixing process. In other words, the existence of the condenser depends on the type of product to be mixed. Once the product type is decided, the condenser variable is either added to the solution or not. The condenser is a typical example of an optional component, which does not have to be present in every solution. Even more important, new constraints defined on such an optional part might become relevant to the problem. This means that the CSP is no longer static, but dynamically evolving during problem solving, characterized by a changing set of variables and constraints. Each change in a constraint network influences the solution sets. With respect to the modus of interaction between the problem solver and its environment, two main lines can be distinguished:

1. The problem is not completely specified at the time of problem definition. Users or more generally the environment are allowed to add additional variables or constraints during problem resolution.

2. Variables and constraints are known intensionally at the time of problem definition, i.e. the types of variables and constraints corresponding to the problem are given, however not their specific instances. In a bridge design, for example, a configuration may have between two to fifty piers. It is the task of the search algorithm to find the variable instances and the constraints defined on them for each solution.

An example of the first approach is non-monotonic logical reasoning. In general, only a partial description of the world is given so that it is important to maintain the consistency of our world when new facts become known. Another example are graphical user interfaces like ThingLab [Freeman-Benson et al., 1990] where constraints are imposed by the users on graphical objects and the system has to adapt its layout in real time.

In this thesis, we concentrate on the second approach assuming that an input definition for a design task is given intensionally. This approach fits well in the overall design process consisting of refining a given goal first into a a set of functions and subsequently into a more structural representation (Section 1.1).

5.2 Background

In order to overcome the limitations of static CSPs, one can model a varying number of variable instances in a synthesis problem like design or configuration by defining a status :IN or :OUT for each variable. Problem solving then includes reasoning on the status of a variable. In other words, the CSP is embedded into a larger task-specific problem solving architecture, in which separate mechanisms are used for creating variables and processing the constraints. This becomes cumbersome and inefficient when the existence of variables depends on specific value assignments and the constraint processing interacts closely with variable creation, because program control oscillates between both search and variable creation [Frayman and Mittal, 1987]. A further integration into the CSP framework has been achieved by adding a NULL value to the domain of optional variables and by reformulating the constraints on these variables to include the NULL value. A variable is assigned a NULL value in the solution if it is not part of the solution. The disadvantage is that in a large constraint problem, all variables and all constraints are taken into account simultaneously even if some are not relevant to the problem at hand. Handling sets of identical elements of varying size like piers in a bridge design would require constraints combining all possibilities of variable existence or non-existence hard-coded in the constraint. Furthermore, we will show later in this chapter that adding constraints in form of dependencies between variables to a given problem cannot always be reformulated locally in the static CSP model.

Other dynamic models exist like incremental constraint satisfaction that can handle a changing set of constraints and variables. Incremental constraint satisfaction is integrated into most constraint logic languages in order to solve constraints in an incremental way transparent to the user. However, no explicit syntax is provided to the user to reason about the existence of a variable in a solution. In ThingLab [Freeman-Benson et al., 1990] constraints are introduced by user requests and the constraint solver tries to resatisfy the changed constraint set incrementally by applying a perturbation method (Section 2.7.4). In a similar way, Constraint Logic Programming (CLP) satisfies constraints incrementally in order to guarantee efficiency. Incremental satisfaction in CLP proceeds by modifying an existing solution in order to account for new constraints resulting from the top-down, left-to-right parsing. Most CLP algorithms transform the constraints into a solved form, a format in which the satisfiability of the constraints is evident or at least easier to infer [Jaffar and Maher, 1994]. Although there are some applications using CLP for synthesis tasks, the majority of work in the CLP field has been devoted to solving combinatorial search problems or analyzing artifacts like circuits or truss structures. The applications written in CLP for designing an artifact first generate a specific structure and then analyze it [Heintze et al., 1987], [Lakmazaheri and Rasdorf, 1989]. The reason for this is that CLP is based on Prolog, a logic programming language restricted to definite clauses. In this formalism queries of the form "Is variable X_1 or X_2 or ... X_n active ?" cannot be handled directly. All combinations of possibly active components first have to be generated and then tested against the constraints.

The model of *dynamic constraint satisfaction* (DCSP) [Mittal and Falkenhainer, 1990]

has been introduced to adapt the CSP paradigm to a changing environment based on the experience with the Cossack expert system [Frayman and Mittal, 1987]. The standard CSP model is extended to include activity constraints. These constraints state conditions on the existence of variables in a solution. The advantage of the DCSP model is that:

1. A mechanism is introduced that allows for explicit reasoning on the existence of variables.
2. This mechanism is part of the CSP model, which thus defines a general framework for constraint processing and variable creation.

The resolution algorithm for DCSPs, however, is based on value enumeration and thus restricted to discrete DCSPs.

[Sabin and Freuder, 1996] introduce the model of composite CSP, which can be seen as the result of three concepts: dynamic constraint satisfaction, hierarchical domain constraint satisfaction [Mackworth et al., 1985] and meta problems in constraint satisfaction. The difference to a standard CSP consists in the fact that values of a variable are no longer restricted to atomic values. A value can be composite in the sense that it defines an entire subproblem. The advantages of composite CSPs is that consistency and search algorithms can easily be adapted. Decisions on a single value for a variable dynamically introduce new variables and constraints. Such decision making can be formalized by activity constraints.

[Bowen and Bahler, 1991] 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.

5.3 Dynamic constraint satisfaction

The dynamic constraint satisfaction problem (DCSP) defined by [Mittal and Falkenhainer, 1990] relaxes the hypothesis of a standard CSP that a variable defined by the problem has to be part of all solutions. New variables are introduced only if they are relevant to the given search space. A variable is *active* whenever it has to be part of a solution and inactive otherwise. If $\mathcal{V}$ is the set of all potential variables of the problem, the existence of a variable in a solution is defined by the predicate *active*.

Definition 5.1 (Active variable; [Mittal and Falkenhainer, 1990]) *For any* $X_i \in \mathcal{V} : active(X_i) \leftrightarrow X_i = x$ *and* $x \in D_i$.

Typical constraints restricting value combinations of variables are called *compatibility constraints* in this model. Such a constraint formulated as set of allowed tuples or as a logical predicate just as in the standard CSP. In contrast to a standard CSP however, a compatibility constraint in a DCSP is only *relevant* to a problem if all the variables of this constraint are active. Let $\mathcal{C}^C$ be the set of compatibility constraints of a DCSP.

Definition 5.2 (Relevant constraint) *For any* $C_{X_1,\ldots,X_j} \in \mathcal{C}^C : active(X_1) \wedge \ldots \wedge active(X_j) \leftrightarrow relevant(C_{X_1,\ldots,X_j})$

It follows that a compatibility constraint is either trivially satisfied if at least one of its variables is not active or if the constraint is satisfied by an assignment according to definition 2.2. This can be made explicit in the algorithm by considering a compatibility constraint only in those problem spaces in which all variables of the constraint are active. We extend the original definition of DCSP by allowing continuous and mixed compatibility constraints as well.

Additionally, a DCSP can post constraints on a variable's activity in a given context of value assignments. Such a constraint is called *activity constraint.* Let $\mathcal{C}^A$ be the set of all activity constraints of a DCSP.

Definition 5.3 (Activity constraint; [Mittal and Falkenhainer, 1990]) *An activity constraint* $C_{X_1,\ldots,X_j} \stackrel{ACT}{\rightarrow} X_k \in \mathcal{C}^A$, $k \neq 1,\ldots,j$ *is defined by the logical implication*

$$C_{X_1,\ldots,X_j} \rightarrow active(X_k)$$

where $C_{X_1,\ldots,X_j}$ *is a single constraint, which expresses an* activation condition *under which the variable* X_k *becomes active.*

In its original form the activation condition was defined to be a set of value assignments. Again, we extend this definition to a more general form of activation condition. In our model, an activation condition can be either a single discrete constraint or a continuous constraint stated as inequality.

Definition 5.4 *An activity constraint* $C_{X_1,\ldots,X_j} \stackrel{ACT}{\rightarrow} X_k \in C^A$ *with* $k \neq 1,\ldots,j$ *is* satisfied *by a set of values* $\{X_1 = x_1,\ldots,X_j = x_j\}$ *with* $x_i \in D_i$ *for* $i = 1,\ldots,j$ *if* $\neg C_{X_1,\ldots,X_j}(x_1,\ldots,x_j) \vee active(X_k)$ *is true.*

The activity constraint is trivially satisfied, if one of the variables in the condition is not active. Otherwise, if all of its variables are active, i.e. *relevant*(C) is true, either the condition must not be satisfied by the given assignment or X_k is to be active. If the activation condition is satisfied by the assignment, we add the variable to the search space, otherwise, we cannot make any assumption on the activity of the new variable.

Thus, a DCSP incrementally defines problem spaces in which different variables are active and it only assigns values to the active variables. An initial nonempty set of variables V_I specifies the variables that are always active, i.e. those which are part of every solution. The formal definition of a DCSP is:

Definition 5.5 (DCSP; [Mittal and Falkenhainer, 1990]) *A* dynamic constraint satisfaction problem $\mathcal{P} = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$ *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 non-empty 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.*

Find all solutions, where a solution is

1. *an assignment* A *of values to a set of variables such that* A *satisfies all constraints in* $C^C \cup C^A$.
2. *minimal; i.e. there is no solution* A' *satisfying all constraints such that* $A' \subset A$.
3. *such that all variables of* V_I *are assigned in* A

Minimal solutions of a configuration or design problem only consider those assignments for which there exist no other assignment that has less identical variable-value pairs satisfying all constraints. Non-minimal solutions might occur when independent options have to be taken into account. These are options that do not depend on the product structure but only on the customer's choices and preferences. In this case, there exist two similar solutions, one with the option and one without. An algorithm generating only minimal configurations would never generate the same tuple with an additional variable assigned if it was not required. Consider again the mixer example with the constraint that each version of a mixer can be optionally equipped with a display showing the conditions inside the vessel. A constraint like $false \overset{ACT}{\rightarrow} Display$ indicating such an option is always satisfied and would not lead to a problem space containing the variable $Display$. In order to include options in a more deterministic way into the DCSP framework, an auxiliary variable $Vopt$ with values *opt* and *noopt* and an activity constraint $Vopt = opt \overset{ACT}{\rightarrow} Display$ could be added. If the variable $Vopt$ has no value assigned, two minimal solutions would satisfy the constraint, one with $Vopt$ set to *opt* and the display variable active and one with the constraint $Vopt = noopt$[1].

A simple example presented in [Mittal and Falkenhainer, 1990] is given here as illustration.

[1]Mihaela Sabin, University of New Hampshire, mentioned the idea of introducing auxiliary variables to formalize independent options in a personal communication.

X_1	X_2	X_3	X_4
a	d	-	-
a	d	f	-
a	d	-	g
a	d	-	h
a	d	e	g
a	d	e	h
a	d	f	g
a	d	f	h
b	c	e	h
b	c	f	-
b	c	f	h
b	c	f	g

X_1	X_2	X_3	X_4
a	d	-	-
b	c	e	h
b	c	f	-

Table 5.1: *First table: solutions of problem $\mathcal{P}_1$. Second table: solutions for the minimal model of $\mathcal{P}_1$ corresponding to the minimal solutions obtained by Mittal and Falkenhainer. Variables marked by a hyphen are not active.*

Problem $\mathcal{P}_1 = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$

Variables $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ with
$D_1 = \{a, b\}$, $D_2 = \{c, d\}$, $D_3 = \{e, f\}, D_4 = \{g, h\}$
$V_I = \{X_1, X_2\}$

Constraints $\mathcal{C} = \{C^1, C^2, C^3, C^4\}$ with
$C^1_{X_1,X_2} := ((a\ d)(b\ c))$
$C^2_{X_2,X_3,X_4} := ((c\ e\ h)(c\ f\ h)(c\ f\ g)(d\ e\ h)(d\ e\ g)(d\ f\ h)(d\ f\ g))$
C^3: $X_1 = b \stackrel{ACT}{\rightarrow} X_3$
C^4: $X_3 = e \stackrel{ACT}{\rightarrow} X_4$

This problem has a solution set S_1 containing three minimal solutions and a complete solution set S_2 containing all 12 solutions including the three minimal ones. Both sets are presented in Table 5.1. The minimal solutions do not include tuples like $(a\ d\ f), (a\ d\ f\ g)$ etc. because there exists a minimal tuple $(a\ d)$ in S_1.

Specializations of activity constraints have been defined by Mittal and Falkenhainer such as *require not* (RN) or *always require* (AR) constraints. A RN constraint $C \stackrel{RN}{\rightarrow} X$ defined as $C \rightarrow \neg active(X)$ is in our DCSP framework simply expressed by a compatibility constraint combining all tuples of $\overline{C}$ with all values for variable X [Haselböck, 1993]. To put it another way, a variable X is never activated when $\overline{C}$ is true. This can be illustrated by the following example: the constraint $C : X_1 = a \stackrel{RN}{\rightarrow} X_4$ added to the definitions of $\mathcal{P}_1$ produce the same solutions as if the constraint $C(X_1, X_4) : ((b\ g)(b\ h))$ was added to $\mathcal{P}_1$.

An AR constraint is an activity constraint whose condition simply consists of specifying the activity of a set of variables using the predicate *active*. It results in the addition of a variable to the problem space based on the existence of other variables. Such a constraint would for example be $X_4 \stackrel{ACT}{\rightarrow} X_5$, adding X_5 to $\mathcal{P}_1$.

5.4 Solving discrete DCSPs

The solution techniques for discrete DCSPs resemble standard CSP techniques in that they are also based on the enumeration of values. We show here two possibilities of solving a given DCSP over discrete variables. One is based on the original formalism proposed by Mittal and Falkenhainer, the other transforms a DCSP into a standard CSP and uses well-known search algorithms to solve it.

5.4.1 Original DCSP algorithm

An algorithm similar to the one proposed in [Mittal and Falkenhainer, 1990] is presented in Figure 5.1. It is limited to generating minimal solutions. The main procedure **activate-choose** is called recursively after each valid value instantiation. Its main cycle loops until no more constraints can be activated for a given value combination. Only then can the next value be instantiated. A minimal solution is found if all active variables are assigned a value and if there is no other activity constraint whose condition becomes relevant (halting condition for minimal solutions). Forward checking can be added to constraint checking in step 12 in order to remove inconsistent values from the search space. Applying the backtrack algorithm to problem $\mathcal{P}_1$, the following solution trace is obtained:

1. Start with X_1, X_2 activated and C_1 relevant
2. Value assignment $X_1 = a$
3. Value assignment $X_2 = c$ violates the constraint C_1, backtrack ...
4. Value assignment $X_2 = d$
5. Solution found $\{X_1 = a, X_2 = d\}$, backtrack ...
6. Value assignment $X_1 = b$
7. C_3 activates X_3
8. Value assignment $X_2 = c$
9. Value assignment $X_3 = e$
10. C_4 activates X_4
11. C_2 becomes relevant
12. Value assignment $X_4 = g$ violates constraint C_2, backtrack ...
13. Value assignment $X_4 = h$
14. Solution found $\{X_1 = b, X_2 = c, X_3 = e, X_4 = h\}$,
15. Value assignment $X_3 = f$
16. Solution found $\{X_1 = b, X_2 = c, X_3 = f\}$, backtrack ...
17. Value assignment $X_2 = d$ violates constraint C_1, backtrack ...

Mittal and Falkenhainer use an ATMS-based implementation. This has the advantage that each constraint only needs to be run once due to caching. Since their backtrack-algorithm is based on value enumeration, activity constraints might have to be reapplied several

times. ATMS-caching "remembers" the conditions under which an activity constraint has to be applied. The ATMS also prevents cycling because new variables introduced are justified by the values of the variables in the activation condition and such justification labels are kept minimal by the ATMS. Another advantage of the ATMS is that incompatible assignments detected during value enumeration are stored as nogoods and will not be revisited by the backtrack algorithm. The generalization of activation conditions to compatibility constraints makes the use of an ATMS unrealistic because each tuple in the constraint would be a different justification for the introduction of a new variable. A serious drawback is also that value enumeration is no longer practicable when continuous variables are involved. An activity constraint may introduce a splitting in the search space due to its condition. Assume that an activity constraint adds a condenser to the configuration of an industrial mixer only if the vessel volume is larger than 150 liters; i.e. $Vessel.volume \geq 150 \stackrel{ACT}{\rightarrow} Condenser$. Suppose further that the vessel volume varies between 0 and 1000 liters. It can be chosen such that the condition is satisfied or not. Both possibilities should be considered, which leads to a splitting in the search space. In the first space the condition is satisfied and a condenser is added and in the second no condenser must be added. In order to be able to address this case, we need a different search algorithm, that is not based on enumeration of variable values.

5.4.2 Why not use a static CSP formulation ?

In the introduction to this chapter, we mentioned some drawbacks of a transformation from the DCSP formalism into a static CSP. Among others, we stated that the addition of activity constraints does not always result in local changes in the corresponding CSP model. In this section, we will provide evidence for this statement.

The transformation of a DCSP into a static CSP was informally known to be feasible using an additional *NULL* value for any variable that may be part of a solution or not [Haselböck, 1993], [Mittal and Falkenhainer, 1990]. A variable with a NULL value assigned is interpreted in the static model as a variable that is not active. Haselböck also stated that this transformation is not at all *straightforward*. In order to examine the feasibility of this approach, we give here an algorithm for this transformation.

Consider, for example, constraint C_3 of $\mathcal{P}_1$. This constraint requires that any solution containing $X_1 = b$ must have a value for X_3 different from *NULL*. Nothing is said about the activity of X_3 if the activation condition is not satisfied. Thus, this activity constraint can be transformed using the following equivalence:

$$C_{X_1,\ldots,X_i} \rightarrow active(X_n) \Leftrightarrow \neg(C_{X_1,\ldots,X_i} \wedge \neg active(X_n))$$

which translates into C_3': $(b\ e)(b\ f)(a\ N)(a\ e)(a\ f)$ with N being the NULL-value.

The transformation of a DCSP into a static CSP is achieved by algorithm A_1 (Figure 5.2). If each constraint is translated according to algorithm A_1, the solution set comprises all solutions, also the non-minimal ones. Since the value N is just treated like

```
procedure DCSP
begin
 1  V ← V_I
 2  C ← compatibility constraints relevant to V
 3  solution ← ∅
 4  activate-choose(V, C, solution)
end

procedure activate-choose(V, C, solution)
 begin
 1  backtrack ← false
 2  while not(backtrack) do
 3    if ∃ relevant activity constraint in C^A with condition satisfied then
 4      add newly activated variable to V
 5    elif ∃ relevant compatibility constraint c in C^C then
 6      add c to C
 7    elif all variables ∈ V are assigned then
 8      print(solution)
 9      backtrack ← true
10    elif (v ← select next variable from V) ≠ ∅ then
11      value(v) ← choose next assignment for v
12      if not(value(v) = ∅ or value(v) violates C) then
13        solution ← solution ∪ {v = value(v)}
14        activate-choose(V, C, solution)
15      fi
16      backtrack ← true
17    fi
18  od
19 end
```

Figure 5.1: *DCSP algorithm for generating minimal solutions.*

any value in a variable domain, the minimal solution set has to be generated explicitly by comparing each new solution found against the already generated solutions. The transformed CSP of $\mathcal{P}_1$ is for example represented by four variables X_1, X_2, X_3, X_4 with domains $D'_1 = \{a, b\}$, $D'_2 = \{c, d\}$, $D'_3 = \{e, f, N\}$ and $D'_4 = \{g, h, N\}$ and the constraints:

$C'_1(X_1, X_2)$: $((a\ d)(b\ c))$

$C'_2(X_2, X_3, X_4)$: $((c\ e\ h)(c\ f\ h)(c\ f\ g)(d\ e\ h)(d\ e\ g)(d\ f\ h)(d\ f\ g)(d\ N\ N)$ $(d\ N\ h)(d\ N\ g)(d\ f\ N)(d\ e\ N)(c\ e\ N)(c\ N\ N)(c\ N\ h)$ $(c\ N\ g)\ (c\ f\ N))$

$C'_3(X_1, X_3)$: $((b\ e)(b\ f)(a\ N)(a\ e)(a\ f))$

$C'_4(X_3, X_4)$: $((e\ g)(e\ h)(f\ g)(f\ h)(f\ N)\ (N\ g)(N\ h)(N\ N))$

Its solutions are the same as those of set S_1 given in Table 5.1. In order to generate only

Algorithm A_1

1. Add a NULL-value N to the domain of each variable X_i that is not in the set of initial variables such that $D'_i = D_i \cup \{N\}$.
2. For each compatibility constraint $C_{X_{i_1} \ldots X_{i_j}}$ do:
   Let T be the set of allowed tuples defined by the constraint. Additional allowed tuples T_{ad} are defined by $D'_{i_1} \times \ldots \times D'_{i_j} \backslash T$ such that at least one NULL value is in the tuple. The tuples of the transformed constraint are $T_{ad} \cup T$.
3. For each activity constraint: $C_{X_{i_1} \ldots X_{i_j}} \rightarrow active(X_k)$ do:
   Build a $(i_j + 1)$- ary constraint. The not allowed tuples T_{na} are the cross-product of $D_{i_1} \times \ldots \times D_{i_j} \times \{N\}$ such that $C_{X_{i_1} \ldots X_{i_j}}$ is true. Take the tuples $T_a = D'_{i_1} \times \ldots \times D'_{i_j} \times D'_k \backslash T_{na}$ as the set of allowed tuples to define the transformed constraint.

