A Mathematical Framework for a General Purpose Constraint Management System

by

Steven James Carden

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy.

The University of Leeds
School of Computer Studies

June 1998

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

Abstract

The use of *constraints* in engineering for designing complex models is very popular. Current constraint solvers are divided into two broad classes: *general* and *domain specific*. Those that are general can handle very general constraint problems but are typically slow; while those that are domain specific can handle only a specific type of problem but are typically fast. For example, numerical algorithms are slow but general, whilst local propagation techniques are fast but limited to simple problems.

It is generally acknowledged that there is a close coupling between *engineering constraints* and *geometric constraints* in the design process and so the solution of constraint problems consisting of engineering and geometric constraints is an important research issue. Some authors attempt to overcome the expressive limitations of domain specific solvers by using *hybrid* systems which try to find a balance between the speed of domain specific solvers and the generality of general solvers.

Previous research at the University of Leeds has led to the development of a number of domain specific solvers that are capable of solving geometric and engineering constraint problems separately. In particular, the Leeds solvers are *incremental* and can find solutions when a new constraint is added very quickly. This thesis investigates the use of a hybrid of the various Leeds solvers with an aim to interactively solving constraint problems in engineering design. This hybrid would have the speed advantages of the domain specific solvers and the expressiveness of a more general solver. In order for the hybrid to be constructed, commonalities of existing engineering constraint solvers must be identified. A characterisation of existing constraint solvers leads to the identification of a number of issues that need to be addressed before the hybrid can be built.

In order to examine these issues, a framework for the *constraint satisfaction process* is presented that allows abstractions of *constraint definition*, *constraint representation* and *constraint satisfaction*. Using the constraint satisfaction framework, it is possible to study the *quality of solution* of constraint solvers. This leads to the identification of important problems in current constraint solvers.

The constraint process framework leads to a study of the use of various *paradigms of collaboration* within the hybrid, such as sequential, parallel and concurrent. The study of the quality of solution allows concrete statements to be made about the hybrid collaborations. A new incremental constraint solver is presented that uses the hybrid collaboration paradigms and provides a first step towards a powerful engineering constraint solver.

Acknowledgments

First of all, I would like to thank Pete Dew, my supervisor. Although I was jealous of more ordered projects, I enjoyed the relatively free hand with just enough rope to hang myself. Pete was (usually) there with an encouraging word and just enough guidance to keep me on the right track. Either that or I finally convinced him that *I* was right!

Terrence Fernando, my second supervisor, was also a great support to me, as was the whole Virtual Working Environment group, especially Martin Thompson, Mingxian Fa, Yung-Teng Tsai and Edgard Lamounier.

The people in BGT and the AI lab also kept me comparatively sane and cheerful, especially on trips to the Pennines.

Russ Bubley was a huge help throughout the three years I have known him and shared an office with him. We continually bounced ideas off each other and whilst I tried not to be too ignorant of what he was talking about, he usually managed to put me straight on my project. His capitulation at squash was also particularly gratifying.

My housemates, Mark, Fred and Stuart, were great fun to be with and we've had some great parties.

My parents and siblings, Jim, Carol, Neil and Claire, put up with many a demonstration at the dinner table about what constraints actually were. I think they're still none the wiser.

And finally, I would like to thank Alice, who, besides being wonderful and sympathetic all the way through, also proof-read my thesis, even though it must have sounded like gobbledigook to her.

Contents

1 Introduction **1**

1.1 Objectives of this thesis . . . 3

1.2 Incremental constraint solvers . . . 5

1.3 Thesis organisation . . . 7

2 Related Work **9**

2.1 The theory of constraints . . . 10

2.1.1 Dimensions . . . 12

2.1.2 Decomposition of constraint problems . . . 13

2.1.3 Hybrid constraint solvers . . . 14

2.1.4 Solution spaces . . . 15

2.2 Constraints in engineering design . . . 16

2.3 Constraint solvers . . . 20

2.3.1 General constraint solvers . . . 23

2.3.1.1 Numerical solvers . . . 23

2.3.1.2 Symbolic solvers . . . 24

2.3.2 Finite domain constraint solvers . . . 24

2.3.2.1 Backtracking . . . 25

2.3.2.2 Forward-checking . . . 25

2.3.2.3 Other finite domain research . . . 25

2.3.3 Geometric constraint solvers . . . 26

2.3.3.1 Under-constrained geometric constraint solvers . . . 27

2.3.3.2 Well-constrained geometric constraint solvers . . . 30

2.3.3.3 Over-constrained geometric constraint solvers . . . 32

2.3.4 Functional constraint solvers . . . 33

2.3.4.1 Over-constrained functional constraint solvers . . . 37

2.3.5 Maintenance and physical constraint solvers . . . 40

2.4 Conclusions . . . 40

3 Solving Problems by Decomposition **42**
3.1 Examples of current constraint solvers 45
3.1.1 DCM 45
3.1.2 INCES 48
3.1.3 IGCS 50
3.1.4 Connectivity Analysis 51
3.2 Decomposition strategies 55
3.2.1 Examples of decomposition strategies 56
3.2.2 Decomposition to domain specific subproblems 57
3.2.3 Advantages of decomposition strategies 58
3.2.4 Limitations of decomposition strategies 59
3.2.5 Incremental issues in decomposition strategies 62
3.2.6 Conclusions 65
3.3 Ordering strategies 65
3.3.1 Examples of ordering strategies 66
3.3.2 Ordering strategies for a constraint solver 67
3.3.3 Incremental issues in ordering strategies 69
3.3.4 Conclusions 69
3.4 Solution of subproblems 70
3.4.1 Examples of solution of subproblems 71
3.4.2 Solving using domain specific knowledge 71
3.4.3 Incremental issues in solving subproblems 72
3.4.4 Conclusions 72
3.5 Conclusions 72

4 Constraint Definition **75**
4.1 Entities 77
4.2 Constraints 79
4.3 Constraint problems 84
4.4 Constraint solvers 90
4.5 Dimensions 91
4.5.1 Definition of dimensions 91
4.5.2 Constrainedness 94
4.6 Conclusions 98

5 Constraint Representation **100**
5.1 Representing entities and constraints . . . 102
5.1.1 Finite-domain entities and constraints . . . 102
5.1.2 Infinite-domain entities and constraints . . . 103
5.2 Representing constraint problems . . . 103
5.3 Example constraint representation schemes . . . 105
5.3.1 Algebraic representation . . . 105
5.3.2 Relationship graph representation . . . 106
5.3.3 Undirected graph representation . . . 106
5.3.4 Hypergraph representation . . . 107
5.3.5 Bipartite representation . . . 108
5.3.6 Valid representation schemes . . . 109
5.4 Reductions . . . 110
5.5 Conclusions . . . 117

6 Constraint Satisfaction **119**
6.1 Constraint solution . . . 120
6.1.1 Solution spaces . . . 121
6.2 A framework for the solution process . . . 122
6.2.1 Solution steps . . . 123
6.2.2 Properties of solution steps . . . 124
6.2.3 Solution processes . . . 127
6.2.3.1 Solution processes always head towards a solution . . 130
6.2.4 Solution process properties . . . 131
6.2.5 Using local properties to draw conclusions about processes . . 135
6.2.6 Consequences of the Local-Global Theorem . . . 136
6.3 Enrichment of the constraint satisfaction framework . . . 137
6.3.1 Constraint priorities . . . 138
6.3.2 Variable-driven satisfaction . . . 140
6.3.3 Backtracking . . . 141
6.3.4 Incremental satisfaction . . . 143
6.4 Conclusions . . . 145

7 Hybrid Collaboration **148**
7.1 Using domain specific knowledge in constraint solvers . . . 150
7.1.1 Using domain specific knowledge is fast . . . 152
7.1.2 Using domain specific knowledge is not enough . . . 153

7.2 Hybrid constraint solvers . . . 154
7.2.1 BALI . . . 158
7.2.2 Enhanced solution spaces . . . 160
7.3 A simple example hybrid constraint solver . . . 161
7.4 Paradigms of collaboration . . . 166
7.4.1 Sequential hybrids . . . 166
7.4.1.1 Limitations of serial hybrids . . . 168
7.5 Solver collaboration language . . . 170
7.6 An example of many solvers in serial . . . 172
7.6.1 Case study . . . 172
7.6.2 The solvers used . . . 172
7.6.3 Results . . . 174
7.7 Conclusions . . . 175

8 New Directions 179
8.1 Decomposition strategy . . . 180
8.2 Ordering strategy . . . 186
8.3 Solution and recombination . . . 187
8.4 Advantages of the Erep/IGCS hybrid . . . 190
8.5 Limitations of the Erep/IGCS hybrid . . . 190
8.6 Incremental implications of new solver . . . 191
8.7 Conclusions . . . 195

9 Future Work 197
9.1 The interactive constraint solver . . . 197
9.1.1 The Erep/IGCS hybrid solver . . . 198
9.1.2 A standard interface for solvers . . . 198
9.1.3 Complex case studies . . . 198
9.1.4 Incremental issues . . . 199
9.2 The mathematical framework . . . 199
9.2.1 Inequality constraints . . . 200
9.2.2 Probabilistic constraints . . . 200
9.3 The Virtual Environment . . . 200
9.3.1 Parallel/concurrent collaboration . . . 201
9.3.2 Direct manipulation issues . . . 201
9.4 Summary . . . 201

10 Conclusions **203**

A Dimensions **210**

B CRS Reductions **213**

C Local - Global Theorem **217**

D Enhanced solution spaces **226**

E Paradigms of collaboration **230**

E.1 Parallel hybrids 230

E.1.1 An example of solvers in parallel 236

E.2 Concurrent hybrids 239

F Solver collaboration language **242**

G An example of many solvers in serial **247**

G.1 Case study 247

G.2 The solvers used 248

G.2.1 Expected behaviour of hybrid 249

G.3 Results 250

G.4 Conclusions 254

H Glossary **257**

List of Figures

2.1 A Racing Car in a Space with Obstacles 17
2.2 The Configuration Space Map for the Racing Car 18
2.3 The Product Design Process 19
2.4 A Hierarchy of Constraint Solvers 22
2.5 An Arm with Two Joints and the Relationship Graph for the Problem 27
2.6 An Example of Locus Analysis 29
2.7 A Pentagon Defined by Distance and Angle Constraints 31
2.8 The Constraint Graph for the Pentagon 31
2.9 Equation Graph for Constraints $C1$ to $C6$ 34
2.10 Tree-like Representation for Equation Graph 35
2.11 Typical result of Gaussian elimination on linearised constraint problem 36
2.12 Example of Incremental Insertion of Constraint. Arrows in Graph Indicate Order of Satisfaction 38
2.13 Example Constraint Graph for Hierarchical Constraint Problem . . . 39

3.1 Constraint problem P describing a pentagon 46
3.2 Decomposed subproblems of problem P 47
3.3 Recombined subproblems of constraint problem P 47
3.4 Constraint/Entity graph of figure 2.9 49
3.5 Decomposed subproblems of figure 3.4 50
3.6 The inverse operation method in IGCS (from [112]) 51
3.7 A connectivity graph for constraint problem P 52
3.8 Residual sets for constraint problem P 53
3.9 Graph of trade-off between complex decomposition and complex solvers 61
3.10 An rigid body composed of two triangles 64
3.11 An Arm with Two Joints and the Relationship Graph for the Problem 67

4.1 Placing a queen on a chessboard 78
4.2 A point, a line and a line segment on a plane 79

4.3 An equality constraint and a distance constraint 83
4.4 A solution to constraint problem G 85
4.5 A solution to constraint problem F 86