Figure 5.2: *Algorithm transforming a DCSP into a CSP.*

Algorithm A_2

1. For each activity constraint $C_{X_{i_1} \ldots X_{i_j}} \rightarrow active(X_k)$ generate additionally a require not constraint $\neg C_{X_{i_1} \ldots X_{i_j}} \rightarrow \neg active(X_k)$
2. proceed by algorithm A_1

Figure 5.3: *Transforming a DCSP into a CSP defining only minimal solutions.*

minimal solutions, we have to add the following two *require not* constraints to $\mathcal{P}_1$ defining a new problem $\mathcal{P}_2$:

C''_3: $X_1 = a \rightarrow \neg active(X_3)$
C''_4: $X_3 = f \rightarrow \neg active(X_4)$

An exclusive interpretation of the form $C_{X_{i1}, \ldots, X_{ij}} \Leftrightarrow active(X_k)$ is used for each activity constraint. Intuitively, a variable should be activated if and only if the condition is true. This can be transformed into a static CSP according to algorithm A_1. For example:

$C'''_3(X_1, X_3)$: $((b\ e)(b\ f)(b\ N)(a\ N))$
$C'''_4(X_3, X_4)$: $((f\ N)(N\ N)(N\ g)(N\ h)(e\ N)(e\ g)(e\ h))$

Note that the constraints C'''_3 and C'_3 as well as C'_4 and C'''_4 can be merged into one constraint by taking the intersection of their tuples respectively. Solving the problem results in the three minimal solutions of solution set S_2. The transformation of the original DCSP into a static CSP generating minimal solutions is presented in algorithm A_2 (Figure 5.4.2).

Algorithm A_2 is correct as long as each variable is activated by maximally ONE activity constraint. Otherwise, constraints activating the same variable have first to be collapsed into one activity constraint before they can be transformed into a compatibility constraint. Consider example $\mathcal{P}_3 = \mathcal{P}_1 \cup C_5$ with $V_I = \{X_1, X_2, X_5\}$ and $D_5 = \{i, j\}$ and C_5: $X_5 = i \rightarrow active(X_3)$. Constraints C_3 and C_5 both activate X_3 and according to the definition

Expression 1: IF mixing-task = dispersion THEN add condenser

$$C_{MT,C} := \{ \begin{array}{l} (dispersion\ C1) \\ (dispersion\ C2) \\ (blending\ NULL) \\ (suspension\ NULL) \\ (entrainment\ NULL)\} \end{array}$$

Expression 2: IF *vessel.volume* ≥ 150 THEN add condenser

?

Figure 5.4: *The first expression reasoning about the existence on the condenser variable can be transformed into a discrete constraint. We suppose here that a condenser can be of type C1 or C2. The second expression links a continuous constraint to the existence of a variable and its translation would require a mixed constraint formulation with NULL values.*

of an activity constraint, the condition for not activating X_3 is $\neg(X_1 = b \wedge X_5 = i)$. In other words, not only $\{X_1 = a, X_5 = j, X_2 = d\}$ is a minimal solution but also $\{X_1 = b, X_5 = j, X_3 = f, X_2 = c\}$, $\{X_1 = b, X_5 = i, X_3 = f, X_2 = c\}$ and $\{X_1 = a, X_5 = i, X_3 = f, X_2 = d\}$ are minimal. This means that an activity constraint can activate a variable independently of other constraints. If we added the require not constraints directly and transformed each of the constraints individually, we would loose the minimal solutions $\{X_1 = b, X_5 = j, X_3 = f, X_2 = c\}$ and $\{X_1 = a, X_5 = i, X_3 = f, X_2 = d\}$.

A local change in the DCSP in form of the addition of an activity constraint may imply a non-local change in the corresponding CSP formulation because activity constraints generating the same variable have to be combined. In terms of maintainability and modularity, this approach is therefore less attractive. The transformation presented in this section is of course only possible under the assumption that all activity constraints are given at the beginning. It does not allow for an interactive introduction of such constraints during problem solving, because the transformation step would have to be repeated at each introduction of a new variable. The assumption that all activity constraints are known from the beginning complies with the intentional definition of a design task as discussed in the introduction of this chapter.

In addition to these drawbacks, the introduction of an optional part may also depend on continuous values in which case a kind mixed constraint would result combining a continuous constraint with a discrete value (Figure 5.4).

Before we show how to identify solutions to general DCSPs, we would like to give a formalism in which a design task can be specified easily and a way to convert the input description into a DCSP.

5.5 Formalizing a design task

As already discussed in the introductory chapters, a convenient formalism to describe a design task is the Object Oriented framework. In this framework, information common to several objects is collected in a generic description. This makes reuse and maintenance easier than in other formalisms. The OO-framework is based on important concepts like classes, instances and their properties and type- respectively part-of hierarchies. In terms of constraint satisfaction, component instances and properties are variables and type or part-of hierarchies are constraints between components. Also, constraints between component properties and between properties and components can be formulated either as compatibility constraints or as activity constraints defining functional dependencies. Such constraints have to be specified in addition to the Object Oriented description of the structure of the artifact.

In this section, we show how the Object Oriented input description of a task can be transformed into a DCSP, thus providing an interface between the input description of the task and its resolution algorithm.

5.5.1 A design description

The description of a design task consists of a set of component classes, often called catalog, in which each class is described by a set of properties plus constraints over classes and properties. An example is the catalog of mixer components in the appendix A.1 in which the mixer and the mixing task are specified as well as all subcomponents occuring in the configuration. Relations between component classes and between component properties constrain the design. Each vessel has for example the property volume and only an elliptical vessel has additionally a small radius called sradius. Another example is the constraint that only a vessel used for chemical reactions needs a cooler. Such relationships express either structural constraints like the two first examples or a functional dependency like the last example. In the next sections, the different parts of the design description are detailed.

5.5.2 Components

The unit to describe a part in design is typically a component. A component may correspond to a structural or an abstract concept as, for example, $Vessel$ and $MixingTask$. In general, a component is an object consisting of basic and composed properties. The basic properties of a component can be seen as the components of smallest granularity. They can be of type discrete, real, integer or array. composed properties are themselves components. Components of similar behavior are described once by a template called *type* (in OO-technology a class). Such a type specifies all properties of a component. An example is the definition of vessel type for a mixer:

```
Type Vessel:
  domain : discrete : {hemispherical,elliptical,cylindrical}
  volume : real : [0,1000]
  diameter : real : [0,1000]
```

```
height : real : [0,1000]
opt1 : Cooler [optional]
opt2 : Condenser [optional]
```

In this example, *Vessel* is a component type that has the basic properties *volume*, *diameter* and *height*. *coolerElement*, *condenserElement* are composed properties because they are themselves components. An instantiation of a component type creates a specific component instance or component for short, i.e. an object with its own identity possessing exactly the properties specified by its type. In the DCSP framework, each component instance as well as each property of an instance is simply a variable. A variable in the DCSP that stands for a component instance represents a collection of variables that are the properties of the component. An activity constraint on this variable without condition can thus be used to generate the collection.

5.5.3 Relations between components or between components and properties

Instead of declaring relations on each of the instances of component types, one would rather declare generic relations on component types and properties of component types whenever possible. This can be realized by generic constraints introduced in [Haselböck, 1993]. A *generic constraint* is a constraint that is specified not directly on variable instances but on types of variables. It has to be applied to each set of variables whose types comply with the types over which the generic constraint is defined.

Definition 5.6 (Generic constraints) *Let $C_{X_1 \dots, X_k}$ be a generic (discrete or continuous) constraint defined on meta-variables $X_1, \dots, X_k$ representing component types, and V^C the set of all current component instances of $\mathcal{V}$. A generic constraint $C_{X_1,\dots,X_k}$ is satisfied by the assignments $V_{ji} = v_{ji}$, $V_{ji} \in V^C$ if and only if*

$$\forall_i \; V_{1i}, \dots V_{ki} : \; C_{V_{1i},\dots V_{ki}}(v_{1i}, \dots, v_{ki}) \wedge X_j = type(V_{ji}) \tag{5.1}$$

Such a constraint defines a relation referring to types of variables (the component types) instead of the variables themselves. When two variable sets of the types required by the generic constraint exist, a constraint is to be created for each of the variable sets separately. If the constraint involves a property p of a component of type T, one of the meta-variables, on which the constraint is declared, is $T.p$ and the constraint is applied to all variables representing the property p of type T. An example of a constraint defined on meta-variables is the relation $C_{MT.slurry\,pressure,V}$:

$$\begin{aligned} C_{MT.slurry\,pressure,V} := \;& \{(high\ hemispherical) \\ & (high\ elliptical) \\ & (low\ cylindrical)\} \end{aligned}$$

between the vessel type and the slurry pressure of a mixing task.

A part-of relationship between objects expresses that one instance is a collection of other instances. Part-of hierarchies show the physical decomposition of the designed product. Such a hierarchy shows all instances created for one configuration and how subcomponents are linked to components. See Figure 2.1 for a physical decomposition of a mixer. The description of a component type inherently shows one level of this hierarchy by enumerating all composed properties the specific component consists of. The part-of hierarchy is successively generated from the instantiation of type templates using activity constraints. Since the existence of an instance of type vessel implies the existence of its properties, activity constraints can be used to generate the properties of a component automatically. Moreover, this generation mechanism can be generalized to any object of a given type [Haselböck, 1993]. For each property p of the type T we define an activity constraint as follows:

$$T \stackrel{ACT}{\rightarrow} p \tag{5.2}$$

where the condition simply requires an instance of type T to be active and p describes the type of property to be created and its name in the form $name : type : domain$ (the name is abbreviated by its initials in case of component types). For example,

$$V \stackrel{ACT}{\rightarrow} diameter : real : [0, 1000]$$
$$V \stackrel{ACT}{\rightarrow} volume : real : [0, 1000]$$

For two instances of a vessel, named V_1 and V_2, two independent properties $V_1.volume$ and $V_2.volume$ are created. The generation properties of components automatically stops at the leaf components of the part-of hierarchy because the domain of a variable representing a leaf component is simply the component type itself.

The properties *coolerElement* and *condenserElement* are optional. This means that not every vessel has a cooler or a condenser and that corresponding components should not be created by a simple activity constraint without condition like the other properties. The creation of these elements is subject to functional dependencies as we show next.

Some relations between components in a configuration are not structural but of a rather functional nature. The goal of a design task is to find an artifact that fulfills a set of functions. Hence, these functions are to be implemented by different component sets. [Mittal and Falkenhainer, 1990] found that each function could be represented by one key component plus varying sets of secondary components dependent on this key component. They realized that activity constraints are a means for expressing such functional dependencies because the additional components can be generated under given conditions. The function of mixer as a catalyst for chemical experiments, which requires the additional component cooler, is for example implemented by the activity constraint

$$Mixer = reactor \wedge Vessel \stackrel{ACT}{\rightarrow} Cooler$$

More generally, the condition under which an additional component of T_2 type is required must be specified in an activity constraint with an activation condition in the form of

$$condition \wedge T_1 \stackrel{ACT}{\rightarrow} T_2 \tag{5.3}$$

Activity constraints are defined on component types and no longer individual components such that they can be applied to any set of instances of the required types similar to generic compatibility constraints. Type $T1$ indicates also the structural dependency; e.g. to which component the new part has to be attached to. In the example, it is the vessel that has to be generated first such that the condenser can be attached to it. This variable is only needed to correctly determine when the constraint can be applied (cf. Combining activity constraints, Section 5.6.2). The reason why one may specify explicitly activity constraints additionally to the OO-description of the design task is that not all components can be generated from the part-of or type hierarchy directly. There may exists functional dependencies in the system across several levels of these hierarchies. A good example is the mixer vessel requiring a cooler if the mixer is a reactor or the requirement of a condenser if the vessel volume is large. In the OO-formalism this is solved by creating a new type of mixer-with-condenser inheriting all properties from the mixer and a component type condenser. A constraint on the mixer then restricts the mixer type to mixer-with-cooler whenever the mixer is a reactor. In the same way, the new types mixer-with-cooler and mixer-with-cooler-and-condenser should be created because a priori all combinations of optional components are possible. This prealable combination of all possible components is one of the drawbacks in OO-technology. Specifying the two activity constraints that create a cooler and a condenser independently is more declarative than the Object Oriented approach, which has to foresee all possible solution types. With activity constraints functional dependencies can be created easily between components, which are much more difficult to formalize in the OO-formalism.

Type-of hierarchies define relations between classes. They group similar classes into a hierarchy such that redundant information can be removed. A type-subtype relationship indicates that the subtype is more specific than the type. A subtype of a component type inherits all its properties and relations on it, but it may add some more characteristics. A relation type-of can be translated into a constraint formalism by regarding the subtypes as values for the type. A type hierarchy thus corresponds to a hierarchical domain in the CSP formalism [Mackworth et al., 1985],[Sabin and Freuder, 1996]. Hierarchical domains can be exploited in consistency algorithms in two ways:

1. if a value high in the hierarchy is found inconsistent, so are its descendents
2. if a value low in the hierarchy is found consistent, all its ancestors are consistent

In the mixer example, there exist three subtypes on the vessel indicated in the domain property of the vessel: Hemispherical, Elliptical and Cylindrical.

```
Type Elliptical of Vessel
  sradius : real : [0,1000]
  bottomarea : real : [0,1000]
```

In this example, the subtype *Elliptical* must be specified explicitly because this type needs the additional properties *sradius* and *bottomarea* such that the vessel volume can be correctly derived from the form of its vessel. The description of the other subtypes

Hemispherical and *Cylindrical* on the other hand is identical with the description of Vessel. Additional properties of a subtype can be easily created by activity constraints. Suppose that a given type T has subtypes T_1, T_2. T_1 has an additional property p. The activity constraint

$$T = T_1 \stackrel{ACT}{\rightarrow} p \tag{5.4}$$

generates the property. For the example above

$$V = elliptical \stackrel{ACT}{\rightarrow} sradius : real : [0, 1000]$$

Hence, subtypes can be understood as values of a variable representing the instance of a type.

5.5.4 Identical components

Identical components are components of the same type appearing in the same configuration. Examples of identical components are engines in train compositions, frames and modules in telephone switching system [Haselböck, 1993] or piers in bridge design. It is useful to group identical components into an array. An individual component is then referred to by an index. In bridge design for example, *piers* would be declared as array and $piers[1], \ldots, piers[n]$ represent the individual components of the array.

It is part of the design process to derive variants in form of different cardinalities of an identical component set such that the constraints on the design are satisfied. Activity constraints are one way of generating new components. In the following, we investigate their use for the generation of entire component sets. Suppose that a variable *piers* has been declared as array and *Pier* is the component type of its elements. One way of formulating the creation of identical elements is to control the generation directly from the array, because the existence of an array variable implies the existence of array elements.

$$piers \stackrel{ACT}{\rightarrow} \forall_{i=1}^{n} piers[i] : Pier$$

More generally, we call *generator* an the activity constraint of the form

$$array\, variable \stackrel{ACT}{\rightarrow} \forall_{i=1}^{n} element[i] : ElementType \tag{5.5}$$

which allows for the generation of n array elements. It has to be noted that n must be bound to a variable of the problem with a label indicating a maximal value for n in order to guarantee termination. We discuss this issue of *restricted variable generation* further in the last section of this chapter.

As a recapitulation, activity constraints without condition generate the properties of a component (Rule 5.2), activity constraints with a condition specialize in a type hierarchy (Rule 5.4) or generate optional components according to a functional dependency (Rule

5.3). Compatibility constraints, which can be specified at any level in the hierarchies between any set of components or properties, restrict the possibilities of further specialization in the part and type-of hierarchies and provide values for the properties. Rules 5.1 to 5.5 serve to translate an Object Oriented description together with additional functional dependencies and compatibility constraints of the design task into a DCSP transparently to users. They do not have to understand the DCSP formalism in order to specify their task.

5.6 Generating problem spaces of a DCSP

Our algorithm for solving DCSPs intends to enhance previous algorithms. First, it accepts activation conditions that are constraints and DCSPs containing continuous constraints as input. Second, the resolution algorithm is no longer based on enumeration but it is constraint-driven. The idea is to construct explicitly the different problem spaces given in the DCSP formulation in a first step and to use then conventional techniques for standard CSPs in order to identify solutions within these problem spaces. The advantage is that the solutions can be presented in a more structured way because the problem spaces have been identified before the exhibition of an individual solution and each solution refers to its problem space. Furthermore, additional decisions on value assignments can be taken as soon as problem spaces have been identified. The different spaces can be compared and provide more knowledge about the structure of the given problem. In the following sections, we discuss the first step of problem space generation. It is based on a constructive interpretation of activity constraints.

5.6.1 Activity constraints

According to the definition of an activity constraint, either the activation condition is relevant and the new variable 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 least two different problem spaces. This observation helps us to specify an incremental algorithm that generates successively the different subspaces consisting of varying sets of variables and constraints that represent the different CSPs. In order to study the addition of a single activity constraint to a single problem space, we first need to define the complement of a constraint C:

Definition 5.7 (Complement of a constraint) *The complement $\overline{C}$ of a constraint C is defined by the subset of the Cartesian product of the variable domains that are the not allowed value combinations.*

The complement exists under the assumption of a closed world; i.e. those tuples that are not given in the constraint are disallowed. It can be represented for a discrete constraint as the set of disallowed tuples and for a continuous constraint as the complement of the constraint regions.

Let P be the current problem space, which consists of a set of variables $\mathcal{V}_P$, a set of relevant constraints $\mathcal{C}_P$, and $C \stackrel{ACT}{\rightarrow} X$ 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 $C \stackrel{ACT}{\rightarrow} X$ 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
  $\mathcal{V}_{P_3} = \mathcal{V}_P$
  $\mathcal{C}_{P_3} = \mathcal{C}_P \cup \{\overline{C}\}$

Only two of the three subspaces created by an activity constraint contain minimal solutions.

Lemma 5.1 *Given a problem space P and an activity constraint $ac : C \stackrel{ACT}{\rightarrow} X$ such that $Vars(C) \subseteq \mathcal{V}_P$. Only the subspaces P_1, P_3 of the DCSP: $P \wedge ac$ contain minimal solutions.*

Proof: Let $C \stackrel{ACT}{\rightarrow} X$ be an activity constraint.
Since all variables of C are active in P, three subspaces exist in $P \wedge C \stackrel{ACT}{\rightarrow} X$:
$P_1 : \mathcal{V}_{P_1} = \mathcal{V}_P \cup \{X\}, \mathcal{C}_{P_1} = \mathcal{C}_P \cup \{C\}$
$P_2 : \mathcal{V}_{P_2} = \mathcal{V}_P \cup \{X\}, \mathcal{C}_{P_2} = \mathcal{C}_P \cup \{\overline{C}\}$
$P_3 : \mathcal{V}_{P_3} = \mathcal{V}_P, \mathcal{C}_{P_3} = \mathcal{C}_P \cup \{\overline{C}\}$
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 because $\forall s\ s[Vars(C)] \in \overline{C} \leftrightarrow s[Vars(C)] \notin C$ where $s[Vars(C)]$ denotes those values of the tuple s that are values for the variables in $Vars(C)$ ▲

Corollary 5.1 *Let $C \stackrel{ACT}{\rightarrow} X$ be an AR activity constraint with a condition C: $active(Y)$, $Y \neq X$ such that Y is active in the current problem space P. Then, only one subspace is created: $P_1 = P \cup \{X\}$*

Proof: Follows directly from 5.1 and the fact that C is satisfied in P ▲

Search for minimal solutions is in general restricted to the subspaces P_1 and P_3 generated by an activity constraint, P_1 is additionally constrained by the activation condition C and P_3 by its complement $\overline{C}$. A discrete activation condition can be any combination of

[2] According to definition 5.3, 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 $C \stackrel{ACT}{\rightarrow} X_1, \ldots, C \stackrel{ACT}{\rightarrow} X_n$.

allowed values. If C is a continuous constraint, we require it to be an inequality defining a region of allowed values. This allows us to represent the complement of a continuous constraint as inequality. A continuous condition cannot be an equality because we are not able to treat constraints of the form $E \neq 0$ with E a constraint expression, which would be the complement of the equality. The activation condition is also restricted to a single constraint. We do not allow an activation condition to be a conjunction of several constraints $C := C_1 \wedge \ldots \wedge C^m$ otherwise the complement results in a further splitting of the problem space P_3 because $\overline{C}$ is in that case the disjunction $\overline{C^1} \vee \ldots \vee \overline{C^m}$.

In the following section, we show how to combine activity constraints thereby identifying problem spaces in which the minimal solutions of a DSCP are found.

5.6.2 Combining activity constraints

A DCSP is a conjunction of activity and compatibility constraints. A compatibility constraint is trivially satisfied if one of its variables is not active. It has only to be taken into account in problem spaces in which all of its variables are active. Similarly, an activity constraint is only to be considered in problem spaces in which its condition is relevant. If one of the condition's variables is not active, the activity condition is trivially satisfied. We have seen in the preceding section that an activity constraint whose condition is relevant splits a problem space into several subspaces. In order to compute a conjunction of activity constraints, two problems have to be solved:

1. how to combine subspaces produced by a set of activity constraints
2. how to find an ordering of the activity constraints such that each constraint has to be considered only once

The second point is important because there exists an implicit dependency between activity constraints. It makes only sense to apply an activity constraint as soon as its condition becomes relevant. A situation occuring frequently is that a variable introduced by an activity constraint is necessary for the next activity constraint to become relevant. Even worse, cyclic dependencies like $C_X \stackrel{ACT}{\rightarrow} Y$ and $C_Y \stackrel{ACT}{\rightarrow} X$ may occur as we will see later in this chapter.