5.1 A Puma Robot Arm 101
5.2 A Hierarchy of Constraint Representation Schemes 102
5.3 An Example of a Relationship Graph with a Solution to the Graph . 106
5.4 An example of an undirected constraint graph with a solution to the graph. 107
5.5 An example of a constraint hypergraph with a solution to the graph. 108
5.6 Example of a Constraint/Entity Graph 109
5.7 Constraint/Entity Representation for Constraint Problem γ 113
5.8 New Construct for Constraint Edges 115
5.9 New Construct for Constraint Loops 115
5.10 Representing Quaternary Constraints in a Constraint/Entity Graph . 116
5.11 A Hierarchy of Constraint Representation Schemes 118

6.1 Solving the 4 queens problem 142

7.1 Two Blocks with an Against Constraint 152
7.2 A Chain of Blocks with Against Constraints 153
7.3 Two Rods 154
7.4 Problem G'' 155
7.5 A solution to constraint problem G of example 4.8 156
7.6 Solutions of constraint problem G'' with $l = 0, m = 8, n = 8$ 157
7.7 The Internal Combustion Engine 163
7.8 A Serial Hybrid of INCES and IGCS 164
7.9 Sequential Collaboration 166
7.10 INCES as a sequential hybrid 166
7.11 IGCS as a sequential hybrid 168
7.12 Two Rods 168
7.13 The 4 bar linkage problem 171
7.14 Case Study of n Piston Problems Linked Together 173
7.15 Serial Hybrid used to Solve n Piston Problems Linked Together . . . 173
7.16 A comparison of the C05NBC function and INCES algorithm with the hybrid solver 174

8.1 Constraint problem Q using distance and angle constraints 182

8.2 Constraint/Entity graph for constraint problem Q 182
8.3 Constraint problem R with three tangent circles 183
8.4 Constraint/Entity graph for constraint problem R 183
8.5 Decomposition of Constraint/Entity graph for problem R 185
8.6 A constraint problem with α, β and γ constraints 188
8.7 Two triangles with a common edge 193
8.8 Constraint graph for figure 8.7 193
8.9 Iteration two of incremental solution of figure 8.7 193
8.10 Iteration ten of incremental solution of figure 8.7 194

E.1 Parallel Collaboration 232
E.2 A simplification of the internal combustion engine 237
E.3 Concurrent Collaboration 240

G.1 Case Study of n Piston Problems Linked Together 248
G.2 Serial Hybrid used to Solve n Piston Problems Linked Together . . . 249
G.3 NAG function with bad initial guess 250
G.4 NAG function with good initial guess 251
G.5 Solving with INCES 252
G.6 Hybrid solution using IGCS and INCES 252
G.7 Solving the functional problem only using INCES without global parametric constraint list 253
G.8 Solving the geometric problem only using IGCS without the dependency hierarchy list 254
G.9 Solution using a hybrid algorithm of IGCS without depH list and INCES without global parametric constraint list 255
G.10 A comparison of the C05NBC function and INCES algorithm with the hybrid solver 256

List of Tables

2.1 Some Relations between Two Finite Domain Variables 11

3.1 Control scheme for solving constraint problems 43
3.2 Control scheme for solving constraint problems using domain specific knowledge and hybrid constraint solvers 44

5.1 Valid Constraint Representation Schemes 110

7.1 Two database tables (from [5]) 160
7.2 The result of joining the *book* and *sales* tables (from [5]) 160
7.3 Results for Solving ICE engine 50000 times on an SGI Indy 165
7.4 Solver collaboration language (adapted from BALI [84]) 171

8.1 Constraints that can be handled by Erep and IGCS 181

F.1 Solver collaboration language (adapted from BALI [84]) 242

Nomenclature

General nomenclature

$\mathbb{R}$	the set of real numbers,
$\mathbb{Q}$	the set of rational numbers,
$\mathbb{Z}$	the set of integer numbers,
$\mathbb{N}$	the set of natural numbers,
$\emptyset$	the empty set,
$\|A\|$	the number of elements in set A,
$(D_1 \times \cdots \times D_n)$	the Cartesian Product of the sets of $D_1, \ldots, D_n$,
$\cup$	the union of two sets,
$\cap$	the intersection of two sets,
$\setminus$	the set minus of two sets,
max	the larger of two numbers,
min	the smaller of two numbers,
$\in$	is a member of,
$\Leftrightarrow$	if and only if,
$A \neq B$	A is not equal to B
$A \subseteq B$	A is a subset or equal to B,
$\wedge$	logical and.

Constraint definition

D	a domain, a set,
E	an entity,
D_E	the domain of entity E,
v	a value for an entity, $v \in D_E$,

$E = v$	an assignation for entity E to the value v, so that $D_E = \{v\}$,
$E = \mathcal{S}$	an assignation of entity E to the set of values $\mathcal{S}$, so that $D_E = \mathcal{S}$,
aRb	a binary relation R between a and b, $R \subseteq D_a \times D_b$,
$S(a_1, \ldots, a_n)$	an n-ary relation S between $a_1, a_2, \ldots, a_n$, $S \subseteq D_{a_1} \times \cdots \times D_{a_n}$,
$f(x_1, \ldots, x_n)$	a boolean test function for a relation S, $f(x_1, \ldots, x_n) = 1 \Leftrightarrow (x_1, \ldots, x_n) \in S$,
CTP	Constraint test procedure, the boolean test function for a relation,
C	a constraint,
$f_C(x_1, \ldots, x_n)$	a constraint test procedure for a constraint C, $f_C(x_1, \ldots, x_n) = 1 \Leftrightarrow (x_1, \ldots, x_n) \in C$,
Φ	a set of entities,
Ψ	a set of constraints,
P	a constraint problem $P = (\Phi, \Psi)$,
P_i	a constraint subproblem (Φ_i, Ψ_i), $\Phi_i \subseteq \Phi, \Psi_i \subseteq \Psi$,
$\{x_1 = y_1, \ldots, x_n = y_n\}$	a configuration for a constraint problem $P = (\Phi = \{x_1, \ldots, x_n\}, \Psi)$,
$C\vert^E$	the enhanced constraint of C with respect to E, $C\vert^E = C \times D_{E_C}$, where D_{E_C} is the domain of those entities not relevant to C,
ξ_C	the imposed set of C,
$\vert\xi_C\vert$	the arity of C - the number of variables that affect C,
dim	a function from domains to the natural numbers,
e	usually an entity,
c	usually a constraint.