First, let us consider the combination of a set of activity constraints with activation conditions. We assume for the moment that all activation conditions are relevant and that there is no cyclic dependency between constraints.

Theorem 5.1 *The minimal solutions of the conjunction of a set of activity constraints $ac_i : C_i \stackrel{ACT}{\rightarrow} X_i$ in a DCSP are found in the cross-product formed of all subspaces P_{i1}, P_{i3}.*

Proof: Let P be a problem space defined by the initial variables V_I and $ac_1 : C_1 \stackrel{ACT}{\rightarrow} X_1$ the first activity constraint of the given set $\{ac_i\}$. By lemma 5.1, only P_{11} and P_{13} contain minimal solutions under the assumption that the condition is relevant in P. Suppose that the theorem is true for the conjunction of i activity constraints and let $\{P_i\}$ be the resulting minimal problem spaces. We will show that it is also true for the $(i+1)$th activity

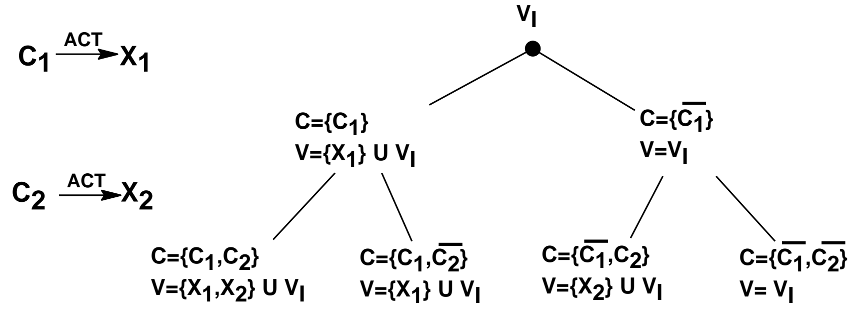


Figure 5.5: *The problem spaces created by combining two activity constraints of a DCSP. It is assumed here, that the variables of C_1 and C_2 are active. First, the constraint $C_1 \stackrel{ACT}{\rightarrow} X_1$ is added to the set of initial variables V_I, then $C_2 \stackrel{ACT}{\rightarrow} X_2$.*

constraint. Since we assume that all variables of the condition C are active in a problem space P_i, the subspaces generated by $C_{i+1} \stackrel{ACT}{\rightarrow} X_{i+1}$ are:
$P_{(i+1)1} : \mathcal{V}_{P_{(i+1)1}} = \mathcal{V}_{P_i} \cup \{X_{i+1}\}, \mathcal{C}_{P_{(i+1)1}} = \mathcal{C}_{P_i} \cup \{C_{i+1}\}$
$P_{(i+1)2} : \mathcal{V}_{P_{(i+1)2}} = \mathcal{V}_{P_i} \cup \{X_{i+1}\}, \mathcal{C}_{P_{(i+1)2}} = \mathcal{C}_{P_i} \cup \{\overline{C_{i+1}}\}$
$P_{(i+1)3} : \mathcal{V}_{P_{(i+1)3}} = \mathcal{V}_{P_i}, \mathcal{C}_{P_{(i+1)3}} = \mathcal{C}_{P_i} \cup \{\overline{C_{i+1}}\}$
Clearly, solutions of $P_{(i+1)2}$ are supersets of solutions in $P_{(i+1)3}$, because $\mathcal{V}_{P_{(i+1)3}} \subseteq \mathcal{V}_{P_{(i+1)2}}$ and $\mathcal{C}_{P_{(i+1)3}} = \mathcal{C}_{P_{(i+1)2}}$. Furthermore, no solution s of $P_{(i+1)3}$ can be a subset of a solution in $P_{(i+1)1}$ and vice versa because $\forall s\ s[Vars(C_{i+1})] \in \overline{C_{i+1}} \leftrightarrow s[Vars(C_{i+1})] \notin C_{i+1}$.
By induction, the theorem is true for any set of activity constraints ▲

The tree in Figure 5.5, which we will call *combination tree*, illustrates the subspaces produced by the combination of two activity constraints. By theorem 5.1 minimal solutions are generated by considering only activity constraints whose condition is relevant to the given problem space. Actually, the problem space generated by an activity constraint when the condition is not relevant is the same as the parent problem space because the activity constraint is satisfied trivially. One inconvenience of combining subspaces is that some subspaces do not add information:

Corollary 5.2 *Given a problem space P with X active and an activity constraint ac : $C \stackrel{ACT}{\rightarrow} X$ 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, $P = P_1 \cup P_3$ ▲

It is easy to detect and prevent this unnecessary splitting. If the variable that is to be activated by the current constraint is already active, no splitting in subspaces takes place.

The application of an activity constraint should be delayed until all the variables of its condition are active. Otherwise, after each variable activation, all activity constraints have to be checked if their condition has become relevant. This suggests that there may exist a certain order in which these constraints should be combined. To determine such an order, we introduce the dependency relation DR between two activity constraints.

Definition 5.8 (dependency relation between activity constraints) *An activity constraint* $C_n \stackrel{ACT}{\rightarrow} X_n$ *is* dependent *on a second activity constraint* $C_1 \stackrel{ACT}{\rightarrow} X_1$*, denoted by* $C_n \stackrel{ACT}{\rightarrow} X_n \; DR \; C_1 \stackrel{ACT}{\rightarrow} X_1$ *if there exists a chain of activity constraints* $C_i \stackrel{ACT}{\rightarrow} X_i, i = 2, \ldots, n-1$ *such that* $X_i \in Vars(C_{i+1}), i = 1, \ldots, n-1$.

If $X_1 \in Vars(C_n)$, the constraint $C_n \stackrel{ACT}{\rightarrow} X_n$ is *directly dependent* on $C_1 \stackrel{ACT}{\rightarrow} X_1$. If, additionally, $C_1 \stackrel{ACT}{\rightarrow} X_1$ is dependent on $C_n \stackrel{ACT}{\rightarrow} X_n$, there is a cycle in the chain of dependencies and we speak of *cyclic dependency.* If all activity constraints are reconsidered at each step, a cycle in the dependencies would lead to an infinite regeneration of variables that are already active. Consider the two activity constraints $C^1_X \stackrel{ACT}{\rightarrow} Y$ and $C^2_Y \stackrel{ACT}{\rightarrow} X$ where the condition C^1 is defined over X and C^2 over Y and the initial problem space P. If $\mathcal{V}_P$ contains none of the variables X or Y, none of the activity constraints is considered and cycling is avoided. On the other hand, if $\mathcal{V}_P$ contains X, $C^1_X \stackrel{ACT}{\rightarrow} Y$ generates the minimal subspaces of which P_1 contains $\{X, Y\}$. The second activity condition C^2_Y is relevant in P_1 and again leads to the generation of two minimal subspaces that both already contain X and so on. Corollary 5.2 avoids such a regeneration and serves as a halting criterion in case of cyclic dependencies.

Lemma 5.2 *Given a set of activity constraints in which no cyclic dependency occurs. Then, the dependency relation establishes a strict partial order*[3] *on the set activity constraints.*

Proof: Let $ac_i : C_i \stackrel{ACT}{\rightarrow} X_i, i = 1, 2, 3$ be three activity constraints. It is sufficient to show that DR is irreflexive and transitive.

1. $\neg(ac_i \; DR \; ac_i)$ because $X_i \notin Vars(C_i)$ by definition. DR is irreflexive.
2. $ac_3 \; DR \; ac_2 \wedge ac_2 \; DR \; ac_1 \rightarrow ac_3 \; DR \; ac_1$ because $X_3 \in Vars(C_2), X_2 \in Vars(C_1)$ is a chain. DR is transitive.

Asymmetry[4] follows from irreflexivity and transitivity. ▲

[3] The designation "ordering" for this type of relation is somewhat misleading because it is an irreflexive relation [Schmidt and Ströhlein, 1988]. There seems to be no universal designation for this type of relation. In [Schmidt and Ströhlein, 1988], an irreflexive, asymmetric and transitive relation is called a strict ordering; in [Golumbic, 1980], it is called a strict partial order.

[4] Asymmetry holds between two activity constraints if $ac_1 DR \; ac_2 \rightarrow \neg(ac_2 DR \; ac_1)$.

The dependency relation establishes a partial order between the activity constraints, which guides the application of activity constraints. This relation can be represented in a directed graph $\mathcal{G} = \langle \mathcal{X}, \mathcal{U} \rangle$ where:

- $\mathcal{X}$ is the set of activity constraints plus a root node which represents V_I
- $\mathcal{U}$ is defined by the relation DR. There exists a directed edge from one node to another if the second node is directly dependent on the first. The arrow always points to the dependent node. Transitive edges are not represented in the graph.

In Figure 5.6, three dependency graphs are shown with their corresponding combination tree. At each node of the tree, we only show the variables and constraints that are added by the current activity constraint. The leaf nodes represent the final problem spaces. In order to reconstruct the set of variables and constraints of such a leaf problem space, one has just to follow the path from the root to this leaf and to gather all variables and constraints at the intermediate nodes. Graph a) represents the dependency relation between the two activity constraints of example $\mathcal{P}_1$ given in Section 5.3. Figure b) shows the graph for a similar problem $\mathcal{P}_2$ with the additional constraints $X_2 = c \overset{ACT}{\rightarrow} X_5$, $X_5 = i \overset{ACT}{\rightarrow} X_3$ and X_5 with domain $\{i, j\}$. A third problem $\mathcal{P}_3$ is derived from $\mathcal{P}_1$ by adding the constraint $X_4 = g \overset{ACT}{\rightarrow} X_3$. On the graph of $\mathcal{P}_3$, an ordering of the activity constraints is not possible because there exists a cyclic dependency between the last two constraints. A way to resolve cyclic dependencies is to remove them by transforming the graph. Cyclic parts in the dependency graph can be collapsed into a super-node. The problem of ordering activity constraints can be translated into the following steps:

1. Remove transitive edges from the original graph (see [Schmidt and Ströhlein, 1988], page 37)
2. Eliminate cycles in the original graph by collapsing all nodes in such cycles into a super-node.
3. Find a partial order on the resulting acyclic graph.
4. Convert the partial order into a full order on the nodes, traverse the graph in this order and apply the activity constraints.

The advantage of the graph-based model is that the detection of cycles in the dependency relation is possible. Since there may be exponentially many cycles, it is indicated to identify sets of nodes that contain at least one cycle. Such a set of nodes is called *strongly connected* (Appendix D). A simple depth-first traversal of the nodes in a directed graph allows us to identify all strongly connected components in a graph in $\mathcal{O}(max(|\mathcal{X}|, |\mathcal{U}|))$ [Gondran and Minoux, 1986]. From these strongly connected components, a reduced graph $\mathcal{G}_R$ is built as follows:

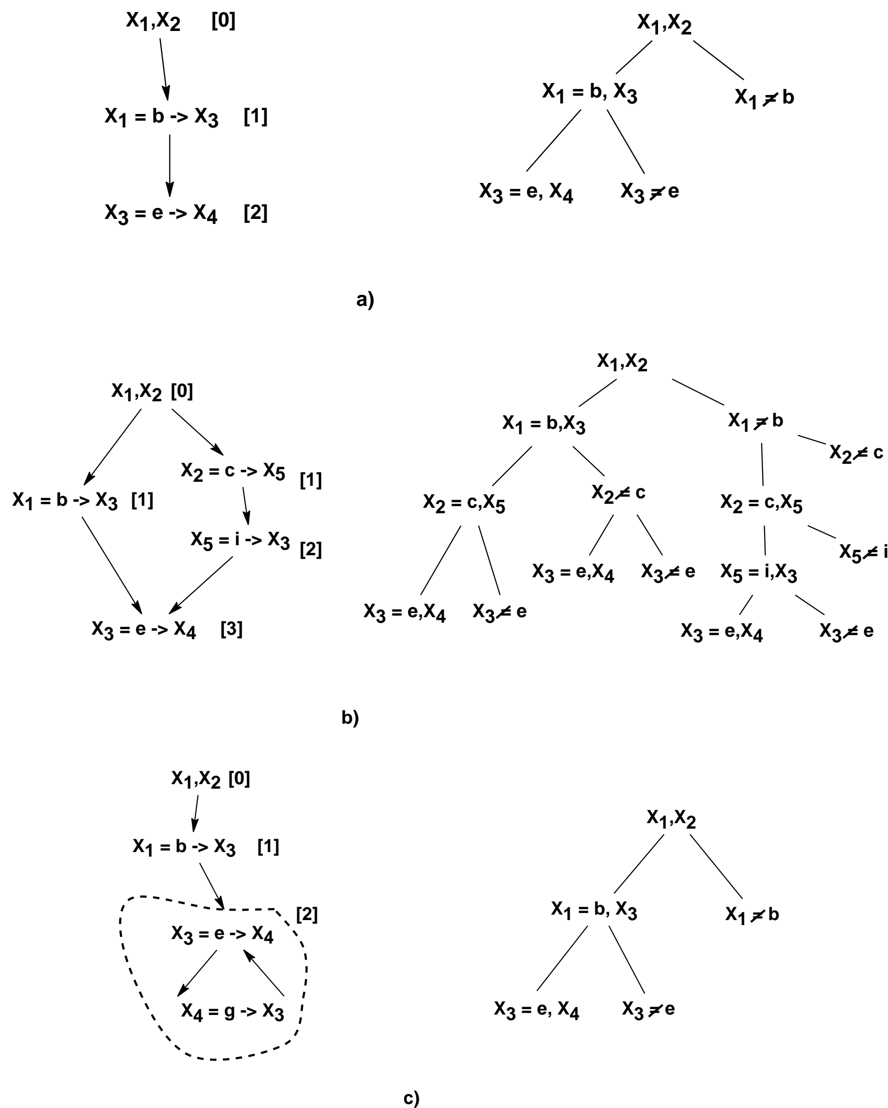


Figure 5.6: *Dependencies in a DCSP represented as directed acyclic graph. The right hand sides show how problem spaces are combined. At each node only the additional variables and constraints are shown. The resulting problem spaces are the leaf nodes of the trees.*

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

The resulting graph is acyclic by construction. In this acyclic graph $\mathcal{G}_R$ a partial order on the nodes exists according to Lemma 5.2 and the reachability of all nodes from the root node can be studied more easily. A node M is *reachable* from a node N if there exists at least one directed path from N to M. In general, more than one path may exist. To compare them, each directed edge is labeled with a unit length. When several edges enter a node, the corresponding activity constraint can safely be applied as soon as the last of

the node's predecessors has been considered. The distance of the current node to the root node is thus 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 minimal bound for when the current node can safely be applied. Before activating the new variable of that activity constraint, it is still to be tested that all the variables of the activation condition are active. Nevertheless, as all predecessor nodes have been reached and their corresponding variables activated, all conditions for the current node to be fired have been satisfied.

The time complexity of finding the longest path from the root node to each other node in a directed acyclic graph is $\mathcal{O}(|\mathcal{X}| \times |\mathcal{U}|)$. The shortest path algorithm of Bellman [Gondran and Minoux, 1986] is adapted to compute the longest path from the root node to each node by labelling 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. The explicit generation of problem spaces is then conducted by traversing the acyclic graph in the given order applying the activity constraint at each node. When a node in $\mathcal{G}_R$ corresponds to a single activity constraint, it is applied to each already existing problem space, which doubles the number of spaces in the worst case. Within a super-node, no ordering between the activity constraints can be established. Thus, all constraints of the super-node have to be searched for the next relevant activity constraint. When the last constraint in a cycle is reached, it is not applied to problem spaces that already contain the variable to be activated according to Corollary 5.2 thus breaking the cycle.

In the three examples in Figure 5.6, the longest distance from the root node to each node is indicated in brackets. a) shows the problem spaces of $\mathcal{P}_1$. For graph b), the activity constraints will be ordered $X_1 = b \stackrel{ACT}{\rightarrow} X_3$, $X_2 = c \stackrel{ACT}{\rightarrow} X_5$, $X_5 = i \stackrel{ACT}{\rightarrow} X_3$, $X_3 = e \stackrel{ACT}{\rightarrow} X_4$. The essential characteristic of this ordering is that the variable X_4, which is dependent on X_3, is not introduced until all possibilities of generating X_3 are exhausted. The elements $X_1 = b \stackrel{ACT}{\rightarrow} X_3$ and $X_2 = c \stackrel{ACT}{\rightarrow} X_5$ have the same distance from the root node, they are *incomparable* in the partial order. It does not matter whether node $X_1 = b \stackrel{ACT}{\rightarrow} X_3$ is treated first or node $X_2 = c \stackrel{ACT}{\rightarrow} X_5$. Furthermore, node $X_5 = i \stackrel{ACT}{\rightarrow} X_3$ does not have to be applied to the subspaces containing X_3 already. In example c) the activity constraints $X_3 = e \stackrel{ACT}{\rightarrow} X_4$ and $X_4 = g \stackrel{ACT}{\rightarrow} X_3$ become a super-node of the reduced graph, which is at a distance of 2. If the super-node is reached during traversal, $X_3 = e \stackrel{ACT}{\rightarrow} X_4$ is the first constraint that can be applied. Hence, $X_4 = g \stackrel{ACT}{\rightarrow} X_3$ is not applied because the only subspace in which X_4 is active already contains X_3.

The partial order established between activity constraints indicates the *earliest* moment at which each constraint may be applied. Additional heuristics may *delay* the application of an activity constraint as well as that of its descendents in the hierarchy further. A good reason to do so is the generation of identical components as we will see at the end of this chapter.

Computational complexity: Ordering the set of activity constraints $\mathcal{C}^A$ is polynomial in the number of activity constraints because the identification of strongly connected compo-

nents can be done in time $max(|\mathcal{C}^A|, |\mathcal{U}|)$ where $\mathcal{U}$ are the direct dependency links and the ordering itself in $|\mathcal{C}^A| \times |\mathcal{U}|$. The combination of problem spaces, however, is exponential because each time an activity constraint is applied the number of subspaces doubles in the worst case. The algorithm for generating all subspaces is thus in $\mathcal{O}(2^{|\mathcal{C}^A|})$.

It has to be noted that this worst case is only reached for incomparable activity constraints as in example $X_1 = b \stackrel{ACT}{\rightarrow} X_3$ and $X_2 = c \stackrel{ACT}{\rightarrow} X_5$ in graph b). Here, all subspaces from both constraints have to be combined because these constraints are independent from each other with respect to DR. As soon as constraints are in series (graph a) or generate the same variable like the constraints $X_1 = b \stackrel{ACT}{\rightarrow} X_3$ and $X_5 = i \stackrel{ACT}{\rightarrow} X_3$ only a subset of the maximal number of combinations is generated. Thus, the number of resulting problem spaces in example b) is 8 and not 2^4 as expected.

Theorem 5.2 *Let $|P|$ be the number of current problem spaces. A set of n incomparable non-trivial (no AR constraints) activity constraints $\{C_i \stackrel{ACT}{\rightarrow} X_i\}$ with $X_i \neq X_j$ for $i \neq j$ and $C_i, i = 1, \ldots, n$ relevant in each space of P generates $P \times 2^n$ problem spaces.*

Proof: By theorem 5.1, the minimal solutions are found in the combinations of their subspaces. Furthermore, all subspaces have to be combined because they do not generate the same variables according to Corollary 5.2 ▲

This observation can be used to impose additional constraints on the problem forcing a choice for the variables in the activation condition of incomparable activity constraints. We will call these variables *critical variables*. If the additional constraint $X_1 = b$ is imposed, only four leaf problem spaces are left in example b). The graph-based algorithm presented here allows us to discover incomparable activity constraints as a source of combinatorial generation of problem spaces under the assumption that the activity constraints are given as input. In an interactive manner, the number of problem spaces can be reduced by forcing a user to decide on specific values for critical variables. Another way to avoid an explicit generation of all problem spaces would be a preprocessing, which checked all combinations of activation conditions of incomparable constraints for consistency and establishes nogoods similar to an ATMS. Since, in our approach, we consider one activity constraint after the other, the problem space tree is binary. The same concept could be extended to sets of activity constraints considered simultaneously.[5]

Each of the problem spaces is smaller than an overall CSP constructed from the DCSP in the sense that its number of variables is a subset of all variables and that additional constraints have been imposed. Finally, no evaluation of the problem spaces has yet taken place and the inconsistency of some problem spaces may be easy to detect. If we add for example the compatibility constraint C_1 of $\mathcal{P}_1$, the subtrees in b) rooted at $X_2 \neq c$ and $X_2 = c, X_5$ will not be generated due to the inconsistencies $X_2 \neq c, X_1 = b$ respectively $X_2 = c, X_1 \neq b$.

[5] Berthe Choueiry, Standford University, personal communication.

The generation of subspaces from a given DCSP can be compared to decomposition models for well-known search algorithms in CSPs [Freuder and Hubbe, 1995]. Here, disjunction is used to formulate a control schema for constraint satisfaction. In a similar manner, the activity constraints in a DCSP formulation can be understood as those constraint that induce a disjunction. Hence, in contrast to CSPs, the disjunctive nature of DCSPs is already present in its problem formulation.

5.6.3 General DCSP algorithm (GDCSP)

Given a dynamic constraint satisfaction problem $\mathcal{P} = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$ with continuous and discrete variables, we aim at generating systematically its problem spaces. As we have seen in the preceding sections, the entire search space of the DCSP can be represented as a tree of problem spaces generated successively. Each space defines a distinct set of active variables together with the relevant compatibility constraints. Each space is thus a standard CSP. The root node of this tree is the set of initially active variables V_I. The algorithm for generating all problem spaces containing minimal solutions, for short the minimal problem spaces, is called general DCSP algorithm (**GDCSP**) (Figure 5.7). It consists of a simple tree traversal. The algorithm first finds an applicable constraint in the set of activity constraints; i.e an activation condition for which all variables are active. For non-trivial activity constraints It then constructs successively the two subspaces by calling **with-condition** and **with-neg-condition** (Figure 5.8). The result of these calls is each time a new set of active variables and a set of relevant constraints, in short a new CSP. After both calls, **GDCSP** is applied recursively. An AR-constraint is treated by the procedure **with-condition** and simply results in a variable addition if the variable in the condition is active. The algorithm stops if the set of activity constraints is finite and all activity constraints have been considered. The function **relevant-constraints** returns all compatibility constraints that are relevant for a given set of active variables. The function **get-relevant** returns an applicable activity constraint for which the condition is relevant in the given space; i.e such that all variables in the condition are active. If $\mathcal{C}^A$ is ordered according to the DR relation, **get-relevant** returns the next activity constraint in the list of ordered activity constraints.

The set of problem spaces or CSPs implicitly defined in a DCSP corresponds to the set of leaf nodes in the search tree generated by the algorithm. The solution spaces are the solution spaces of each of the standard CSPs. The inconvenience of the proposed algorithm **GDCSP-minimal-problems** is that all problem spaces will be generated and have to be searched for solutions.

5.6.4 Example

We apply in this section the algorithm GDCSP-minimal for minimal problem space generation to the mixer example from section 2.3. Additional constraints on the vessel volume are introduced in order to extend the example. Initially, two variables are given, variable M representing an instance of a mixer and MT that of a mixing task, in other words

procedure **GDCSP-minimal-problems**$(\langle \mathcal{V}, \mathcal{C}^C \cup \mathcal{C}^A, \mathcal{D}, V_I \rangle)$
begin
1 $Vact \leftarrow V_I$
2 $Crel \leftarrow$ **relevant-constraints**$(Vact, \mathcal{C}^C)$
3 **GDCSP**$(Vact, Crel, \mathcal{C}^A)$
end

procedure **GDCSP**$(Vact, Crel, \mathcal{C}^A)$
begin
4 $ac \leftarrow$ **get-relevant**$(Vact, \mathcal{C}^A)$
5 **if** $ac \neq \emptyset$ **then**
6 remove ac from $\mathcal{C}^A$
7 $P_1 \leftarrow$ **with-condition**$(ac, Vact, Crel)$
8 **GDCSP**$(V_{P_1}, C_{P_1}, \mathcal{C}^A)$
9 $P_3 \leftarrow$ **with-negative-condition**$(ac, Vact, Crel)$
10 **GDCSP**$(V_{P_3}, C_{P_3}, \mathcal{C}^A)$
11 **else**
12 new problem space: $\langle Vact, Crel \rangle$
13 **fi**
end

Figure 5.7: *General DCSP algorithm for generating minimal problem spaces.*

function **with-condition** $(C_i \overset{ACT}{\rightarrow} X_i, Vact, Crel)$
1 **begin**
2 $Vnew \leftarrow Vact \cup \{X_i\}$
3 $Cnew \leftarrow Crel \cup \{C_i\}$
4 $Cnew \leftarrow Cnew \cup$ **relevant-constraints**$(Vnew, \mathcal{C}^C)$
5 **return** $\langle Vnew, Cnew \rangle$
end
function **with-negative-condition** $(C_i \overset{ACT}{\rightarrow} X_i, Vact, Crel)$
6 **begin**
7 $Cnew \leftarrow Crel \cup \{\overline{C_i}\}$
8 **return** $\langle Vact, Cnew \rangle$
end

Figure 5.8: *Generation of the minimal subspaces P_1 and P_3 adding a single activity constraint $C_i \overset{ACT}{\rightarrow} X_i$ to the problem space $P = \langle Vact, Crel \rangle$.*

$V_I = \{M, MT\}$. From the input description the following activity constraints have been derived among others:

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

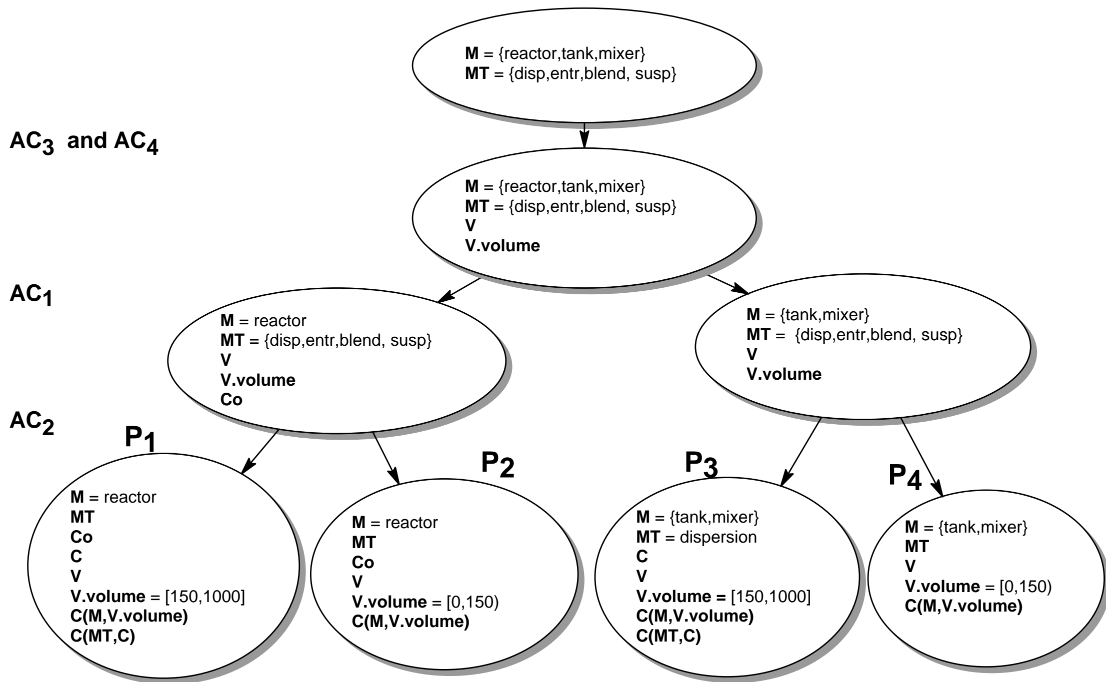


Figure 5.9: *Four problem spaces are created by two activity constraints introducing a cooler and a condenser for the mixer configuration.*

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 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: $\mathcal{V}_{P_3} = \{M = \{storage\ tank, mixer\}, V, V.volume, C\}$
P_4: $\mathcal{V}_{P_4} = \{M = \{storage\ tank, mixer\}, MT, V, V.volume\}$

Some of the compatibility constraints restrict their solution spaces:

> $C(M, V.volume)$: 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.

The activity constraints of the mixer example are generic in the sense that they have to be applied for each set of instances complying with the types required in the activity

constraints. The algorithm **GDCSP-minimal-problems** has to take into account this matching between instances and types. This is the subject of the following section.

5.6.5 Adapting GDCSP to generic constraints

The generalization to generic constraints in **GDCSP-minimal-problems** requires a matching between the active variables and the types on which generic constraints are defined. The algorithm is adapted to generic constraints in the following way:

- Activity constraints are defined on types of components. Each activity constraint has first to be matched against the set of active variables in order to find a set of variables matching the type description of the constraint (procedure **find-matches** in Figure 5.10 line 4). For each combination of active variables matching the constraint types, **GDCSP** is to be called recursively.
- A compatibility constraint is relevant with respect to a set of active variables if their types matches the types on which the constraint is defined according to Definition 5.6. **Relevant-constraints** thus returns instantiated constraints for each generic constraint that matches the active variables.
- The use of generic constraints makes it also easy to generate identical components. From the existence of a container variable, an array in this case, new instances of a given type can be generated by a single activity constraint. Activity constraints generating identical components are called *generators*. The function **generator?** in line 17 returns such an activity constraint for which there is still a set of identical components to be generated; i.e. for which the set with a maximal number of identical components has not yet been generated. The algorithm **GDCSP** is recursively called on this constraint and results in the addition of a problem space with a set of identical elements.

5.6.6 Refining the GDCSP-algorithm

The GDCSP-algorithm presented systematically generates all problem spaces from the problem formulation. Each leaf node corresponds to a static CSP that can be searched for solutions by conventional search methods. This approach resembles a "generate-and-test" technique. First, all problem spaces are generated and then they are tested for consistency. The general search strategies presented in Section 2.6.1 for discrete CSPs a priori also apply to the more abstract search space of a GDCSP. Here, a node of the search tree corresponds to a problem space (a partial CSP) and an edge between two nodes represents the choice of how to satisfy the next activity constraint. A consistency check at a node implies solving an entire CSP. Since deciding for each intermediate problem space if there is a solution or not is too costly, one possibility is to apply a *partial consistency check*. Natural candidates for such partial checks are local consistency techniques. They will only detect local inconsistencies in a problem space with the advantage that they are of

procedure GDCSP-minimal-problems$(\langle \mathcal{V}, \mathcal{C}^C \cup \mathcal{C}^A, \mathcal{D}, V_I \rangle)$
begin
1 $Vact \leftarrow V_I$
2 $Crel \leftarrow$ **relevant-constraints**$(Vact, \mathcal{C}^C)$
3 **GDCSP**$(Vact, Crel, \mathcal{C}^A)$
end

procedure GDCSP$(Vact, Crel, \mathcal{C}^A)$
begin
1 $ac \leftarrow$ **get-relevant**$(Vact, \mathcal{C}^A)$
2 remove ac from $\mathcal{C}^A$
3 **if** ac is not instantiated **then**
4 $new \leftarrow$ **find-matches**$(ac, Vact)$
5 $ac \leftarrow$ **pop**(new)
6 $\mathcal{C}^A \leftarrow new \cup \mathcal{C}^A$
7 **fi**
8 **if** $ac \neq \emptyset$ **then**
9 $P_1 \leftarrow$ **with-condition**$(ac, Vact, Crel)$
10 **if** $P_1 \neq \emptyset$ **then**
11 **GDCSP**$(V_{P_1}, C_{P_1}, \mathcal{C}^A)$
12 **fi**
13 $P_2 \leftarrow$ **with-negative-condition**$(ac, Vact, Crel)$
14 **if** $P_2 \neq \emptyset$ **then**
15 **GDCSP**$(V_{P_2}, C_{P_2}, \mathcal{C}^A)$
16 **fi**
17 $genac \leftarrow$ **generator?**(ac)
18 **if** $generator \neq \emptyset$ **then**
19 **GDCSP**$(Vact, Crel, \{genac\} \cup \mathcal{C}^A)$
20 **fi**
21 **else**
22 new solution: $\langle Vact, Crel \rangle$
23 **fi**
end

Figure 5.10: *GDCSP algorithm for generating all minimal problem spaces in design tasks.*

polynomial complexity at least in the discrete case. Similar to a backtrack algorithm, local consistency checks are applied to the intermediate problem spaces with the goal of filtering inconsistent spaces as early as possible. The earlier a problem space is removed because it contains no solution, the larger the subtree, which does not have to be explored. A simple demonstration of this idea is given when solving $\mathcal{P}_2$ already mentioned in Section 5.6.2:

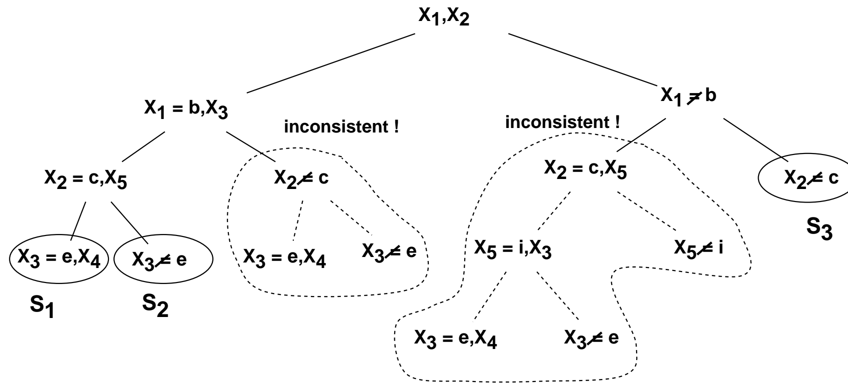


Figure 5.11: *Pruning of problem spaces due to consistency checks.*

Problem $\mathcal{P}_2 = \langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$

Variables $\mathcal{V} = \{X_1, X_2, X_3, X_4, X_5\}$ with
$D_1 = \{a, b\}$, $D_2 = \{c, d\}$, $D_3 = \{e, f\}, D_4 = \{g, h\}$, $D_5 = \{i, j\}$
$V_I = \{X_1, X_2\}$

Constraints $\mathcal{C} = \{C^1, C^2, C^3, C^4, C^5, C^6\}$ with
$C^1_{X_1,X_2} := ((a\ d)(b\ c))$
$C^2_{X_2,X_3,X_4} := ((c\ e\ h)((c\ f\ h)(c\ f\ g)(d\ e\ h)(d\ e\ g)(d\ f\ h)(d\ f\ g))$
C^3: $X_1 = b \overset{ACT}{\rightarrow} X_3$
C^4: $X_3 = e \overset{ACT}{\rightarrow} X_4$
C^5: $X_2 = c \overset{ACT}{\rightarrow} X_5$
C^6: $X_5 = i \overset{ACT}{\rightarrow} X_3$

From the dependency graph shown in Figure 5.6 b), one possible ordering of activity constraints is derived: $X_1 = b \overset{ACT}{\rightarrow} X_3$, $X_2 = c \overset{ACT}{\rightarrow} X_5$, $X_5 = i \overset{ACT}{\rightarrow} X_3$, $X_3 = e \overset{ACT}{\rightarrow} X_4$. The resulting minimal problem spaces are shown in Figure 5.11. Only the subspaces S_1, S_2, S_3 are consistent. In this case, inconsistency is already detected at the nodes $X_2 \neq c$ and $X_2 = c, X_5$ and the following subspaces do not have to be generated. Local consistency methods like arc-consistency are already sufficient to discover the inconsistencies in this example.

The new algorithm **GDCSP-minimal-spaces**, which runs local consistency at each subproblem is shown in Figure 5.12. The function **locally-consistent?** is applied in **with-condition**, **with-negative-condition** in order to remove inconsistencies (only **with-condition** is explained in the figure, the other algorithms are adapted in the same manner). It executes an algorithm ensuring local consistency with the side effect that some of the variables' labels will be refined. Those spaces that have been proven inconsistent are removed from the search tree. The result of **GDCSP-minimal-spaces** is therefore a set of

function **with-condition** $(C_i \overset{ACT}{\rightarrow} X_i, Vact, Crel)$
begin
  $Cnew \leftarrow Crel \cup \{C_i\}$
  $Vnew \leftarrow Vact \cup \{X_i\}$
  $Cnew \leftarrow Cnew \cup$ **relevant-constraints**$(Vnew, \mathcal{C}^C)$
  if **locally-consistent?**$(\langle Vnew, Cnew \rangle)$ **then**
    return $\langle Vnew, Cnew \rangle$
  else
    return $\emptyset$
  fi
end

Figure 5.12: *Function* **with-condition** *generating ensuring local consistency on the solution spaces.*

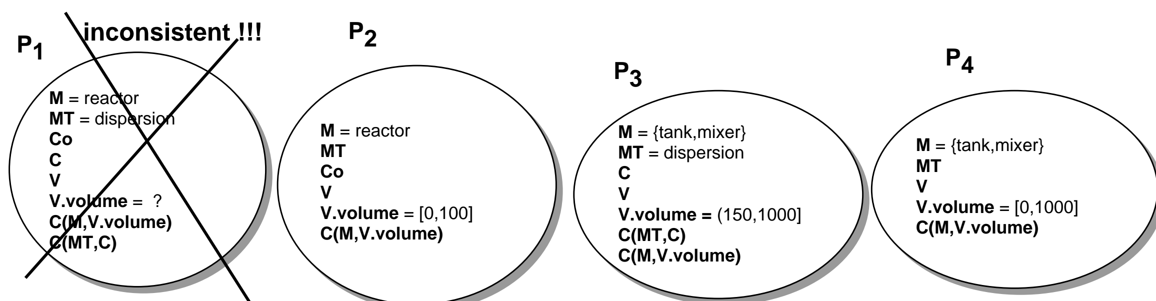


Figure 5.13: *Four problem spaces are created by two activity constraints introducing a cooler and a condenser into the mixer configuration. One of the problem spaces, P_1, has an inconsistent solution space due to constraints on the vessel volume. This inconsistency is detected when enforcing local consistency.*

locally consistent problem spaces appearing as leaf nodes in the search tree.

When applied to the mixer example, our algorithm for generating minimal solution spaces results in some pruning. Consider the problem spaces created by the algorithm **GDCSP-minimal-spaces** when adding the vessel volume to the existing problem spaces (Figure 5.13). Remember also the compatibility constraints on the vessel volume:

$C(M, V.volume)$: 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.

When local consistency is run on each of the problem spaces, P_1 becomes inconsistent due to a conflict occuring between $C(M, V.volume)$ and the activation condition of AC_2 restricting $V.volume$ to be greater than 150. This implies that there exists no solution for configuration P_1.

It is worth mentioning that a DCSP that only contains activity constraints derived from a strict hierarchical representation of objects correspond to a hierarchical domain

CSP. The advantages of hierarchical domain CSP are thus taken into account in our approach. Our algorithm ensures that, if a value corresponding to a component type in the hierarchy is found to be inconsistent, all its descendents will be inconsistent. If, for example, our condenser had two different subtypes C_1 and C_2 with additional properties, the inconsistency of problem space P_1 detected will prevent from generating further subspaces containing C_1 or C_2.

5.6.7 Completeness and soundness

We consider here the algorithm **GDCSP-minimal-spaces** given in Figures 5.10 and 5.12. We assume that the number of activity constraints treated is finite.

Theorem 5.3 GDCSP-minimal-spaces *is complete.*

Proof: All minimal solutions are contained in the generated solution spaces because:

1. **GDCSP-minimal-space** generates all minimal problem spaces according to theorem 5.1.
2. Due to the completeness of local consistency algorithms no solution is removed by the function **locally-consistent?**

▲

It is obvious that the algorithm also generates unsound solutions because a new variable is introduced on the basis of a locally sound solution space. Some of these solution spaces may prove inconsistent when using a global consistency algorithm.

5.7 Finite versus infinite number of problem spaces

In this section, we would like to examine if the number of problem spaces defined in a DCSP may be infinite. A reasonable working hypothesis is that the number of initially active variables V_I as well as the number of generic activity constraints of the DCSP is finite. The question is then if it is possible to generate an infinite number of problem spaces given a finite description of a DCSP in the form $\langle \mathcal{V}, \mathcal{C}, \mathcal{D}, V_I \rangle$.

If, at each step there is a finite set of active variables, and a finite set of activity constraints that may become relevant and since cycles between activity constraints are cut, the resulting number of problem spaces is also finite. A typical example is the configuration of a car. It is easy to write down explicitly a finite set of constraints for configuring such a car: there is an engine, a frame, at least four wheels and perhaps a spare wheel (optional). This situation changes radically if a problem specification demands a large variety in the number of identical elements. In this case, so-called generators introduce new variables. Since in a formulation with generators the exact number of elements is not given like in the car example, all problem spaces have to be generated from the minimal containing one element to the maximal with n elements. An activity constraint defined on

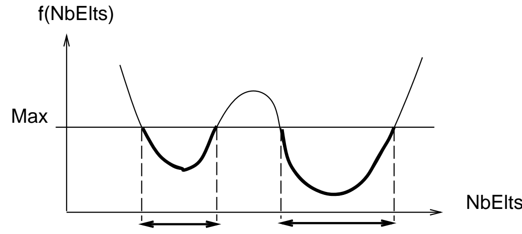


Figure 5.14: *A condition on the number of elements like $f(NbElts) \leq Max$ might generate several feasible intervals.*

an array a priori generates all problem spaces containing up to $n = \infty$ elements. Thus, our initial hypothesis of the finiteness of the set of activity constraints is no longer true. For the generation process to be finite, a halting condition is required similar to the halting condition in a simple loop construct in programming languages. Such a loop is proved to terminate if there is an invariant that does not change during the iterations. The invariant in a loop generating new elements is typically a function of the number of elements or of the condition in the loop.

Consider again the example of conceptual bridge design in which the bridge length is 150 meters and a large number of piers might be generated, which are situated at equal distance. Suppose that the variable $piers.nbPiers : integer : [2, 10]$ is a property of $piers$, $piers[i].position$ denotes the x-position of pier i and consider the following constraints:

C_4: $piers \overset{ACT}{\rightarrow} \forall_{i=1}^{piers.nbPiers} piers[i] : Pier$
C_5: $piers.nbPiers \leq 3$
C_6: $\forall_{i=1}^{piers.nbPiers-1} (piers[i+1].position - piers[i].position \geq 50)$

If we take into account only C_4 and C_5, two problem spaces are generated (Figure 5.15). If the constraints C_4 and C_6 are considered, all eight problem spaces with two to ten piers have to be generated and tested against C_6. Only two of the eight problem spaces are consistent. For a constant bridge length and an equal distribution of the piers, the maximal number of piers in this example can be deduced easily because the invariant is $piers.nbPiers * (piers[i+1].position - piers[i].position) = bridge.length$ for a given i. Deducing the upper limit is however not always possible without knowledge about other constraints on the design. For the algorithm to remain complete, all elements have to be generated in the second case because the number of elements might be restricted to several feasible intervals (Figure 5.14).