Constraint representation

CRS	a Constraint Representation Scheme,
α, β	constraint representation schemes,
A, B	problems in representation schemes,
Φ	a function from representation scheme to representation scheme, a representation,
$\alpha \equiv \beta$	representation scheme α is equivalent to representation scheme β,
γ	a Constraint/Entity graph,
$\mathcal{E}$	the set of entities in the algebraic representation,
$\mathcal{C}$	the set of constraints in the algebraic representation,
n, m	usually used to represent integer counters,
e	an edge in a graph.

Constraint satisfaction

$\boldsymbol{\mathcal{D}}$	a solution space, a set of configurations,
$\boldsymbol{\mathcal{D}}(0)$	the initial domain of the solution process, $(D_1 \times \cdots \times D_n)$,
k, i, j	an integer, usually representing an intermediary step,
Ψ_k	a subset of Ψ, usually used to represent intermediary constraints to consider,
$\boldsymbol{\mathcal{D}}(k)$	the domain of possible solutions at step k,
$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$	a solution step with respect to Ψ_k,
$\bigcap_{C \in \Psi} C$	the intersection of all C in Ψ,
Ψ'	subset of Ψ, usually representing a local set of constraints,
$\boldsymbol{\mathcal{C}}(\Psi')$	the set of solutions to all of the constraints in Ψ', equivalent to $\bigcap_{C \in \Psi} C$,
κ	an integer, representing a terminal step in a process,
$\boldsymbol{\mathcal{D}}(\kappa)$	final domain of the solution process, the terminal solution space,
$\boldsymbol{\mathcal{D}}(0) \xrightarrow{\Psi'}{}^{*} \boldsymbol{\mathcal{D}}(\kappa)$	a sequence of solution steps, a solution process,
$f_k(\Psi_k, \boldsymbol{\mathcal{D}}(k-1))$	a function, equivalent to a soluton step.

Constraint priorities

α_i	a constraint priority strength, a lower i is more important,
α_0	a 'required' constraint strength,
β_i	a constraint priority variable,
$\mathcal{H}_i$	a constraint hierarchy level,
$\mathcal{H}$	the set of all constraints in the constraint hierarchy,
better	a comparator between two solutions,
$\mathcal{S}_\prime$	the set of solutions to the required constraints,
$\mathcal{S}$	the set of solutions to the constraint hierarchy using the comparator *better*,
Υ	a set of constraint priority constraints.

Variable driven satisfaction

Φ_k	a set of entities,
$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Phi_k} \boldsymbol{\mathcal{D}}(k)$	a variable driven solutoin step.

Backtracking

r, s	integers, used to represent intermediate steps in a solution process,
$\boldsymbol{\mathcal{D}}(r) \xleftarrow{\Psi_r,\ldots,\Psi_{(r-s+1)}} \boldsymbol{\mathcal{D}}(r-s)$	backtracking by s steps.

Incremental solution

$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{C} \boldsymbol{\mathcal{D}}(k)$	an incremental solution step, adding a new constraint,
$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{E} \boldsymbol{\mathcal{D}}(k)$	an incremental solution step, adding a new entity.

Hybrid constraint solvers

S_i	a constraint solver,
$\boldsymbol{\mathcal{D}}_\Phi(i)$	the domain at step i of the entities in Φ,

$\boldsymbol{\mathcal{D}}_\Phi\|^{\Phi\cup\Phi'}(i)$	the enhanced domain of $\boldsymbol{\mathcal{D}}_\Phi(i)$ with respect to Φ',
$\boldsymbol{\mathcal{D}}_\Phi\|_{\Phi'}(i)$	the embedded domain of Φ' in Φ at step i,
$(geom, (x, y, z))$	a geometric variable $geom$ which has x rotational degrees of freedom, y scalar degrees of freedom and z translational degrees of freedom,
P_{geom}	a geometric subproblem,
P_{alg}	an algebraic subproblem,
f	a function for combining solution spaces found in a parallel collaboration,
ψ	a choice function for selecting solutions spaces in a concurrent collaboration.

A constraint management system

$(P_i, \mathcal{S}_i)$	A subproblem-solver pair. P_i is a subproblem, $\mathcal{S}_i$ is a set of constraint solvers.
$\triangle ABC$	A triangle formed from three vertices, A, B and C.
$A, B, C, D, E, F, G, H, I, J$	Typically entities in a constraint problem. Sometimes residual sets.
$<$	A partial order.
$(P_i, \mathcal{S}_i); (P_j, \mathcal{S}_j)$	Sequential collaboration.
$(P_i, \mathcal{S}_i) \parallel (P_j, \mathcal{S}_j)$	Parallel collaboration.
De	A decomposition strategy. Decomposes a constraint problem and a set of solvers into a set of subproblem-solver pairs.
$De_{strong\ components}$	Decompose a constraint problem into subproblems that are strongly connected and subproblems that are not strongly connected.

$De_{connected\ components}$	Decompose a constraint problem into subproblems that are connected.
Cl	Typically a cluster of constraints and entities describing a rigid body.
s	Typically a constraint solver.
$s(P_i) : \boldsymbol{\mathcal{D}} \longrightarrow \boldsymbol{\mathcal{D}}(i)$	Initiating constraint solver s on constraint problem P_i using initial solution space $\boldsymbol{\mathcal{D}}$ results in solution space $\boldsymbol{\mathcal{D}}(i)$.
$y \rightsquigarrow z$	A path from y to z in a graph.
T	A type of constraint.
α, β, γ	Types of constraint capable of being solved by Erep or IGCS.
Q, R	Typically constraint problems.
C_1, C_2, C_3	Typically constraints.
$Circ_1$	A circle.