Since additional constraints should restrict the number of elements generated, they must either be formulated directly on the number of elements like in C_5 or on a property of the elements that implicitly restricts the maximal number of elements like in C_6. Such constraints on the maximal number of elements are only effective if the number of elements is bound to a problem variable and can be refined *before* generating the elements. This is not possible for constraint C_6 since the pier positions are properties of the piers, which means that they are generated once the piers exist. In case of constraint C_5 however, the generation of identical elements can be delayed by specifying a very large distance on the

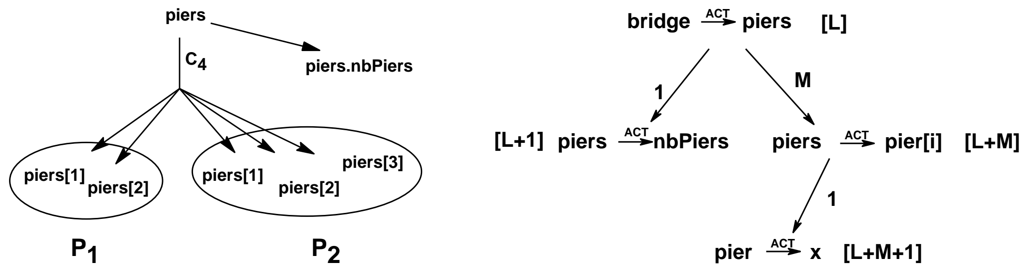


Figure 5.15: *Left: The generation of new elements is restricted by the constraint* $piers.nbPiers \leq 3$. *Three problem spaces are created of which only* P_2, P_3 *are consistent. Right: The generation of new elements is delayed in order to get a chance to reduce the domain of piers.nbPiers first. The distance M between the two activity constraints can be chosen arbitrarily large.*

link in the dependency graph between the activity constraint generating the array and the one generating the individual elements from the array (Figure 5.15). Assuming that some kind of consistency test is used, the number of elements will be refined before any element is generated.

In our actual implementation, we require designers to specify the number of identical elements as a problem variable. A generator constraint like C_4 generates the problem spaces with cardinalities determined by the label of the problem variable $piers.nbPiers$. Another possibility would be to leave the task of specifying a correct generation (one which terminates) to designers.

5.8 Searching within locally consistent solution spaces

This section is intended to give an idea of how solution spaces generated by our algorithm can be searched. In the beginning, the subject of this thesis was entirely limited to generating the locally consistent problem spaces of a DCSP. It has turned out that working with mixed problem spaces in which continuous and discrete variables coexist, is not so different from uniform CSPs. In discrete CSPs, solutions are generated using a backtracking algorithm interleaved with different degrees of consistency checks; e.g. forward checking, MAC. In continuous CSPs, most researchers use domain splitting together with local consistency in order to isolate solutions. Since we have defined propagation rules for discrete and continuous constraints, search methods described for discrete CSPs are also applicable to continuous and mixed CSPs under the condition that a kind of value enumeration is also defined for continuous variables.

5.8.1 A generic search algorithm

A generic algorithm is described in Figure 5.16. It first chooses the next variable to assign and then splits its label into two subintervals. The variable is instantiated to the mid-value of each interval. Splitting continues until a given distance w between the subsequent values enumerated is reached (w is 1 for discrete labels). **left(I)** and **right(I)** represent the left

```
function recursive-search(𝒱, 𝒞, width)
begin
  if all variables in 𝒱 are assigned then
    show solution 𝒱
  else
    𝒱 ← reorder-variables(𝒱)
    X ← next variable from 𝒱
    I_X ← next interval from L_X
    split(X, I_X, V, C, width)
end

function split(X, I_X, V, C, w)
begin
  if right(I) − left(I) > w then
    m ← some point of I_X (for example the middle)
    L_X ← m
    C ← C ∪ add-mixed-constraints(C)
    inconsistent? ← check(V, C)
    if not(inconsistent?) then
      recursive-search(V, C, w)
    fi
    reset-values(V)
    reset-mixed-constraints(C)
    I_1 ← [left(I_X), m]
    I_2 ← [m + w, right(I_X)]
    split(X, I_1, V, C, w)
    split(X, I_2, V, C, w)
  fi
end
```

Figure 5.16: *A generic search algorithm.*

respectively the right bound of an interval I. The function **check** can be any pruning algorithm from checking of fully instantiated constraints or forward checking to full local consistency. After a value has been instantiated, relevant mixed constraints are added to the current search space by **add-mixed-constraints** (see Section 5.8.2). This algorithm profits from the following ideas:

- Value enumeration for discrete variables may also refine continuous variable labels due to mixed constraints.
- Discrete value combinations occuring in a discrete-continuous constraint impose further continuous constraints on the problem spaces.
- Domain splitting can be applied to all variables. Instead of enumerating values of discrete variables, their domains can be split like continuous domains under the assumption that the discrete values are ordered and discrete domains represented

as intervals. This simply corresponds to imposing a new constraint on the problem space and to extend the problem space tree.

- Value enumeration can also be defined for continuous variables for example using the transformation functions defined for mixed constraints instead of splitting a continuous interval into two equal subintervals.

The first two points help to formulate a strategy for variable ordering. Variables occuring in discrete-continuous constraints should be enumerated first because they add continuous constraints to the problem spaces. Other discrete variables are enumerated subsequently constraining continuous variables through mixed constraints. A variable ordering is achieved according to the max degree criterion (see Section 2.6.4) first and the secondary criterion of domain size. For discrete variables, the ordering max degree + min domain (first fail principle) and for the continuous variables, the ordering max degree + max domain perform well on the examples presented in Chapter 6. This heuristic is chosen for all those variables that have not yet been reduced to a single value. In the examples presented in Chapter 6, we limit value enumeration to discrete variables and use the variable ordering mentioned. Forward checking is employed during enumeration and domain splitting into equal subintervals for continuous variables. At the first generated solution, our search algorithm backtracks to the next combination of discrete variables. The reason for this choice is that enumeration of values in the continuous labels results in very similar solutions at a distance w of the previous solutions in under-constrained systems.

When applying forward-checking to continuous variables and constraints, the corresponding ternary constraints become binary, and the intersections between two ternary constraints differing in their third variable can be computed due to the binary refine operator given in Figure 3.8. Consider two ternary constraints C_{XYZ_i} and C_{XYZ_j} with $i \neq j$ and the assignment $\{Z_i = z_{i_k}, Z_j = z_{j_k}\}$. The binary system of those constraints forms a total constraint $C^t_{XY} = \{C_{XY}(X, Y, z_{i_k}), C_{XY}(X, Y, z_{j_k})\}$ that is considered in a single propagation step by the binary refine operator. Therefore, instantiating continuous variables leads to a tightened binary constraint network and the binary refine operator takes into account additional constraint intersections.

5.8.2 Discrete-continuous constraints

Constraints in which discrete tuples are mixed with continuous constraints as for example the constraint on the vessel form in Figure 3.24 are propagated during search. The strategy is to enumerate the values of discrete variables in such constraints first in order to define exactly those subspaces in which the continuous part of the constraint is valid.

Consider the three remaining subspaces $P_i, i = 2, 3, 4$ of the mixer example illustrated in Figure 5.13. The following activity constraints define a new subspace for the elliptical vessel type:

AC_5: $V = Elliptical \overset{ACT}{\rightarrow} sradius : real : [0, 1000]$

AC_6: $V = Elliptical \overset{ACT}{\rightarrow} bottomarea : real : [0, 1000]$

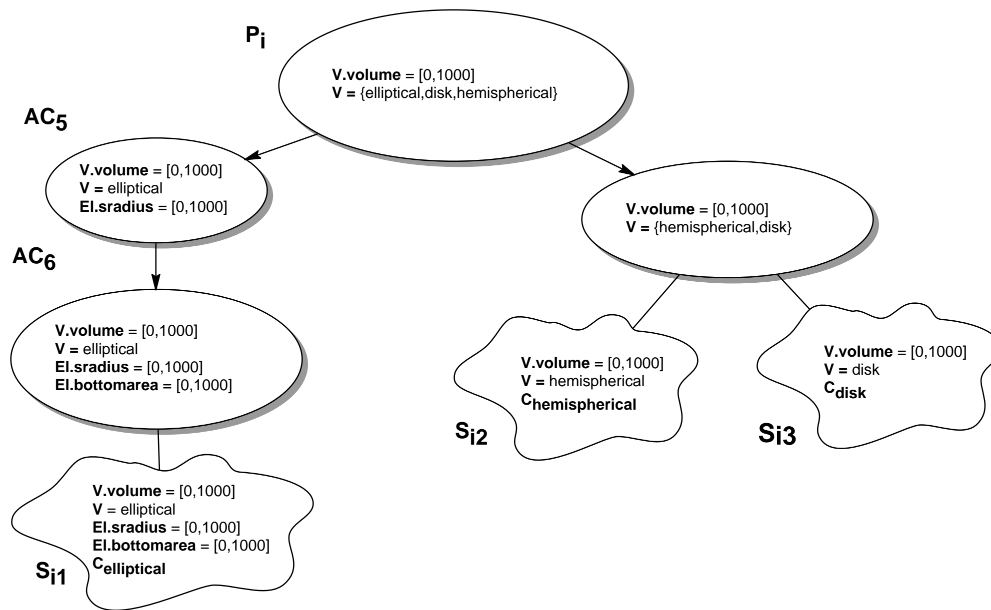


Figure 5.17: *The problem spaces created from $P_i, i = 2, 3, 4$ adding the activity constraints AC_5 and AC_6 . Subsequent search finds three different search spaces according to the values of the vessel type. The search spaces are S_{i1},S_{i2} and S_{i3}.*

The subspaces in which the continuous constraints of the mixed constraint $C(V, V.volume, V.diameter)$ of Figure 3.24 are applied are shown in Figure 5.17. $C_{elliptical}$, $C_{hemispherical}$ and $C_{cylindrical}$ are the continuous constraints determining the vessel volume in function of the vessel type.

Another example of continuous-discrete constraints are piecewise defined functions. In general, a piecewise defined function consists of a set of functions each of which is defined over a given domain of definition. Such a function is usually written as conditional statement of the form:

$$F(X) = \left\{ \begin{array}{ll} F_1(X) & if\ X \in [min_1, max_1] \\ \ldots & \\ F_n(X) & if\ X \in [min_n, max_n] \end{array} \right\}$$

with $min_i, max_i \in \mathbb{R}$. Mixed constraints can be used to encode this kind of knowledge as follows

$$\begin{aligned} \mathbf{C}(\mathbf{X}, \mathbf{Y}) \quad := \quad & \{([min_1, max_1]\, Y - F_1(X) = 0) \\ & \ldots \\ & ([min_n, max_n]\, Y - F_n(X) = 0)\} \end{aligned}$$

5.9 Summary

This chapter generalizes dynamic constraint satisfaction to problems involving continuous variables. Design and configuration problems typically require decisions about value as-

signments to be taken in a given context of already assigned variables, which might be of discrete and continuous type. A dynamic constraint satisfaction framework allows for introducing new variables based on partial assignments or, more generally, restrictions of value combinations. A new method is presented, which generates all problem spaces of a dynamic CSP. Each problem spaces defines its own variable and constraint set. A basic assumption is that the complement of activation conditions can be formulated and that the activation condition is formulated by a single constraint. Our generation method has several advantages:

- Continuous and discrete constraints can be treated in a dynamic constraint satisfaction framework, in which also continuous activation conditions are allowed. In general, an activation condition can be a constraint and is not restricted to a partial assignment like the original algorithm by [Mittal and Falkenhainer, 1990].
- The dependency graph defined by the activity constraints of a DCSP is analyzed in order to determine the order in which they have to be applied. This analysis also identifies
    1. cycles between activity constraints so that they can be cut, which implies that looping is avoided like in the ATMS-algorithm of Mittal and Falkenhainer,
    2. incomparable activity constraints that are responsible for a combinatorial number of problem spaces.
- The search tree generated can be seen as a higher level CSP, in which each node defines a partial problem space or CSP. Hence, search methods defined for static constraint satisfaction problems can be applied to each node. Furthermore, solutions within one problem space are similar in that they are defined on the same variables and restricted by the same constraints.
- Due to a generic formulation of constraints, the number of variables must not be explicitly known at the time of problem formulation. However, for variables created by generators a maximal limit must be given to prevent a unlimited variable generation. This last point is not a severe restriction, because in practical design problems the number of components is always limited.
- The detection of incomparable activity constraints can be signaled to the user who then can minimize the maximal number of problem spaces generated by fixing some or all variables in the conditions of these constraints.
- Local consistency is not only of use in order to prune inconsistent problem spaces as early as possible but also to exhibit a single consistent solution of a problem space. Techniques like forward checking or MAC employ local consistency extensively to enhance search especially in the continuous domain.

Finally, an interface for representing design tasks in the dynamic CSP framework is proposed.

Chapter 6

Results

".... it is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting point and the conclusion, one may produce a startling, though possibly a meretricious, effect."

Sherlock Holmes in *The Adventure of the Dancing Men*, The Strand Magazine (1903-1904)

6.1 Introduction

In this chapter, three examples are presented from the domain of configuration and design and the results of applying systematic enumeration of solution spaces are discussed. The application examples are:

1. Two simple examples of binary respectively ternary constraint systems.
2. Configuration of trains.
3. Configuration of industrial mixers.
4. Preliminary design of bridges.

The first examples compare the pruning power of different consistency algorithms on numeric CSPs. The train example is a mixed CSP and shows how local consistency applied during search finds solutions. The example on mixer configuration summarizes the results of GDCSP and search in locally consistent spaces and is used throughout the thesis. Preliminary design of bridges emphasizes the generation of identical elements (the bridge piers).

The current version of GDCSP as well as the propagate algorithm are implemented in Allegro Common Lisp. The propagation rule **simple-propagate** for continuous constraints is implemented in Maple. The communication of results between Maple and Lisp is based on interprocess communication. It is clear that a more efficient version should include the algorithm **simple-propagate** directly into the application code. The programs are run on a two-processor Sun Sparc Ultra Enterprise II.

6.2 A binary constraint set

In order to show how the local solution spaces compare to solution spaces from which a solution can be instantiated backtrack-free, we implemented a small example consisting of pure binary constraints given in Haroud's thesis [Sam-Haroud, 1995]. The example consists of six total constraints on pairs of variables in X, Y, Z, T:

$$\begin{array}{ll} X + Y \leq 8 & \\ Y - X \leq 0 & Y + (Z-2)^2 \leq 3.75 \\ Y - \frac{1}{(X-4)^2} \geq 0 & Y - (Z-2.3)^2 \geq 0.25 \end{array}$$

$$\begin{array}{ll} Z - e^{(X-4)} \leq 1.5 & Y + 0.2T \geq 1 \\ Z - e^{(4-X)} \leq 2 & Y + 0.25T \leq 5.5 \\ Z - 0.5X \geq -2 & Y - 1.5T \leq 1 \\ Z + 1.25X \geq 1.25 & Y - 0.6T \geq -3 \\ Z - 0.0075X \geq 0 & Y - 0.334T \leq 2 \end{array}$$

$$\begin{array}{ll} T + (Z-2)^2 \leq 3.75 & T - e^{(X-4)} \leq 1.5 \\ T - (Z-2.3)^2 \geq 0.25 & T - e^{(4-X)} \leq 2 \end{array}$$

The total constraints are represented in Figure 6.1. In this example, path-consistency allows for the computation of globally consistent solutions because certain partial convexity properties are satisfied (refer to [Sam-Haroud, 1995], page 144-146). The labels computed by path-consistency are approximations of two-dimensional shapes represented by quadtrees . These approximations have been computed with a certain degree of precision, which has to be taken into consideration when comparing the results. Furthermore, we show here the projections of these quadtrees onto each axis. The result of local consistency given the initial labels $[0, 10]$ is

```
X : [0.4893628958,3.407938147] [4.592061853,7.510637104]
Y : [0.4893628958,2.852763964]
Z : [0.8356560572,2.553185521]
T : [0.25,2.553185521]
```

compared to the path-consistent solution spaces (projected onto the axes):

```
X : [0.75,3.4375] [4.5625,7.4375] Y : [0.53125,2.875]
Z : [0.820312,2.57812]
T : [0.234375,2.57812]
```

Only the lower bounds for X and Y have been refined more tightly, all other deviations results from the discretized representation of the constraints in form of 2^k-trees. In this example, seven levels of binary decomposition have been used resulting approximatively in a precision of 0.08. It must also be noted that, in the algorithm achieving path- and higher degrees of consistency, the level of precision depends on the largest initial interval that must be taken into consideration. In our algorithm achieving local consistency, the choice of initial labels is less critical. If some of the initial labels are not known, one can simply choose a sufficiently large domain; even $[-\infty, +\infty]$ would be acceptable.

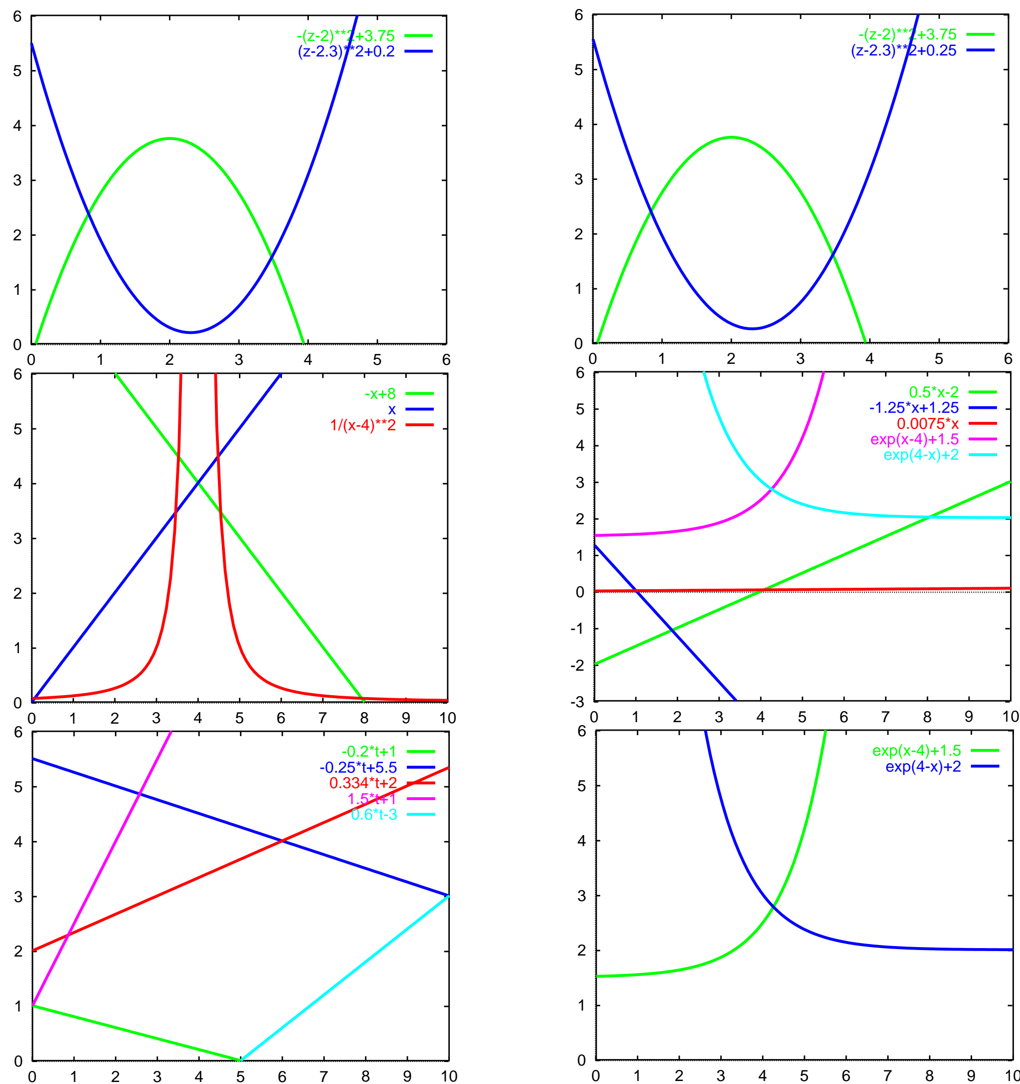


Figure 6.1: *The binary constraint example.*

6.3 Comparison between Davis' and Falting's propagation rules

We applied the Davis propagation rule to the following example:

	Constraint set 1
C_1	$-4 * X_5 - 1 * X_6 + 6 * X_9 + 6 > 0$
C_2	$4 * X_4 - 9 * X_5 - 5 * X_8 - 5 > 0$
C_3	$-9 * X_2 + 7 * X_3 + 4 * X_5 + 4 > 0$
C_4	$-1 * X_6 + 10 * X_9 - 4 * X_{10} - 4 > 0$
C_5	$8 * X_4 + 3 * X_7 + 4 * X_8 + 4 > 0$
C_6	$-6 * X_3 - 2 * X_5 + 6 * X_6 + 6 > 0$
C_7	$6 * X_2 - 1 * X_4 + 7 * X_5 + 7 > 0$
C_8	$8 * X_1 - 5 * X_2 - 8 * X_3 - 8 > 0$

X_i	**Local C.**	**Davis**	**Simplex**
X_1	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
X_2	$[-10, 7.8339]$	$[-10, 10]$	$[-10, 7.8339]$
X_3	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
X_4	$[-6.8826, 10]$	$[-9.25, 10]$	$[-5.8122, 10]$
X_5	$[-6.5396, 9.4445]$	$[-10, 9.4445]$	$[-4.8745, 9.4445]$
X_6	$[-10, 10]$	$[-10, 10]$	$[-9.7322, 10]$
X_7	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
X_8	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$
X_9	$[-4.6, 10]$	$[-4.6, 10]$	$[-3.29, 10]$
X_{10}	$[-10, 10]$	$[-10, 10]$	$[-10, 10]$

Table 6.1: *Solutions for constraint set 1 using various consistency techniques.*

A comparison of the results of Davis' rule and the local consistency rule (Table 6.1) reveals that more pruning of labels is obtained by taking into account intersections between constraints defined on pairs of variables. This can be clearly seen in the variable X_5, which is involved in the constraints C_1,C_2,C_3,C_6 and C_7 forming many variable pairs with other variables. As reference, we also calculated the globally optimal solution for constraint set 1 using the simplex method. As the labels of X_2, X_4 and X_5 show, local consistency is always better than an individual propagation of constraints (e.g. Davis) but does not achieve global consistency.

6.4 Configuration of trains

This is a small example of a train configuration, in which, given the maximum acceleration, maximum speed, and the vehicle payload, the required power as well as the number, and type of engines have to be deduced. This example consists of mixed constraints, inequalities, nonlinear equalities, and involves discrete variables as well. In a first step, local consistency is computed from the initial domains. Second, a search is conducted within the locally consistent solution spaces using forward checking. The variables of this example are given in Table 6.2. For the variables VM and RP, initial domains were not given. Therefore, they were chosen large enough so as to cover the entire feasible region of the individual constraints. The constraints are:

Variable	Description	Domain
$MaxAcc$	Maximum Acceleration	$1.1m/s^2$
$MaxSpeed$	Maximum Speed	$30m/s$
VP	Vehicle Payload	$16000kg$
$VAxles$	Vehicle Axles	$[2..8]$
E	Engine Type	$\{E1, E2, E3\}$
EP	Engine Power	$[100, 400]$
$EPMass$	Engine Mass	$[3000, 5000]$
VM	Vehicle Mass	$[80000, 500000]$
RP	Required Power	$[0, 100000]$
$NoEngine$	Number of engines	$[2..16]$

Table 6.2: *Variables and their domains used in the train configuration example.*

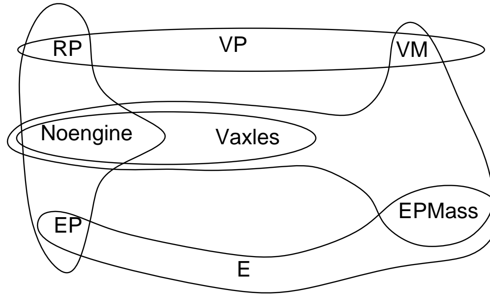


Figure 6.2: *The constraints of the train configuration example form cycles in the corresponding hypergraph.*

$$VM = 80000kg + NoEngine * EPMass + VAxles * 1000kg$$
$$RP = (VM * 1,06 + VP) * MaxAcc * 0,000434 * MaxSpeed + (VM + VP) * 0,0014715$$
$$NoEngine = \lceil RP/EP \rceil$$
$$2 * VAxles > NoEngine$$
$$NoEngine > VAxles$$
$$VAxles \neq 5$$
$$VAxles \neq 7$$
$$C(E, EP, EPMass) :== ((E1\ 100\ 5000)(E2\ 400\ 3000)(E3\ 300\ 3300))$$

The constraints form a hypergraph with cycles as shown in Figure 6.2. Local consistency is thus not expected to compute globally consistent solution spaces. The constraint $NoEngine = \lceil RP/EP \rceil$ is replaced by the two constraints $NoEngine \geq RP/EP$ and $NoEngine < RP/EP + 1$. The constraints are ternarized and local consistency on the ternarized system establishes the following solution spaces:

```
ENGINE : {E2,E3}
NOENGINE : {5,6,7,8}
VP : 16000
VAXLES : {4,6}
MAXACC : 1.1
EP : {300,400}
RP : [1901.32518,2124.472968]
MAXSPEED : 30
EPMASS : {3000,3300}
VM : [99000,112400]
```

The result is a list of variables with their locally consistent labels. A label is either a list indicated by brackets for discrete and integer variables or a set of intervals for a real variable. When comparing the results with the initial labels given in Table 6.2, the attentive reader can see,that the initial labels of RP, VM, $NoEngine$, and $Vaxles$ have been reduced considerably. Furthermore, one of the engine types, $E1$, does not supply enough power in order to be a candidate for a solution.

In a second step, the resulting locally consistent solution spaces are searched for single solutions. The search strategy chosen in this example is to enumerate first the discrete (symbolic and integer) variables according to the first-fail principle and to propagate value assignments through the continuous parts of the constraint network using local consistency. This heuristic proves interesting here because after the instantiation of the discrete variables E, $NoEngine$, and $VAxles$ the constraint network is tree-structured and local consistency filters globally consistent solutions. The four solutions found are:

```
E = E2,NOENGINE = 5,VAXLES = 4,RP = 1901.32518,VM = 99000
E = E3,NOENGINE = 7,VAXLES = 4,RP = 2036.21302,VM = 107100
E = E3,NOENGINE = 7,VAXLES = 6,RP = 2069.51866,VM = 109100
E = E3,NOENGINE = 8,VAXLES = 6,RP = 2124.47297,VM = 112400
```

Some of the combinations of were rejected because the constraint $Vaxles < NoEngine < 2 * Vaxles$ was violated. On the other hand, the constraint $RP/EP \leq NoEngine < RP/EP + 1$ associated small values of $NoEngine$ to the engine type $E2$ and larger values of $NoEngine$ to $E3$.

6.5 Configuration of industrial mixers

This example has been adapted from F5 described in [van Velzen, 1993]. The example is shortly described in section 2.3 together with a figure of the part-of hierarchy. A list of variables occuring in this problem is presented in the Appendix A.1. A complete description of the constraints is also found in the appendix. The activity constraints describe conditions under which a cooler and a condenser must be added to the mixer configuration. The various compatibility constraints listed in the appendix are relevant in those problem spaces in which the constraint's variables are active. In Figure 6.3, the hypergraph of the problem

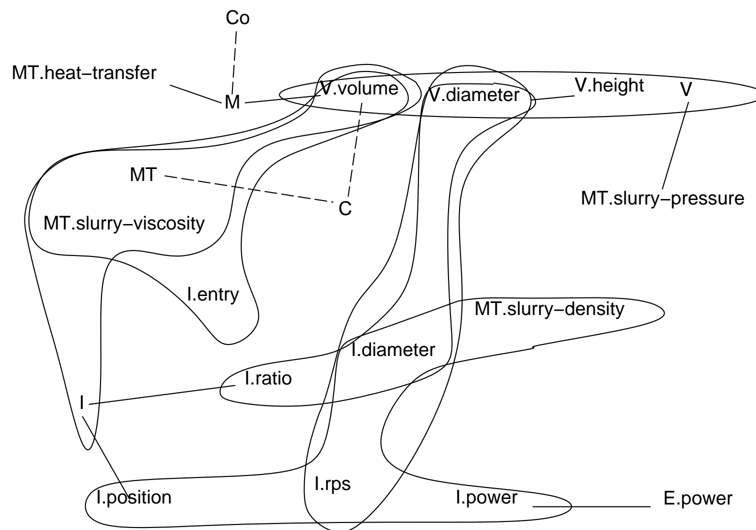


Figure 6.3: *The constraints of the mixer configuration example form cycles in the corresponding hypergraph.*

restricted to a non-elliptical vessel form is shown, with those constraints dotted that are not relevant to all problem spaces.

Throughout this thesis the examples has been reused in order to illustrate the generation of problem spaces. The spaces containing a cooler and a condenser simultaneously are detected to be inconsistent. The six resulting locally consistent spaces can be can be found in the Appendix in Table A.1.3. There are all combinations containing either a cooler or a condenser or no option at all. Furthermore, each of the mixer configurations has an elliptical, cylindrical or spherical vessel.

We choose one of the spaces with $M = \{mixer, storage\ tank\}$ and $V.volume = [150, 1000]$ in order to find single solutions using forward checking with dynamic variable ordering. The search heuristic used was the one described in Section 5.8. Furthermore, we enumerated exhaustively the integer and discrete domains but looked for only one solution for the continuous subnetwork. In this search space there are exactly 12 solutions according to our instantiation heuristics because four discrete variables with a domain size of two have to be searched and because the value combination $\{V = hemispherical, MT.slurry\ pressure = low\}$ is not allowed. We concentrate on those solutions with $M = storage\ tank$ and $MT.slurry\ viscosity = low$. The resulting individual solutions are listed in Table 6.3.

It can be seen that the first two solutions are identical except from the choice for *MT.slurry pressure*. These two solutions are *interchangeable* in the context of the locally consistent solution space given. The last solution has different values for the vessel parameters because the vessel volume is computed by a different constraint.

Variable	Solution 1	Solution 2	Solution 3
M	storage tank	storage tank	storage tank
M.agitator	agitator	agitator	agitator
M.vessel	cylindrical	cylindrical	hemispherical
MT	dispersion	dispersion	dispersion
MT.heat-transfer	false	false	false
MT.slurry-pressure	high	low	high
MT.slurry-viscosity	low	low	low
MT.slurry-density	1.5	1.5	1.5
A.impellers	radial-turbine	radial-turbine	radial-turbine
A.shaft	shaft	shaft	shaft
E.power	4375.025	4375.025	4375.025
I.diameter	1.9615	1.9615	1.9615
I.entry	top	top	top
I.pos	5	5	5
I.power	1875.025	1875.025	1875.025
I.ratio	0.329102	0.329102	0.329102
I.rps	2.0496	2.0496	2.0496
V.diameter	5.9602	5.9602	5.9602
V.height	11.9204	11.9204	11.9204
V.opt2	condenser	condenser	condenser
V.volume	332.586	332.586	304.8705

Table 6.3: *Three solutions of the first locally consistent solution space of Figure 7.4.*

The time requirements for this search with forward checking and with simple backtracking are:

```
Forward checking:
cpu time (non-gc) 1,840 msec user,650 msec system
cpu time (gc) 50 msec user,0 msec system
cpu time (total) 1,890 msec user,650 msec system
real time 15,249 msec
```

```
Backtracking:
cpu time (non-gc) 383,280 msec (00:06:23.280) user,19,180 msec
system
cpu time (gc) 17,850 msec user,50 msec system
cpu time (total) 401,130 msec (00:06:41.130) user,19,230 msec
system
real time 552,029 msec (00:09:12.029)
```

6.6 Preliminary design of bridges

In this example, the configurations shown in Figure 1.1 are generated. Again, the description of the problem can be found in the appendix A. The example was initialized with:

```
BRIDGE.ABUTMENT1 :  8
BRIDGE.ABUTMENT2 :  7
BRIDGE.ABUTMENTS : 15
GAP.SOIL : GOOD
GAP.CENTERLINE : 77.5
GAP.DEPTH : 30
GAP.GEOMETRY : MOSTLY-SYMMETRICAL
GAP.RATIO : 0.2
GAP.WIDTH : 150
OBSTACLE : ROAD
ROAD.WIDTH : 11.5
ROAD.ALIGNMENT : SLIGHTLY-CURVED
OBSTACLE.RSIZE : 0.1
OBSTACLE.SIZE : 15
OBSTACLE.X1 :  70
OBSTACLE.X2 :  85
BRIDGE.PROFILE : (ABOVE-DECK CONSTANT-DEPTH)
```

The bridge profiles were restricted to above-deck and constant-depth in order to show the two alternatives as mentioned in the introduction to this thesis. The first two alternatives consist of a beam bridge with 3 respectively 5 spans, which corresponds to 2 respectively 4 piers.

```
BRIDGE : BEAM BRIDGE.PROFILE : CONST-DEPTH
BRIDGE.NBSPANS : 3
BRIDGE.BLENGTH : 135
SPANS[1].LLENGTH : [42.75,47.25]
SPANS[2].LLENGTH : [42.75,47.25]
SPANS[3].LLENGTH : [42.75,47.25]
SPANTYPE.LLENGTH : [1,50]
```

```
BRIDGE : BEAM BRIDGE.PROFILE : CONST-DEPTH
BRIDGE.NBSPANS : 5
BRIDGE.BLENGTH : 135
SPANS[1].LLENGTH : [25.65,28.35]
SPANS[2].LLENGTH : [25.65,28.35]
SPANS[3].LLENGTH : [25.65,28.35]
SPANS[4].LLENGTH : [25.65,28.35]
SPANS[5].LLENGTH : [25.65,28.35]
SPANTYPE.LLENGTH : [1,30]
```

The third alternative is an arch bridge which only has one span.

```
BRIDGE : BEAM BRIDGE.PROFILE : ABOVE-DECK
BRIDGE.NBSPANS : 1
BRIDGE.BLENGTH : 135
SPANS[1].LLENGTH : 135
SPANTYPE.LLENGTH : [1,30]
```

Although this example is quite simple, it shows well the generation of spans and also how the according generic compatibility constraints are adapted to the varying number of spans.

Chapter 7

Conclusions

“Every problem becomes very childish when once it is explained to you.”

Sherlock Holmes in *The Adventure of the Dancing Men*, The Strand Magazine (1903-1904)

7.1 Scope of this research

As we have laid out in the introduction, the stage of conceptual design was deliberately chosen as starting point for an identification of solutions, since, at that point, parts of the structure of the artifact are already decided on and constraints have been specified. A DCSP has been identified as computational model for such a task because it provides a declarative formulation of combinatorial as well as dynamic problems, in which new variable and constraint instances are derived during resolution.

This dissertation defends the idea that approximating the solution sets of a DCSP by local consistency techniques is of use in a support system for designers because

1. Knowledge of several solution spaces makes potential choices explicit and gives a more informed basis for decisions. Thus, key decisions have not to be taken early but can be delayed.

2. Once, a solution space has been committed to, the corresponding CSP can be derived and serves as input to search algorithms. During search, local consistency techniques are again useful to reduce the search space further.

Under these assumptions, algorithms for local consistency providing tight labellings, especially in continuous domains, have to be devised and integrated with existing discrete local consistency algorithms in order to be able to treat mixed (continuous-discrete) constraints. Furthermore, the DCSP formalism has to be extended in order to take into account that new variables might be introduced on the basis of a partial value assignment to continuous variables. All these requirements imply the need for a method that approximates solution sets in mixed DCSPs.

7.2 Summary of major results

The main contributions can be summarized as follows:

- We generalize the original local consistency algorithm for continuous binary constraints given by [Faltings, 1994] to ternary constraints. The new algorithm uses the same local properties of solution spaces as the binary one: stationary points of ternary constraints and intersections between triplets and pairs of constraints. A constructive proof of correctness is given based on topological considerations, which also allows us to identify a restriction: solution regions containing holes require additional information to be treated correctly. Moreover, only intersections between ternary constraints in the same three-dimensional space are considered. Otherwise, the proof would have to be extended to constraint regions k-ary space with in $k > 3$ and this extension of the proof is not trivial.

- The integration of local consistency methods for discrete and continuous constraints into a fix-point algorithm is achieved by specifying refine operators for both constraint types. Additionally, variables shared by constraints of different type may have their labels approximated by a transformation function because not all refine operators compute in the same variable domain. Since refine operators are using approximate variable domains, some locally unsound values may remain in the labellings after the application of an operator. In the conversion of discrete domains to continuous domains, for example, discrete values representing open intervals are approximated by closed intervals, because our continuous refine operator cannot handle strict inequalities.

- We extend the original DCSP framework of [Mittal and Falkenhainer, 1990] to introduce new variables based on value combinations formulated as constraints. In this formalism, an activation condition can be a discrete or a continuous constraint under the assumption that the complement of such a condition exists. Additionally, constraints are generic, i.e. formulated on variable types instead of instances, and can thus be applied several times.

  A solution method is presented that generates all problem spaces of a dynamic CSP. Each problem space defines its own variable and constraint set. Since we cannot enumerate solutions (continuous variables !) and proving that a solution exists for each intermediate problem space is too costly, local consistency algorithms are used to prune inconsistent problem spaces.

  An analysis of the dependency graph generated from the activity constraints allows us to identify cycles in the graph and eliminate them, to find an order in which the activity constraints have to be applied and finally, to identify incomparable activity constraints that are responsible for an exponential number of problem spaces. If all variables in the conditions of such incomparable constraints are fixed, the number of problem spaces generated can be reduced.

7.3 Applicability and limitations

Initial problem representation. Our method for problem space generation relies on an input in form of constraints and variables, each with its initial domain. Such domains are given naturally in most physical systems in form of some lower and upper bounds of feasibility. Otherwise, a sufficiently large range of potential values can be taken as starting point, since the efficiency of our local consistency method does not depend on the size of the initial domains but rather on the identification of stationary points of constraint regions.

Soundness of local consistency techniques and their effect in the generation algorithm. Since shared variable labels are approximated when enforcing local consistency over mixed constraints, locally unsound values might be included in the labels. If a solution is situated on the boundary defined by the feasible region of continuous activation condition, it is included in both subspaces. This is due to the restriction of our continuous consistency algorithm, which only treats closed regions. The complement of a closed region being an open region we cannot represent this open region exactly. For the same reason, some conflicts situated exactly on the boundary of a continuous activation condition may remain undetected.

Termination of problem space generation. In order to guarantee termination of problem space generation, we require the definition of the maximal number of identical elements to be generated by a generic activity constraint. This seems reasonable in the light of an initial problem representation in which each variable has to be given an initial domain.

Generality and reliability of local consistency algorithms. In this thesis, a generic algorithm achieving local consistency is proposed, which does not depend on specific constraint types and which treats continuous and discrete constraints by means of specialized refine operators. The refine operator for continuous constraints relies on a precomputation step, which determines local properties like intersections and stationary points of ternary constraint regions. Under the assumption that these properties can be determined, the locally consistent labellings can be computed.

7.4 Open research issues

Most difficulties in finding solutions to mixed dynamic constraint systems are encountered when enforcing local consistency on the continuous subsystem. Some of the future issues therefore relate to the subject of consistency techniques for continuous constraint systems.

7.4.1 Representation of continuous constraints

Neither a discretized nor the analytical representation of constraints is satisfying. The fundamental problem is that continuous constraints can define arbitrarily complex shapes

in continuous space. A simple shift from from binary to ternary constraints results in an impressive increase of topological complexity: different types of connectivity are possible from simply to multiply connected regions. It is not realistic to think of a further investigation into higher arity constraints resulting in higher dimensional spaces before having solved the representational problem. It is rather a question if this spatial complexity can be tamed by introducing an approximation of constraint regions. The appearance of holes in three dimensions has complicated in an unforeseen manner our algorithm. Since a hole is by definition embedded in a feasible region and we are only interested in the projection of this region onto one axis, holes should be neglected by a refine operator. To this purpose, stationary points lying on the boundary of holes should be recognized as such. An approximation of feasible regions should thus remove holes from consideration.

A further research direction could aim at investigating the links between different theories in topology and the consistency condition. Our work, for example, has strong similarities with the Morse theory in topology. Another branch in topology concerning simplical surfaces could be another means of approximating the surface of constraint regions.

7.4.2 Treatment of equalities

Our algorithm for local consistency has been designed for inequalities. Equalities are represented as a pair of inequalities in the prototype implementation. An improved implementation could integrate the treatment of equalities directly into the algorithm. It can be observed that many engineering systems are described by constraints that contain few but relatively large cycles and many equalities without cycles representing derivations of intermediate variables. An algorithm taking also advantage of value propagation during propagation seems reasonable in such cases.

7.4.3 Search in continuous and mixed CSPs

The search algorithm proposed in Section 5.8 uses domain splitting for all variable types in a very straightforward manner. Integer and discrete variable domains are entirely scanned whereas the continuous domain is scanned at regular intervals of given width. After each instantiation forward checking is used to propagate the new value. Further search techniques for discrete CSPs could be adapted easily to mixed CSPs. Little effort has been invested in search in continuous domains until now. It is for example not clear, if the widely acknowledged first-fail principle for discrete domains is also useful in continuous domains. Little research has been conducted with respect to variable ordering and similar heuristics.

7.4.4 Extension of the DCSP model

In this thesis, the DCSP model is restricted to activity constraints the condition of which is a single discrete or continuous constraint and a single new variable is activated, in other words activity constraints are restricted to clauses. This could be extended to include activity constraints with disjunctions in the head and conjunctions in the body (conditional

part). Recently, non-monotonic propositional logic has been applied to solve configuration tasks [Soininen and Niemelä, 1998]. The links that exist between non-monotonic logic and dynamic constraint satisfaction could help to extend the existing DCSP model.

7.4.5 Reverse engineering of DCSPs

As we have seen in Chapter 5, a set of constraint problems is generated from a given CSP. An interesting viewpoint is now to ask if one can learn from the collection of CSPs in order to reformulate the DCSPs. One goal could be the design of better configurable products[1]. Another investigation line might try to discover a DCSP structure in a given CSP thus dividing up the problem in subproblems that could be treated in parallel.

7.4.6 Representation of results

Our prototype implementation presents the locally consistent solution spaces issued from the enumeration of problem spaces in the form of a list of variable - interval pairs. In the same manner, individual solutions are presented. From the viewpoint of human-computer interaction, a potential user would be drowned with numerical information. Single solutions can be used directly to represent a finished design product within a CAD-environment. Such a single solution could be a representative for the entire set of locally consistent solutions from which it was computed. Moreover, additional information for important variables like the range of locally consistent values could be highlighted within the CAD drawing. Examples of important variables in a conceptual bridge design are the span distribution, construction method, the type of section and the bridge type (Section 1.1).