Glossary of Solvers

Solver	Reference	Description
Concept Modeler	[100]	Solves triangular systems of equations using local propagation. Solves simultaneous subproblems using numerical techniques.
Connectivity Analysis	[67]	Solves geometric constraint problems by splitting problems into residual sets and then solving residual sets in order.
DCM	[86]	Solves well-constrained geometric problems by identifying subproblems consisting of only three objects that can be fixed.
Erep	[14]	Very similar to DCM. Solves well-constrained geometric problems by building up clusters of objects that can be relatively fixed and combining clusters to give rigid bodies.
GCE	[59]	Geometric constraint solver using locus analysis to solve simultaneous subproblems and action analysis to solve simple subproblems by local propagation.
Gröbner bases	[16]	Symbolic algebraic solver acting on systems of equations.
ICBSM	[27]	Geometric constraint solver using local propagation to solve 3D geometric problems without loops using local propagation.

Solver	Reference	Description
IGCS	[112]	Improvement of ICBSM to handle loops and more complex problems.
INCES	[62]	Incremenal version of Concept Modeler. Allows geometric constraint problems to be described and solved.
MechEdit	[15]	Geometric constraint system for solving systems of planar linkages. Uses local propagation and numerical techniques.
Newton-Raphson	[70]	Numerical solver that takes a system of equations and finds solution using iterative techniques.
SkyBlue	[94]	Solves hierarchical constraint problems using local propagation. Can solve for constraints added incrementally.

Chapter 1

Introduction

Constraints have become popular in the engineering design field as a means of designing and building complex product models [3]. The power of constraints as a modelling paradigm lies in their descriptive power and the implicit knowledge that lies within a constraint's description. A constraint can vary from a nonlinear equation to a complex description of the assembly of a geometric model. Consequently, constraints are used in a number of different contexts and solutions to constraint problems are found using many different techniques, depending on the nature of the constraints involved.

A system of nonlinear equations for example, with no discernible structure, will typically be solved using numerical [70] or symbolic techniques [16, 56]. *General* constraint solvers such as numerical or symbolic techniques are slow but can find solutions for a wide variety of constraint problem.

On the other hand, a geometric constraint problem, consisting of relative positioning instructions for lines, points and circles, can be solved relatively efficiently using degrees of freedom analysis [27, 59] or ruler-and-compass construction techniques [14, 86]. The geometric constraint solver takes advantage of *domain specific knowledge* of the geometric problem domain. In the case of degrees of freedom analysis, the domain specific knowledge is that the geometric objects - the lines, points and circles - can only translate and rotate in space in a very limited number of ways. Degrees of freedom analysis uses this knowledge to manipulate the rigid bodies to satisfy the constraints applied to them. In the case of ruler-and-compass construction, the domain specific knowledge used is that all geometric constraints can be described using distance and angle constraints and that using such constraints, any unknown object can be positioned relative to two known objects using two constraints. Ruler-and-compass solvers then position the geometric objects one by one

until all of the constraints have been satisfied. Constraint solvers that take advantage of knowledge implicit in their domain are referred to as *domain specific* in this thesis.

Domain specific constraint solvers include finite domain solvers, scheduling constraint solvers, functional constraint solvers and physical constraint solvers. Finite domain constraint solvers take advantage of the finite number of possible solutions by using sophisticated exhaustive search techniques. Scheduling constraint solvers use the linear nature of time to help simplify the search for solutions. Functional constraint solvers simplify systems of equations to identify subproblems that are simple to solve. Physical constraint solvers take advantage of inertia, momentum and other physical laws to simplify the calculation of new positions, reusing old information as much as possible. For reasons of brevity, only finite domain, geometric and functional constraint solvers are studied in detail in this thesis.

It is generally acknowledged that there is a close coupling between *functional constraints* and *geometric constraints* [3, 19, 64, 100] in the design process, particularly in the early stages of conceptual design. A constraint solver for engineering design must therefore be capable of handling both functional *and* geometric constraints simultaneously. Similarly, finite domain, scheduling and physical constraints are integral parts of the design process and should be considered simultaneously also.

The current state of the art is that domain specific constraint solvers will always outperform general solvers for problems within their domain. Not surprisingly, domain specific solvers are inappropriate for problems outside this domain.

Some authors attempt to overcome the expressive limitations of domain specific solvers by using a general solver as a backup for when the domain specific solver cannot find a solution [15, 62, 85, 87, 112]. Similar to the terminology adopted by Prosser [91], solvers which use multiple solution techniques interacting with each other are referred to as *hybrid constraint solvers* in this thesis.

There has been a long-standing research project at the University of Leeds to use constraints to build complex engineering models within a Virtual Environment [29–32, 62, 78, 112]. Three main constraint solvers have been developed for this project:

- ICBSM [31, 32] was developed by Fa *et al.* to build geometric constructs of engineering models. The key innovations of ICBSM were the Allowable Motion method of solving the constraint problem and the Automatic Constraint Recognition method for using direct manipulation to build the model. Allowable Motion calculates the possible movements of geometric objects in the

model depending on the constraints placed on the objects. Allowable Motion is described in more detail in section 2.3.3.1 and [27]. Automatic Constraint Recognition utilised the user's manipulation of objects within the model to suggest possible constraints that could be applied. This simplified the construction of the geometric model.

- In ICBSM, all geometric objects had to be applied one after the other in a sequential fashion. ICBSM could not cope with the simultaneous definition of geometric constraints. Consequently, IGCS [112] was designed to allow simultaneous constraints to be solved.
- ICBSM could only deal with geometric constraints, such as forcing two objects to remain in contact with each other, or to have two cylinders concentric. INCES [62] was developed in order to solve systems of equations describing the *function* of an engineering model rather than the physical *form*. INCES could also solve the geometric constraints describing the engineering model by converting them to equations. INCES cannot solve all such systems of equations and so resorts to a numerical algorithm when it cannot succeed on its own.

The purpose of this thesis was to investigate existing constraint solvers and to see if Fa's Allowable Motion approach could be applied to more general problems. In particular, it was realised that, as it stood, Allowable Motion could not be applied to loops and functional problems. Since these are integral parts of engineering design constraint problems, it was important that they be integrated into any constraint solver that would handle engineering design constraint problems.