7.5 Final conclusion

During this thesis work, some surprising facts have emerged.

One aspect of this thesis has been the investigation into dynamic CSPs. It turned out that a careful designed algorithm is able to guide designers in their decision making before even starting to generate solution spaces. The fact that independent conditions activating different variables are a source of combinatorial explosion is obvious, once it has been recognized. This combinatorial effect can be avoided by forcing decisions on the variables in the identified activation conditions.

The second discovery (a bit less surprising perhaps) has been the increase in topological complexity when moving from binary to ternary continuous constraint systems. Although we have not completely achieved the aim of enforcing local consistency in continuous ternary systems and some questions remain open, I think that an investigation into the topological properties of the feasible space defined by continuous constraints has enabled some insight into the problem and might generate fruitful ideas in the future.

[1] Mihaela Sabin, personal communication

Appendix A

Examples from configuration and design

We list here the full specifications of three examples treated in this thesis: the configuration of an industrial mixer, preliminary design of bridges, and configuration of trains. All the examples share similar characteristics: they involve constraints on continuous as well as discrete variables. Furthermore, tasks in configuration and design are preferably represented in a modular way using Object Oriented features, like type and part-of hierarchies. In order to formulate these configuration and design examples, we use the following design knowledge

1. part-of structure
2. types and subtypes of components
3. compatibility constraints defined on types of components (applicable to all components of the given type)
4. activation constraints defined on types of components

Notational convention: variables representing component types are abbreviated by their initial. Properties of a component type are abbreviated by <component-type-initial . name>.

A.1 Configuration of an industrial mixer

This example has been adapted from F5 described in [van Velzen, 1993]. The example is shortly described in section 2.3 together with a chart of the part-of hierarchy. We present here the OO-description of the mixer.

A.1.1 Variables

```
Type MTask:
  domain : discrete : {suspension dispersion entrainment blending}
  slurry-viscosity : discrete : {low high}
  slurry-pressure : discrete : {low high}
  slurry-density : real : [1,2000]
  heat-transfer : discrete : {true false}

Type Mixer:
  domain : discrete : {reactor mixer storage-tank}
  vessel : Vessel
  agitator : Agitator


Type Vessel:
  domain : discrete : {cylindrical elliptical hemispherical}
  volume : real : [0.01,1000]
  diameter : real : [0.01,1000]
  height : real : [0,1000]
  opt1 : Cooler [optional]
  opt2 : Condenser [optional]

Type Agitator:
  engine : Engine
  impellers : Impeller
  shaft : Shaft

Type Impeller
  domain : discrete : {axial-turbine helical-ribbon propeller silverson-highshare
                       dented-disk radial-turbine scaba anchor-stirrer}
  entry : discrete : {top side off-center}
  pos : real : [0,5]
  diameter : real : [0.1,1000]
  ratio : real : [0,1]
  rps : real : [1,100]
  power :real : [0,5000]

Type Shaft

Type Engine:
  power : real : [0,5000]

Type Cooler

Type Condenser

Type Elliptical of Vessel
  sradius : real [0.01,1000]
```

```
bottomArea : real [0.01,1000]
```

A.1.2 Constraints

A mixer may optionally have a condenser or a cooler depending on the mixing task for which it is used. The activity constraints are therefore:

$$\begin{aligned}\mathbf{AC}(\mathbf{M}, \mathbf{V}.\mathbf{cooler}) :=& \quad M = \ reactor \wedge V \rightarrow Co \\ \mathbf{AC}(\mathbf{V}.\mathbf{volume}) :=& \quad V.volume \geq 150 \rightarrow C\end{aligned}$$

The three categories discrete, mixed and continuous constraints are present in this example:

$\mathbf{C}(\mathbf{V}, \mathbf{MT}.\mathbf{slurry\ pressure}) :=$
$\{(hemispherical\ high)(elliptical\ high)(cylindrical\ low)(cylindrical\ high)\}$

$\mathbf{C}(\mathbf{MT}, \mathbf{MT}.\mathbf{slurry\ viscosity}, \mathbf{V}.\mathbf{volume}, \mathbf{I}.\mathbf{entry}) :=$
$\{(blending\ low\ small\ offcenter)(blending\ low\ large\ side)$
$all\ other\ combinations\ with\ top\}$

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

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

$\mathbf{C}(\mathbf{MT}, \mathbf{C}) :=$
$\{(dispersion\ condensor\}$

$\mathbf{C}(\mathbf{MT}, \mathbf{MT}.\mathbf{slurryviscosity}, \mathbf{V}.\mathbf{volume}, \mathbf{I}) :=$
$\{(blending\ high\ [0, 1000]\ anchorstirrer)$
$(blending\ high\ [0, 1000]\ hellicalribbon)$
$(blending\ low\ [0, 1000]\ propeller)$
$(blending\ low\ [1, 100]\ axialturbine)$
$(entrainment\ low\ [0, 1000]\ radialturbine)$
$(entrainment\ high\ [0, 1000]\ radialturbine)$
$(suspension\ low\ [0, 1000]\ axialturbine)$
$(suspension\ high\ [0, 1000]\ axialturbine)$
$(dispersion\ low\ [0, 1000]\ radialturbine)$
$(dispersion\ high\ [0, 1000]\ radialturbine)$

$\mathbf{C}(\mathbf{MT}, \mathbf{MT}.\mathbf{slurryviscosity}, \mathbf{V}.\mathbf{volume}, \mathbf{I}.\mathbf{entry}) := $
$\{(blending\ low\ [0,1]\ offcenter)$
$(blending\ low\ [100,1000]\ side)$
$(blending\ high\ [0,1000]\ top)$
$(entrainment\ low\ [0,1000]\ top)$
$(entrainment\ high\ [0,1000]\ top)$
$(dispersion\ low\ [0,1000]\ top)$
$(dispersion\ high\ [0,1000]\ top)$
$(suspension\ low\ [0,1000]\ top)$
$(suspension\ high\ [0,1000]\ top)$

$\mathbf{C}(\mathbf{I}, \mathbf{I}.\mathbf{ratio}) := $
$\{(axialturbine\ 0.399414)(denteddisk\ 0.5)(anchorstirrer\ 0.949219)$
$(hellicalribbon\ 0.949219)\ for\ all\ others\ 0.329102\}$

$\mathbf{C}(\mathbf{I}, \mathbf{I}.\mathbf{position}) := $
$\{(radialturbine\ 5)(axialturbine\ 1.5)(propeller\ 0.35)$
$for\ all\ others\ [0.15, 0.2]\}$

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

$El.sradius \leq 1/2 * V.diameter$

$El.bottomArea = El.bottomArea * V.height$

$V.volume = El.sradius * V.diameter$

$1/2 * V.diameter \leq V.height \leq 2 * V.diameter$

$I.diameter = I.ratio * V.diameter$

$I.rps \leq V.diameter / I.diameter$

$I.power = MT.slurrydensity * I.rps^3 * I.pos * I.diameter^5$

$E.power \geq 2 * I.power$

A.1.3 Locally consistent solution spaces

All locally consistent solution spaces resulting from the eight problem spaces are listed in Table A.1.3.

A.2 Preliminary design of bridges

This bridge example has been elaborated by Sylvie Boulanger, Steel Structures Institute, EPFL.

A.2.1 Variables

```
Type gap:
  width : real : [20,1000]
  depth : real : [5,200]
  ratio : real : [0,1]
  soil  : discrete : {very-poor, poor, average, good, excellent}
  geometry : discrete : {symmetrical, mostly-symmetrical, not-symmetrical}

Type obstacle:
  domain : {valley, road, river, railway}
  size : real : [0,1000]
  rsize : real : [0.001,1]
  x1 : real : [0,1000]
  x2 : real : [0,1000]

Type road of obstacle
  width : real : [10,15]
  alignment : real : {linear, slightly-curved, curved}

Type bridge:
  domain : discrete : {beam, arch, cable, frame}
  blength : real : [1,1000]
  abutment1 : real: [0,1000]
  abutment2 : real: [0,1000]
  abutments : real: [0,1000]
  profile  : discrete : {const-depth, haunched, at-foundation, above-foundation,
                      below-deck, through-deck, above-deck, stayed-1,
                      stayed-2, suspension}
  construction : discrete : {by-crane, launching, cantilever}
  girder : discrete : {i-shape, open-box, closed-box, truss}
  nbspans : integer : [1,10]
  spanref : Spanref
  spans : array : Span

Type Beam of bridge :
  spantype : Spantyp
```

```
Type Span:
  llength :  real : [1,1000]
  aux : real : [1,1000]

Type Spanref
  llength : real : [1,1000]

Type Spantyp
  llength : real : [1,1000]
```

A.2.2 Constraints

Some constraints are indicated as logical implications in the form IF .. THEN .. in order to avoid the enumeration of a large number of tuples.

$\mathbf{C}(\mathbf{O}.\mathbf{rsize}\ \mathbf{G}.\mathbf{depth}\ \mathbf{Spanref}.\mathbf{llength}) :=$
$\{([0.15, 1]\ [5, 200]\ [15, 25])$
$([0, 0.15]\ [5, 75]\ [45, 60])$
$([0, 0.15]\ [75, 200]\ [15, 20])\}$

$\mathbf{C}(\mathbf{B}\ \mathbf{Spanref}.\mathbf{llength}) :=$
$IF\ B = beam\ THEN\ Spanref.llength = [10, 250]$

$\mathbf{C}(\mathbf{B}\ \mathbf{B}.\mathbf{profile}) :=$
$\{(beam\ constDepth)$
$(beam\ haunched)$
$(frame\ atFoundation)$
$(frame\ aboveFoundation)$
$(arch\ belowDeck)$
$(arch\ throughDeck)$
$(arch\ aboveDeck)$
$(cable\ stayed1)$
$(cable\ suspension)$
$(cable\ stayed2)\}$

C(B G.width G.ratio B.profile) $:=$
$IF\ B = arch \wedge G.width \leq 200 \wedge G.ratio \leq 0.25\ THEN$
$\neg(B.profile = throughDeck)$

C(B B.spanref B.profile) $:=$
$IF\ B = arch \wedge B.spanref \leq 50\ THEN$
$\neg(B.profile = haunched)$
$ELIF\ B = frame \wedge B.spanref \leq 80\ THEN$
$\neg(B.profile = aboveFoundation)$

C(B G.width B.profile) $:=$
$IF\ B = cable \wedge G.width < 150\ THEN$
$\neg(B.profile = stayed2)$
$ELIF\ B = cable \wedge G.width < 600\ THEN$
$\neg(B.profile = suspension)$

C(B.profile G.width B.nbspans) $:=$
$IF\ B.profile = aboveDeck \wedge G.width < 200\ THEN$
$B.nbspans = 1$
$ELIF\ B.profile = constantDepth\ THEN$
$B.nbspans = 3 \vee B.nbspans = 5$

C(B.profile G.depth G.soil Spantyp.llength) $:=$
$IF\ B.profile = constDepth \wedge G.depth < 50\ \wedge$
$(G.soil = average \vee G.soil = good)\ THEN$
$Spantype.llength = [30, 60]$

$$O.size = O.x2 - O.x1$$
$$O.rsize = O.size/G.width$$
$$G.ratio = G.depth/G.width$$
$$B.abutments = B.abutment1 + B.abutment2$$
$$G.width = B.blength + B.abutment$$
$$B.nbspans < B.blength/(0.9 * spantyp.llength)$$
$$B.spans[i].llength \geq 0.95 * (B.blength/B.nbspans)$$
$$B.spans[i].llength \leq 1.05 * (B.blength/B.nbspans)$$
$$spans[i].aux = spans[i].llength + spans[i-1].aux$$
$$spans[1].aux = spans[1].llength$$
$$spans[n].llength = B.blength$$

Generator: $B.spans$ with counter $B.nbspans$
The last three equations verify that $\sum_{i=1}^{B.nbspans} spans[i] = B.blength$.

M	{mixer,storage-tank}	reactor	reactor
M.agitator	agitator	agitator	agitator
M.vessel	{cylindrical,hemispherical}	elliptical	{cylindrical,hemispherical}
MT	dispersion	{susp,entr,blend,disp}	{susp,entr,blend,disp}
MT.heat-transfer	false	true	true
MT.slurry-pressure	{low,high}	high	{low,high}
MT.slurry-viscosity	{low,high}	{low,high}	{low,high}
MT.slurry-density	[1,2000]	[1,2000]	[1,2000]
A.engine	engine	engine	engine
A.impellers	radial-turbine	$D_{Impellers}$	$D_{Impellers}$
A.shaft	shaft	shaft	shaft
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]	[0.01,100]
V.opt1	-	cooler	cooler
V.opt2	condenser	-	-
V.elliptical	-	elliptical	-
EL.sradius	-	[0.01,10.6172]	-
El.bottomArea	-	[0.01,354.1355]	-

M	{mixer,storage-tank}	{mixer,storage-tank}	{mixer,storage-tank}
M.agitator	agitator	agitator	agitator
M.vessel	elliptical	elliptical	{cylindrical,hemispherical}
MT	dispersion	{susp,entr,blend,disp}	{susp,entr,blend,disp}
MT.heat-transfer	false	false	false
MT.slurry-pressure	high	high	{low,high}
MT.slurry-viscosity	{low,high}	{low,high}	{low,high}
MT.slurry-density	[1,523.0584]	[1,2000]	[1,2000]
A.engine	engine	engine	engine
A.impellers	radial-turbine	$D_{Impellers}$	$D_{Impellers}$
A.shaft	shaft	shaft	shaft
E.power	[9.5591,5000]	[0.2,5000]	[0.2,5000]
I.diameter	[0.991,3.4658]	[0.4573,6.9883]	[0.4573,6.9883]
I.entry	top	D_{entry}	D_{entry}
I.position	5	D_{pos}	D_{pos}
I.power	[4.7795,2500]	[0.1,2500]	[0.1,2500]
I.ratio	0.329102	D_{ratio}	D_{ratio}
I.rps	[1,8.0572]	[1,29.2402]	[1,29.2402]
V.diameter	[3.0112,10.5309]	[0.4817,21.2344]	[0.4817,21.2344]
V.height	[1.7221,21.0618]	[0.2408,42.4688]	[0.2408,42.4688]
V.volume	[150,1000]	[0.01,150]	[0.01,150]
V.opt1	-	-	-
V.opt2	condenser	-	-
V.elliptical	elliptical	elliptical	-
EL.sradius	[0.4305,5.2655]	[0.01,10.6172]	-
EL.bottomArea	[7.1219,87.09994]	[0.01,354.1355]	-

Appendix B

Topology

In this appendix, we will give definitions of some important topological concepts used in this thesis. These concepts are among others the boundary of a region as well as properties like connectedness or compactness. They are taken from [Gellert et al., 1975], [Bloch, 1997].

$\mathcal{N}_\epsilon(p)$ is the set of all points whose distance from point p is less than ϵ, where ϵ is an arbitrary positive number.

$$\mathcal{N}_\epsilon(p) = \{x \| \|x - p\| < \epsilon\} \quad x \in \mathbb{R}^N, \epsilon > 0.$$

A set E is *open* if for every $x \in E$, there exists $\epsilon > 0$ such that $\mathcal{N}_\epsilon(x) \subset E$. A set E is *closed* if its complement E^c is open; we denote a closed space by $\overline{E}$. The boundary $B(E)$ of a set E is the set of points whose ϵ-neighborhood is not entirely contained in E (Figure B.1). We use the word *region* to designate an arbitrary set of points R in $\mathbb{R}^2$ or $\mathbb{R}^3$.

A subset E of $\mathbb{R}^N$ is *bounded* if there exists a real number M such that for each $x \in E$, $\|x\| < M$. A set E is *compact* if for each collection of open sets covering E, a finite subset of sets can be extracted from the collection which covers E. In $\mathbb{R}^N$, a compact set is closed and bounded and vice versa.

In our thesis, regions in Euclidean space are considered. A region consists of a point set E which is characterized with respect to its connectedness. A set E is *path-connected* if any two points p and q of E are connected by a path in E. In other words, a connected region cannot be represented as the union of two non-empty disjoint open sets. Furthermore, we

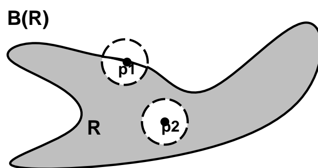


Figure B.1: *Point p_1 lies on the boundary of region R whereas p_2 is an interior point of R.*

consider the maximal set of points which is connected as one region; e.g each region is connected and there exists no region such that the former is a proper subset of it. A path-connected region in $\mathbb{R}^2$ or $\mathbb{R}^3$ can still have *holes*. Certain types of these can be excluded by requiring a set E to be *simply-connected*. A set E is simply-connected if any closed curve in E can be contracted inside E to a point. Simply-connected figures in the plane thus cannot have holes. In three-dimensional regions, this condition excludes channels but not cavities. Here, holes can be of two types: *cavities* like in a Swiss cheese or *channels* like in a sieve.

Appendix C

Analysis

The following definitions are from [Gellert et al., 1975], [Douchet and Zwahlen, 1983], [Douchet and Zwahlen, 1986].

A function $F(X)$ is *continuous* at a point x_0 if $lim_{X \to x_0} F(X) = F(x_0)$. $lim_{X \to x_0}$ is equivalent to $lim_{X \to x_0^-} F(X) = lim_{X \to x_0^+} F(X)$. It is continuous over an open interval I if it is continuous in every point of I.

Given $n \geq 2$, a subset S of $\mathbb{R}^n$ and $E : S \to \mathbb{R}$ a function continuous at a point $a = (a_1, \dots, a_n)$ with partial derivatives also continuous at a and such that $E(a_1, \dots, a_n) = 0$ and $\partial E(a_1, \dots, a_n)/\partial X_n \neq 0$. The *Implicit Function Theorem* states that there exists a function F in the neighborhood of a such that $a_n = F(a_1, \dots, a_{n-1})$. This allows us to find the partial derivatives of F in the neighborhood of a:

$$\partial F/\partial X_i = \frac{\partial E/\partial X_i}{\partial E/\partial X_n} \quad i \neq n$$

A point meeting theses conditions is *regular*, i.e. a single tangent at the surface exists at that point. Otherwise, it is a *singular* point. A singular point is to be expected if all partial derivatives are zero: $\partial F/\partial X_i = 0$, $i = 1, \dots, n$ (necessary but not sufficient condition). A sufficient condition is that all partial derivatives of the function given in a parametric representation are zero. A Taylor expansion at the singular point x can be used to determine the type of the singularity at x: singular points with two distinct tangents (a double point), with two coincident tangents (a double point) and with no tangents (an isolated point).

In vector analysis, a function $\phi(X, Y)$ is called a scalar field. A scalar $\phi(\vec{r})$ is assigned to each point with position vector $\vec{r} = \begin{bmatrix} X \\ Y \end{bmatrix}$. Such a scalar field can be visualized by the level surfaces at $\phi(X, Y) = const$. The level surfaces of the scalar field $X^2 + Y^2 + Z^2 = c^2$ are circles with center at the origin. Given the unit vectors $\vec{i}, \vec{j}, \vec{k}$ of the Euclidean space, the gradient of a scalar field is the magnitude of increase of ϕ in all directions of the unit vectors: $\nabla\phi = \partial\phi/\partial X\vec{i} + \partial\phi/\partial Y\vec{j} + \partial\phi/\partial Z\vec{k}$. A small change of the field along a vector $\vec{r}$

is thus measured by $d\phi = \nabla\phi * d\vec{r}$. If $\vec{r}$ is chosen along a level surface ($\phi = const$) then $d\phi = 0$ and the gradient is perpendicular to the level surface. Furthermore, ϕ increases most rapidly in direction of the steepest gradient. In other words, the gradient points in direction of increasing values of ϕ, or towards the feasible region defined by $\phi(X, Y) \geq 0$.

An important concept is that of *convexity*. A function $f : I \rightarrow F, I \subset \mathbb{R}$ is convex if for two points $a, b \in I$ and for $\lambda \in [0,1]$ we have $f(\lambda * a + (1-\lambda) * b) \leq \lambda * f(a) + (1-\lambda) * f(b)$. It is concave if $f(\lambda * a + (1-\lambda) * b) \geq \lambda * f(a) + (1-\lambda) * f(b)$. The convexity of a function should not be confused with the convexity of a region: A set $C \in \mathbb{R}^n$ is convex if for two points $\vec{x}, \vec{y} \in C$ and for $\lambda_1, \lambda_2 \geq 0$ such that $\lambda_1 + \lambda_2 = 1$, $\lambda_1 * \vec{x} + \lambda_2 * \vec{y} \in C$. In other words, if two points $\vec{x}$ and $\vec{y}$ are part of C then the line segment joining the two points is also part of C.

The following lemma shows a way for characterizing local extrema by gradients:

Lemma C.1 *Let R be a region in $\mathbb{R}^2$ such that the boundary $Y = F(X)$ determined by $E(X, Y) = 0$ is a continuous, twice continuously differentiable function over an interval I and it is convex (concave) over I. Let (x_1, y_1),(x_2, y_2), $x_1 < x_2$ be two points of $F(X)$ with $x_1, x_2 \in I$. Assume further that there exists $\beta > 0$ such that $\nabla E(x_1, y_1) + \beta * \nabla E(x_2, y_2) = \vec{N}$ with $\vec{N}$ parallel to the Y-axis. Then, there exists a local minimum (maximum) (x_e, y_e) in Y with $x_1 < x_e < x_2$.*

Proof: If $F(X)$ is convex (concave), its first derivative $F'(X)$ is increasing (decreasing) and $F''(X)$ is positive (negative) over I ([Douchet and Zwahlen, 1983]). From $\nabla E(x_1, y_1) + \beta * \nabla E(x_2, y_2) = \vec{N}$ and $\beta > 0$ follows $\partial E(x_1, y_1)/\partial X + \beta * \partial E(x_2, y_2)/\partial X = 0$ which implies that $\partial E(x_1, y_1)/\partial X$ and $\partial E(x_2, y_2)/\partial X$ are of different signs. Since $F'(X)$ is strictly increasing (decreasing) there must exist a point (x, y) for which $F'(X)$ is zero and $F''(X)$ is positive (negative). By definition, (x, y) is a local minimum (local maximum) of $F(X)$ ▲

Interval Analysis

The natural extension of a real function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ to an interval function $F(I) : \mathbb{I} \rightarrow \mathbb{I}$, with I an interval and $\mathbb{I}$ the set of all intervals, is computable by the bounds of I if f is continuous and monotonic over I. Otherwise, I has to be decomposed into a series of monotonic and continuous subintervals and $F(I)$ is applied to each subinterval. The real function $Y = exp(X)$, for example, can be extended directly to an interval function $F([a, b]) = [exp(a), exp(b)]$ whereas a sine function has to be decomposed into subintervals of the form $[k * \pi/2, (k+1) * \pi/2]$ for $k \in \mathbb{N}$.

Appendix D

Graph theory

Since a constraint problem can be represented as a graph, graph-related concepts are important in CSP research. We define here some terminology used in our work.

Definition D.1 (Hypergraph; [Gondran and Minoux, 1986]) *A* hypergraph $\mathcal{H}$ *is a tuple* $\langle \mathcal{S}, \mathcal{E} \rangle$ *with*

1. $\mathcal{S}$ *a set of nodes* $\{s_1, \ldots, s_n\}$
2. $\mathcal{E}$ *a set of edges, each being a set of nodes* $\{E_1, \ldots, E_m\}$

such that $\begin{cases} E_i \neq \emptyset & i = 1, \ldots, m \\ \bigcup_i E_i = \mathcal{S} \end{cases}$

Each hyperedge consists of a set of nodes. This generalizes the usual notion of an edge connecting two nodes. A k-ary constraint problem $\mathcal{P} = \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ can be represented by a hypergraph $\mathcal{H} = \langle \mathcal{S}, \mathcal{E} \rangle$ such that each variable in $\mathcal{V}$ represents a node in $\mathcal{S}$ and each edge in $\mathcal{E}$ a constraint in $\mathcal{C}$. A directed graph is a graph the edges of which are oriented.

Definition D.2 (Directed graph; [Gondran and Minoux, 1986]) *A* directed graph $\mathcal{G}$ *is a tuple* $\langle \mathcal{X}, \mathcal{U} \rangle$ *with*

1. $\mathcal{X}$ *a set of nodes*
2. $\mathcal{U}$ *a set of ordered tuples* $\{\ldots, (u, v), \ldots\}$, $u, v \in \mathcal{X}$ *called arcs*

A *successor node* $s_{i+1} \in \mathcal{X}$ of a node s_i is a node such that $(s_i, s_{i+1}) \in \mathcal{U}$. s_i is called *predecessor* of s_{i+1}. An undirected graph, called graph for short, has edges $\{u, v\} \in \mathcal{U}$ instead of arcs.

The concept of cycles in a graph is important to define subclasses of CSPs which are solvable without backtracking. A *cycle* is a chain of length q if $\{s_1, E_1, s_2 \ldots, s_q, E_q, s_{q+1}\}$ so that $s_1 = s_{q+1}$ and $\forall_{k=1}^{q} s_k \in E_k \wedge s_{k+1} \in E_k$. A *(hyper)tree* is a connected (hyper)graph with no cycles. The *root* of a tree is a node with no predecessors and *leaf nodes* in a tree are nodes which have no successors.

A graph $\langle \mathcal{X}, \mathcal{U} \rangle$ is *strongly connected* if for each pair of nodes $i, j \in \mathcal{X}$ there is a path from i to j and a path from j to i or $i = j$. This is an equivalence relation. If only a set of nodes $X\mathcal{X}$ of the graph is strongly connected, X is called a *strongly connected component.*

Bibliography

[Alefeld and Herzberger, 1983] Alefeld, G. and Herzberger, J. (1983). *Introduction to Interval Computation*. Academic Press, NY.

[Avriel, 1976] Avriel, M. (1976). *Nonlinear Programming: Analysis and Methods*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, New Yersey.

[Benhamou, 1996] Benhamou, F. (1996). Heterogeneous constraint solving. In Hanus, M. and Rodríguez-Artalejo, M., editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 62–76, Aachen, Germany. Springer.

[Benhamou et al., 1994] Benhamou, F., McAllester, D., and Hentenryck, P. V. (1994). CLP(intervals) revisited. In Bruynooghe, M., editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 124–138, Massachusetts Institute of Technology. The MIT Press.

[Bessière, 1994] Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190.

[Bessière and Régin, 1996] Bessière, C. and Régin, J.-C. (1996). Mac and combined heuristics: two reasons to forsake fc (and cbj). In Freuder, E. and Jampel, M., editors, *Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*. Springer.

[Bessière and Régin, 1997] Bessière, C. and Régin, J.-C. (1997). Arc consistency for general constraint networds: preliminary results. In *IJCAI-93*, pages 398–404.

[Bloch, 1997] Bloch, E. D. (1997). *A first course in Geometric Topology and Differential Geometry*. Birkhäuser-Verlag, Boston.

[Boulanger et al., 1995] Boulanger, S., Gelle, E., and Smith, I. (1995). Taking advantage of design process models. In *IABSE Colloquium*.

[Bowen and Bahler, 1991] Bowen, J. and Bahler, D. (1991). Conditional existence of variables in generalized constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 215–220.

[Brown and Chandrasekaran, 1988] Brown, D. C. and Chandrasekaran, B. (1988). *Design Problem Solving, Knowledge Structures and Control Strategies.* Research series in Artificial Intelligence. Morgan Kaufman.

[Buchberger, 1965] Buchberger, B. (1965). *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal.* PhD thesis, Institut fur Mathematik, Universitat Innsbruck.

[Buchberger, 1985] Buchberger, B. (1985). *Groebner Bases: An Algorithmic Method in Polynomial Ideal Theory*, pages 184–232. N.K. Bose Ed., D. Reidel Publishing Co.

[Chiu and Lee, 1994a] Chiu, C. K. and Lee, J. H. M. (1994a). Interval linear constraint solving using the preconditioned interval gauss-seidel method. In Yap, R. H. C., editor, *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.

[Chiu and Lee, 1994b] Chiu, C. K. and Lee, J. H. M. (1994b). Towards practical interval constraint solving in logic programming. In Bruynooghe, M., editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 109–123, Massachusetts Institute of Technology. The MIT Press.

[Cohen, 1990] Cohen, J. (1990). Constraint logic programming. *Communications of the ACM*, 33(7).

[Collins, 1975] Collins, G. E. (1975). *Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition, LNCS 32.* Springer Verlag.

[Colmerauer, 1993] Colmerauer, A. (1993). Naive solving of non-linear constraints. In Benhamou, F. and Colmerauer, A., editors, *Constraint Logic Programming, Selected Research*, pages 88–112. The MIT Press.

[Cooper, 1989] Cooper, M. C. (1989). An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1):89–95.

[Coyne et al., 1990] Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M., and Gero, J. S. (1990). *Knowledge-based design systems.* Addison Wesley, 1990, 567 pages.

[Danzig, 1965] Danzig, G. G. (1965). *Linear programming and extensions.* Princeton University Press.

[Dasgupta, 1991] Dasgupta, S. (1991). *Design Theory and Computer Science.* Cambridge Tracts in Theoretical Computer Science 15. Cambridge University Press.

[Davis E., 1987] Davis E. (1987). Constraint propagation with interval labels. In *Artificial Intelligence 32.*

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

[Douchet and Zwahlen, 1983] Douchet, J. and Zwahlen, B. (1983). *Calcul différentiel et intégral: Fonctions réelles d'une variable réelle*, volume 1. Presses polytechniques romandes, Lausanne.

[Douchet and Zwahlen, 1986] Douchet, J. and Zwahlen, B. (1986). *Calcul différentiel et intégral:Fonctions réelles de plusieures variables réelles*, volume 2. Presses polytechniques romandes, Lausanne.

[Faltings, 1994] Faltings, B. (1994). Arc consistency for continuous variables. In *Artificial Intelligence 65(2)*.

[Frayman and Mittal, 1987] Frayman, F. and Mittal, S. (1987). Cossack: A constraints-based expert system for configuration tasks. In Sriram, D. and Adey, R., editors, *Knowledge Based Expert Systems in Engineering: Planning and Design*, pages 143–166. Computational Mechanics Publications.

[Freeman-Benson et al., 1990] Freeman-Benson, B. N., Maloney, J., and Borning, A. (1990). An incremental constraint solver. *Communications of the ACM*, 33(1):54–63.

[Freuder and Hubbe, 1995] Freuder, E. and Hubbe, P. (1995). A disjunctive decomposition control schema for constraint satisfaction. In Saraswat, V. and Hentenryck, P. V., editors, *Principle and Practice of Constraint Programming*, The Newport Papers. The MIT Press.

[Freuder, 1978] Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966. Also published as MIT AI MEMO 370, Cambridge, MA, USA, 1976.

[Freuder, 1982a] Freuder, E. C. (1982a). A sufficient condition for backtrack-bounded search. *A.C.M.*, 32(4):755–761.

[Freuder, 1982b] Freuder, E. C. (1982b). A sufficient condition for backtrack-free search. *A.C.M.*, 29(1):24–32.

[Fulton, 1995] Fulton, W. (1995). *Algebraic Topology, a first Course.* Springer-Verlag, New York.

[Gaschnig, 1977] Gaschnig, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In Reddy, R., editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 457–457, Cambridge, MA. William Kaufmann.

[Gaschnig, 1979] Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University.

[Gelle and Smith, 1996] Gelle, E. and Smith, I. (1996). Dynamic constraint satisfaction with conflict management in design. In Jampel, M., Freuder, E., and Maher, M., editors, *OCS'95 Workshop on Over-Constrained Systems at CP'95*, pages 237–251. LNCS Springer Verlag.

[Gellert et al., 1975] Gellert, W., Gottwald, S., Hellwich, M., Kästner, H., and Küster, H. (1975). *Concise Encyclopedia of Mathematics.* Van Nostrand Reinhold, NY.

[Gero, 1990] Gero, J. S. (1990). Design prototypes: a knowledge representation schema for design. *AI Magazine, Winter 1990*, 11(4):26–48.

[Glover, 1989a] Glover, F. (1989a). Tabu search–Part I. *ORSA Journal on Computing*, 1(3):190–206.

[Glover, 1989b] Glover, F. (1989b). Tabu search, part II. *ORSA Journal on Computing*, 2:4–32.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA.

[Golumbic, 1980] Golumbic, M. (1980). *Algorithmic Graph Theory and Perfect Graphs.* Acadmic Press, New York, NY.

[Gondran and Minoux, 1986] Gondran, M. and Minoux, M. (1986). *Graphs and Algorithms.* John Wiley & Sons, Chichester, 1 edition.

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313.

[Haroud et al., 1995] Haroud, D., Boulanger, S., Gelle, E., and Smith, I. (1995). Management of conflict for preliminary engineering design tasks. In *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, volume 9, pages 313–323. Cambridge University Press.

[Haselböck, 1993] Haselböck, A. (1993). *Knowledge-based Configuration and Advanced Constraint Technologies.* PhD thesis, Technical University of Vienna.

[Heintze et al., 1987] Heintze, N. C., Michaylov, S., and Stuckey, P. J. (1987). CLP($\mathcal{R}$) and some electrical engineering problems. In Lassez, J.-L., editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 675–703, Melbourne. The MIT Press.

[Hollman and Langemyr, 1993] Hollman, J. and Langemyr, L. (1993). Algorithms for non-linear algebraic constraints. In Benhamou, F. and Colmerauer, A., editors, *Constraint Logic Programming, Selected Research*, pages 113–131. The MIT Press.

[Hong, 1993] Hong, H. (1993). Risc-clp(real): Logic programming with non-linear constraints. In Benhamou, F. and Colmerauer, A., editors, *Constraint Logic Programming, Selected Research*, pages 133–159. The MIT Press.

[Hyvönen, 1992] Hyvönen, E. (1992). Constraint reasoning based on interval arithmetic. the tolerance propagation approach. In *Artificial Intelligence*.

[Jaffar and Maher, 1994] Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582.

[Jégou, 1993] Jégou, P. (1993). On the consistency of general constraint-satisfaction problems. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 114–119, Menlo Park, CA, USA. AAAI Press.

[Kondrak, 1994] Kondrak, G. (1994). A theoretical evaluation of selected backtracking algorithms. Technical Report TR-94-10, University of Alberta.

[Kondrak and van Beek, 1995] Kondrak, G. and van Beek, P. (1995). A theoretical evaluation of selected backtracking algorithms. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 541–547, San Mateo. Morgan Kaufmann.

[Kumar, 1992] Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44.

[Lakmazaheri and Rasdorf, 1989] Lakmazaheri, S. and Rasdorf, W. J. (1989). Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157–173.

[Lhomme, 1993] Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *IJCAI-93*, pages 232–238.

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

[Mackworth, 1977b] Mackworth, A. (1977b). On reading sketch maps. *IJCAI-77*, pages 598–606.

[Mackworth et al., 1985] Mackworth, A., Mulder, J., and Havens, W. (1985). Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126.

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

[McGregor, 1979] McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250.

[Metropolis et al., 1953] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091.

[Milnor, 1963] Milnor, J. (1963). *Morse Theory*, volume 51 of *Annals of Mathematics Studies*. Princeton University Press, Princetom.

[Mittal and Falkenhainer, 1990] Mittal, S. and Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In Dietterich, Tom; Swartout, W., editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32. MIT Press.

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

[Mohr and Masini, 1988] Mohr, R. and Masini, G. (1988). Good old discrete relaxation. In Kodratoff, Y., editor, *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munich, FRG. Pitman Publishers.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132.

[Moore, 1966] Moore, R. E. (1966). *Interval Analysis*. Prentice-Hall, Englewood Cliffs,NJ.

[Nash and Sofer, 1996] Nash, S. G. and Sofer, A. (1996). *Linear and Nonlinear Programming*. The McGraw-Hill Companies, Inc.

[Nelson, 1985] Nelson, G. (1985). Juno, a constraint-based graphics system. *Computer Graphics*, 19(3):235–243.

[Older and Vellino, 1993] Older, W. and Vellino, A. (1993). Constraint arithmetics on real intervals. In Benhamou, F. and Colmerauer, A., editors, *Constraint Logic Programming, Selected Research*, pages 175–195. The MIT Press.

[Pesant and Boyer, 1994] Pesant, G. and Boyer, M. (1994). QUAD-CLP(R): Adding the power of quadratic constraints. In Borning, A., editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[Prosser, 1993a] Prosser, P. (1993a). Domain filtering can degrade intelligent backjumping search. In *IJCAI-93*.

[Prosser, 1993b] Prosser, P. (1993b). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[Régin, 1996] Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 209–215, Menlo Park. AAAI Press / MIT Press.

[Reiner Horst and Thoai, 1995] Reiner Horst, P. M. P. and Thoai, N. V. (1995). *Introduction to Global Optimization*, volume 3 of *Nonconvex Optimization and its Applications*. Kluwer Academic Publisher.

[Sabin and Freuder, 1994] Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In Borning, A., editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[Sabin and Freuder, 1996] Sabin, D. and Freuder, E. C. (1996). Configuration as composite constraint satisfaction. In *Configuration – Papers from the 1996 Fall Symposium. AAAI Tech nical Report FS-96-03*.

[Sakai and Aiba, 1989] Sakai, K. and Aiba, A. (1989). CAL: A theoretical background of CLP and its applications. *Journal of Symbolic Computation*, 8(6):589–603.

[Sam-Haroud, 1995] Sam-Haroud, D. (1995). *Constraint Consistency Techniques for Continuous Domains*. PhD thesis, Swiss Federal Institute of Technology, EPFL.

[Sam-Haroud and Faltings, 1996] Sam-Haroud, D. and Faltings, B. (1996). Consistency techniques for continuous constraints. In *Constraints*, volume 1, pages 85–118.

[Sannella, 1993] Sannella, M. (1993). The SkyBlue constraint solver and its applications. In Kanellakis, P., Lassez, J.-L., and Saraswat, V., editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI.

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

[Schmidt and Ströhlein, 1988] Schmidt, G. and Ströhlein, T. (1988). *Relationen und Grafen*. Springer-Verlag.

[Serrano and Gossard, 1992] Serrano, D. and Gossard, D. (1992). Tools and techniques for conceptual design. In Tong, C. and Sriram, D., editors, *Artificial Intelligence in Engineering Design*, volume Volume 1: Design representation and models of routine design, pages 71–116. Academic Press, Inc.

[Simon, 1981] Simon, H. A. (1981). *The Sciences of the Artificial*. MIT Press, Cambridge, Massachusetts, second edition.

[Soininen and Niemelä, 1998] Soininen, T. and Niemelä, I. (1998). Formalizing configuration knowledge using rules with choices. Technical Report TKO-B142, Laboratory of Information Processing, Helsinki University of Technology (HUT), Helsinki, Finland.

[Sutherland, 1963] Sutherland, I. E. (1963). Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, Detroit, Michigan.

[Thornton, 1993] Thornton, A. (1993). *Constraint specification and satisfaction in embodiment design.* PhD thesis, Department of Engineering, University of Cambridge, UK.

[Tinelli and Harandi, 1996] Tinelli, C. and Harandi, M. (1996). Constraint logic programming over unions of constraint theories. *Lecture Notes in Computer Science*, 1118:436–450.

[Trombettoni and Neveu, 1997] Trombettoni, G. and Neveu, B. (1997). Computational complexity of multi-way, dataflow constraint problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI '97)*, pages 358–363, Nagoya.

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

[Tsang, 1998] Tsang, E. (1998). No more partial and full looking ahead. *Artificial Intelligence*, 98(1-2):351–361.

[van Beek, 1992] van Beek, P. (1992). On the minimality and decomposability of constraint networks. In Swartout, W., editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 447–452, San Jose, CA. MIT Press.

[Van Hentenryck, 1997] 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.

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

[Van Hentenryck et al., 1995] Van Hentenryck, P., McAllester, D., and Kapur, D. (1995). Solving polynomial systems using a branch and prune approach. *SIAM Journal of Numerical Analysis.* (Accepted). (Also available as Brown University technical report CS-95-01.).

[van Velzen, 1993] 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.

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

Curriculum Vitae

Personal Data

Name:	Esther Gelle
Date and Place of Birth:	November 27, 1966 in Onex/Geneva, Switzerland
Nationality:	Swiss
Languages:	German, Swiss German, French, English, Spanish

Education

1983 - 1987	Kantonsschule Alpenquai, Luzern, Degree: Kantonale Maturität, Typus B (latin and modern languages), Award: Dr. Robert Huber
1987 - 1988	Special Mathematics Course (CMS), Ecole Polytechnique Fédérale de Lausanne (EPFL)
1988 - 1993	Computer Science Department, EPFL Degree: Diplôme d'ingénieur informaticien EPF Diploma Thesis: Diagnostic System for a heating-cooling unit in collaboration with Landis & Stäfa, Zug, Switzerland
1993 - 1998	Research assistant for Prof. B.V. Faltings, Artificial Intelligence Laboratory (LIA), EPFL Degree: Docteur ès sciences, june 1998 Ph.D. Thesis: On the generation of locally consistent solution spaces in mixed dynamic constraint problems

Publications

1. E. Gelle and B. Faltings,
Diagnosis of Heating, Ventilation and Air Conditioning Systems,
Proc. of SPICIS-94 , B309-314,Singapore, 1994

2. S. Boulanger, E. Gelle and I.Smith,
Taking advantage of design process models,
IABSE Colloquium,Bergamo, March 1995

3. D. Haroud , S. Boulanger, E. Gelle and I. Smith,
Management of Conflict for Preliminary Engineering Design Tasks,
Artificial Intelligence for Engineering Design, Analysis and
Manufacturing, 9, 313-323, Cambridge University Press, 1995

4. D. Haroud , S. Boulanger, E. Gelle and I. Smith,
Strategies for Conflict Management in Preliminary Engineering Design,
AID-94 Workshop on Conflict Management in Design, Lausanne, August 1994

5. E. Gelle and I. Smith,
Dynamic Constraint Satisfaction with Conflict Management in Design,
editors: Michael Jampel and Eugene Freuder and Michael Maher,
publisher: LNCS Springer Verlag, 237-251, 1996

6. E. Gelle and R. Weigel,
Interactive Configuration based on Incremental Constraint Satisfaction,
IFIP TC5/WG 5.2 Workshop Series on Knowledge Intensive CAD , 117-126,
Helsinki, September 1995

7. E. Gelle and R. Weigel,
Interactive Configuration using Constraint Satisfaction Techniques,
Second International Conference on Practical Application of Constraint
Technology, PACT-96, 57-72, London, April 1996

8. E. Gelle and R. Weigel,
Interactive Configuration using Constraint Satisfaction Techniques,
AAAI-Symposium on Configuration, 37-44, Boston, November 1996

9. B. Faltings and E. Gelle,
Local Consistency for ternary numeric Constraints,
IJCAI-97, 392-397, Nagoya, Japan, August 1997