A glossary is included in this thesis that can be used for reference. Words denoted by a† indicate definitions that can be found in the glossary. The glossary is included in appendix H.

1.1 Objectives of this thesis

A broad objective of the Virtual Working Environment group in the University of Leeds is to develop an interactive environment for developing engineering designs within a Virtual Environment. The interactive constraint modelling subgroup is mainly involved with developing algorithms and prototypes that use constraints as a paradigm for describing and finding solutions to engineering designs. Previously, the constraint modelling subgroup developed ICBSM, IGCS and INCES and these

algorithms are capable of describing and solving a variety of constraint problems. The objectives of the research in this thesis were to:

- Investigate ways of generalising the techniques developed in ICBSM, IGCS and INCES in such a way that more problems can be solved at little or no loss of speed.
- Study the state of the art in constraint solvers and identify commonalities amongst them that can be utilised.
- Create an interactive constraint solver capable of dealing with the needs of engineers using available technology.

Progress on these research goals are reported in this thesis:

- Current engineering design constraint solvers have been studied and their strengths and weaknesses identified. This allows the creation of a taxonomy of constraint solvers by their strengths. The use of domain specific knowledge to improve the efficiency of these constraint solvers has been explored. Domain specific knowledge has been identified as an important means of improving the efficiency of constraint solvers.
- The characterisation of current constraint solvers has led to an identification of the common processes adopted by these constraint solvers. The divide-and-conquer strategy adopted by virtually all current constraint solvers uses a decomposition strategy to identify subproblems to solve; an ordering strategy to decide in which order to solve the subproblems; and a set of solution techniques to solve the subproblems. The use of this divide-and-conquer strategy leads to a number of issues and questions that need to be addressed.
- In order to address these issues, a mathematical description of the constraint process has been created which is capable of describing general engineering design constraint problems, and the representation and solution of engineering constraint problems. The mathematical framework allows the description of most current constraint solvers and consequently provides a unifying framework for the constraint solvers. The framework also captures the concepts of consistency, soundness and completeness which describe the quality of solution of the constraint solvers. A theorem has been devised that allows the quality of solution of constraint solvers to be deduced from the workings of the constraint solver.

- Using the mathematical characterisation of the constraint problem and constraint solution, a study of hybrid collaborations has been carried out. The use of hybrid collaborations, such as sequential, parallel and concurrent, allows existing solvers to be joined together in a formal way that allows new solvers to be created. The mathematical framework developed allows concrete statements to be made about the nature of these solvers.

- A new constraint solver has been defined that consists of a hybrid of Erep and IGCS. This constraint solver combines the ability of Erep to solve well-constrained geometric constraint problems with loops and the ability of IGCS to solve under-constrained geometric constraint problems.

1.2 Incremental constraint solvers

Constraint solvers take a set of constraints and variables and find a number of solutions to the constraint problem defined by the constraints and variables. Most constraint solvers solve a constraint problem from scratch. That is the whole problem is defined before solutions to the problem are found.

However, this is not how designs are created. Most designs evolve over a period of time as designers add new objects or new constraints. The *specify-then-solve* approach of existing constraint solvers means that every time a new constraint is added, the whole constraint problem needs to be resolved.

To counter this problem, *incremental constraint solvers* have been developed. Incremental constraint solvers assume that a constraint problem will evolve over time and that at every iteration of the design process, a set of variables and constraints is added to the existing constraint problem and new solutions to the constraint problem need to be found.

The challenge for incremental solvers is to find solutions to the new problem as quickly as possible. Most current incremental constraint solvers try to do this by reusing the information found by solving the previous constraint problem.

For example, assume that a constraint problem P_1 has been solved for and a set of solutions S_1 has been found. Then a set of new constraints and variables is added to P_1 to give P_2. Solutions must now be found for P_2 as quickly as possible.

There are several sources of information that can be used to assist incremental constraint solution. The number, type and quality of solutions in S_1 can help to indicate solutions to P_2. For example, if P_1 has no solutions and S_1 is empty, then

adding more entities and constraints cannot increase the number of solutions and so P_2 will have no solutions either.

The most useful source of information to assist incremental solution is the process whereby P_1 was solved. If P_2 is a similar problem to P_1 then it will probably be solved in a similar fashion. Consequently, it may be possible to reuse some of the information used to solve P_1 in order to solve P_2.

For example, suppose P_1 was solved by splitting it into a number of subproblems that were solved in the order:

$$P'_1, P'_2, P'_3, P'_4.$$

Suppose also that the new constraints and variables are added to subproblems P'_3. Assuming that P_2 can be solved in the same way as P_1, then it can probably be split into a similar set of subproblems

$$P'_1, P'_2, P''_3, P''_4, P''_5.$$

In particular, P'_1 and P'_2 are the same because they are not affected by the new constraints and variables. Thus P'_1 and P'_2 do not need to be resolved, saving time and effort.

Incremental solvers such as INCES [62] and SkyBlue [94] try to take advantage of this structure and so speed up solution of each iteration of the design process. IGCS [112] tries to solve for a new constraint by manipulating objects whilst maintaining old constraints. Thus, when a new constraint is added, only that constraints needs to have solutions found for as all other constraints have already been solved for.

In general, the aim of a specify-then-solve constraint solver is to be able to solve a constraint problem in time $\mathrm{O}(n)$, where n is the number of constraints. An incremental constraint solver aims to solve for a new constraint in time $\mathrm{O}(1)$.

A constraint problem is *well-constrained* if it has as many constraints as variables. A well-constrained problem usually has a finite number of solutions. A constraint problem is *under-constrained* if it has more variables than constraints. An under-constrained problem usually has an infinite number of solutions. A constraint problem is *over-constrained* if it has more constraints than variables. An over-constrained problem usually has no solutions. A more precise definition of these terms is left to later in this thesis.

A constraint problem usually progresses from being under-constrained to being well-constrained to being over-constrained. Initially, a constraint problem has no

variables and no constraints. The user will then add a number of variables. At this point, the problem is under-constrained as no constraints have been created. Gradually, the user will add more and more constraints to the problem until it becomes well-constrained. The user may then add more constraints and the problem becomes over-constrained.

Thus an incremental solver that allows the user to add constraints and variables one at a time must be able to deal with under-, well- and over-constrained problems. However, few constraint solvers can do this. Most constraint solvers can only solve problems that are one of under-, well- and over-constrained. ICBSM [27], for example, cannot cope with cycles in constraint graphs. In fact, cycles correspond to well-constrained subproblems and ICBSM is particularly well suited to under-constrained problems.

Therefore, a good incremental constraint solver will be one that reuses previous information well and also can handle under-, well- and over-constrained problems. Since incremental solvers are an important means of solving constraint problems quickly, and therefore in an interactive environment, they form a significant part of this thesis.

1.3 Thesis organisation

This thesis is divided into ten chapters including this introduction. Chapter 2 presents a detailed study of current constraint solvers with an aim of categorising them and understanding their underlying principles. Chapter 2 also discusses subsidiary research on set theory, dimensions and some basic relational algebra.

Chapter 3 takes the characterisation of the constraint solvers in chapter 2 and identifies the common features of these constraint solvers. In particular, this chapter notes that most existing constraint solvers use a divide-and-conquer approach, whereby a decomposition strategy is used to identify subproblems; an ordering strategy is used to determine in which order subproblems are to be solved; and a set of solution techniques find solutions to the subproblems. A detailed study of each strategy leads to a number of issues that need to be addressed. In order to study these issues thoroughly, it is first necessary to understand the constraint solution process in detail. In particular the incremental issues associated with each strategy are identified.

Consequently, chapters 4, 5 and 6 formalise the constraint process with an aim of capturing the various properties of constraint solvers. Chapter 4 defines the

constraint problem in terms of its constituent parts. Chapter 4 also discusses the notion of the *dimension* of a set and uses this to describe certain types of constraint problem. Chapter 5 discusses the problems of representing a constraint problem on a computer. The various different representation schemes currently used are compared and a representation scheme capable of describing general constraint problems is identified.

Chapter 6 presents an abstraction of the constraint satisfaction process, whereby constraint solvers take as input a constraint problem and produce as output a set of solutions. The importance of properties such as *consistency*, *completeness* and *soundness* of constraint solvers is highlighted and a theorem is presented that allows statements to be made about constraint solvers' properties depending on individual steps. In order to prove the flexibility of the constraint satisfaction abstraction, it is used to describe several advanced constraint solution techniques such as *backtracking*, *constraint priorities* and *incremental* techniques.

The constraint process abstraction built up over chapters 4, 5 and 6 is then used in chapter 7 to discuss the use of domain specific knowledge in constraint solvers and also the use of more than one constraint solver to build up *hybrid* systems. The various paradigms available for joining solvers together are introduced in terms of the satisfaction framework and demonstrated using examples. A hybrid collaboration language is used to describe the interaction of constraint solvers in a hybrid.

Chapter 8 uses the hybrid collaborations identified in chapter 7 to build a new hybrid constraint solver consisting of IGCS and Erep. Since IGCS can solve under-constrained geometric constraint problems and Erep can solve well-constrained geometric constraint problems with loops, the hybrid should be able to solve constraint problems consisting of well- and under-constrained subproblems. Correspondingly, the power of IGCS has been increased at little computational cost and a constraint solver more appropriate for engineering design has been created.

Chapter 9 discusses future work to realise the goal of an interactive constraint based system for engineering design. Chapter 10 presents conclusions from the work in this thesis.

Chapter 2

Related Work

This thesis investigates the use of hybrid constraint solvers using domain specific knowledge using case studies in engineering design. This chapter describes the state-of-the-art in constraint-based design, hybrids and domain specific constraint solvers. A number of constraint solvers have been studied so that an abstract framework for constraint solution can be developed which can then be used to study hybrid constraint solvers.

The abstraction of the constraint satisfaction process leads to the study of dimensions, decomposition of constraint problems, the study of solution spaces and the use of hybrids to solve constraint problems. Work related to these topics is covered in section 2.1.

Section 2.2 discusses the state-of-the-art in constraint solution for engineering design. Section 2.3 describes a number of constraint solvers. The solvers are categorised and discussed in terms of figure 2.4. General constraint solvers consist of algebraic or numerical techniques and are introduced in section 2.3.1.

Finite domain constraint solvers are a well-understood type of domain specific solver where objects in constraint problems have only a finite size. Although not currently used much in engineering design, finite domain solution techniques have a large body of literature and form a useful basis for discussing constraint satisfaction. Finite domain solvers are described in section 2.3.2.

Geometric constraint solvers are used in engineering design to capture the designer's intent. For example, if a designer draws a line that happens to be vertical, then it is likely that the designer intended that line to be vertical no matter what else happened to the model. Consequently, the designer's intent, to have a vertical line, is retained through the use of constraints. The use of geometric constraints is becoming more widespread and they are now available in commercial CAD packages

such as Unigraphics [117] and Pro/ENGINEER [20]. Geometric constraint solution is discussed in section 2.3.3.

Functional constraint solvers are also becoming popular in engineering design. Functional constraint solvers are used to describe the function of an engineering design in terms of algebraic equations. These are discussed in section 2.3.4.

Constraints are also being used in maintenance and physical simulation. This is an exciting new development for the use of constraints and is still a relatively new development. The state-of-the-art is covered in section 2.3.5.

Section 2.4 presents conclusions from this chapter.

2.1 The theory of constraints

A large number of problems in artificial intelligence and computer science can be described as constraint satisfaction problems. As such, many researchers have investigated efficient methods of solving constraint satisfaction problems and a large body of literature exists. Kumar [60], Dohmen [22] and Meseguer [81] present survey papers on constraint satisfaction algorithms and Jaffar and Maher [52] discuss constraint logic programming - the merger between constraint satisfaction and logic programming.

A constraint problem consists of a set of objects and a set of restrictions on the values the objects can take. A constraint satisfaction algorithm attempts to find solutions to the constraint problem using a number of heuristic techniques. A simplification of constraint satisfaction is that of relational algebra [5]. In relational algebra, a finite number of objects with a finite number of relations on them are queried and the result is a set of solutions to the query.

Finite domain constraint satisfaction [103,114] is equivalent to relational algebra although the relations used are more complex. For example, a relation between two finite domain objects that can each take n values is represented by the $n \times n$ table of values that are allowed. Table 2.1 demonstrates the definition of the standard relations $x = y$ and $x \neq y$, where x and y both have domains $\{0, 1, 2\}$. A 1 in position $x = a, y = b$ means that $x = a, y = b$ is allowed in the relation. A 0 in the same position means that $x = a, y = b$ is not permitted. For example, for the relation $x = y$, when $x = 1, y = 2$, the value in the table is 0 and so this configuration is not allowed. The other two relations defined, $x \bullet y$ and $x \circ y$, indicate relations that cannot be simply described using relational algebra.

The general concept of constraint problems allows objects to have infinite do-

$x = y$		x		
		0	1	2
	0	1	0	0
y	1	0	1	0
	2	0	0	1

$x \neq y$		x		
		0	1	2
	0	0	1	1
y	1	1	0	1
	2	1	1	0

$x \bullet y$		x		
		0	1	2
	0	1	1	1
y	1	1	1	0
	2	1	1	1

$x \circ y$		x		
		0	1	2
	0	0	0	1
y	1	1	0	1
	2	0	1	0

Table 2.1: Some Relations between Two Finite Domain Variables

mains. Likewise, constraints can be described using an infinite number of tuples and are also called infinite domain. Such a definition allows many more difficult problems to be described using constraints, such as geometric problems [58, 86, 112], algebraic problems [62, 98] and scheduling problems [124]. However the use of infinite domain constraints negates the use of the search algorithms prevalent in finite domain constraint satisfaction become impractical for infinite domain constraint satisfaction.

The use of techniques other than search form an important area of research for infinite domain constraint problems. Solvers that tackle infinite domain problems are discussed in more detail later in this chapter.

Some authors have compromised between finite domain and infinite domain constraint satisfaction by using integer constraints. For example Jaffar *et al.* [51] discuss the use of integer constraints and suggest that the restricted class of integer constraint they use "can be solved more efficiently than in the general case but which remains reasonably expressive".

Other authors have generalised constraints to allow more general structures. Bistarelli *et al.* [10] describe constraint satisfaction problems in terms of semiring based structures. The advantage of such a generalisation is that it describes not only the type of constraint satisfaction problem discussed above, but also fuzzy problems, weighted problems and partial constraint satisfaction. The semiring formalisation is elegant but somewhat cumbersome for the purposes of this thesis.

In order to combat problems that have too many constraints, Borning *et al.* [11] introduce constraint hierarchies which allow the creation of constraint problems

where not all of the constraints need to be satisfied. Constraint hierarchies have become an important weapon in a constraint programmer's arsenal and as such are used in many different applications such as SkyBlue [94], Differential Manipulation [43] and Multi-Garnet [95]. Constraint hierarchies are described in more detail in section 2.3.4.1.

The common link between all of these theories is that a constraint problem is a set of objects with domains and a set of constraints which are relations. Many constraint satisfaction techniques take advantage of the size of the domains of the objects or of decomposing the constraint problem. Consequently, the *dimension* of a domain is discussed in section 2.1.1 and work using the decomposition of constraint problems is discussed in section 2.1.2. There exist a number of constraint satisfaction algorithms that find solutions by using *hybrid solvers*. These are investigated in section 2.1.3. The concept of a *solution space* is also significant in this thesis and related work on configuration spaces is presented in section 2.1.4.

2.1.1 Dimensions

Researchers such as Latham and Middleditch [67], Fa *et al.* [26, 27, 31, 32], Tsai *et al.* [112] and Kramer [57–59] use the dimension of an object as a measurement of progress towards solution. Fa *et al.*, Tsai *et al.* and Kramer also use the dimension of geometric objects to provide efficient interactive constraint satisfaction. The notion of dimension is also associated closely with the constrainedness of a problem [67]. The dimension of an object is a measure of the freedom of an object in terms of the number of possible values or positions that can be assigned to the object.

For example, a 0-dimensional object only has a finite number of possible values or positions, whereas a 1-dimensional object can be assigned values from some subset of the real line.

Dimensions for simple geometric objects and constraints are straightforward and well-understood. However, the generalisation of dimension to general constraints and objects has received little attention. Latham and Middleditch assume that constraints and objects have a dimension but note that it is very difficult to provide a formal definition of dimensions for general sets due to the existence of Peano curves and other space-filling curves.

For example, the dimension of a point moving in three dimensional space, $\mathbb{R}^3$, is 3, as the point needs three parameters to fully define it. Similarly, the dimension of a variable that can take any value in the set of integers, $\mathbb{Z}$, is 1, as the variable

needs one parameter to fully define it. However, the dimension of a variable that can take any value in the set $\mathbb{Z}^2$ is more difficult to define. Space filling curves, such as the Peano curve, mean that it is possible to define a mapping from $\mathbb{Z}^2$ to $\mathbb{Z}$ such that, for all intents and purposes, they are the same set. Consequently, $\mathbb{Z}^2$ could have dimension 1 or dimension 2. Such questions need to be dealt with to have a common definition of the dimension function.

For the purposes of this thesis, the identification of the constrainedness of constraint problems is very important. The constrainedness of a constraint problem is related to the number of solutions that the constraint problem has. Simply put, a constraint problem is *well-constrained* if the constraint problem has only a finite number of solutions. A constraint problem is *under-constrained* if it has an infinite number of solutions and is *over-constrained* if it has no solutions.

An investigation has been made into the definition of a dimension function and *manifolds* have been identified as fulfilling the most important properties of the dimension function. Since the *use* of the dimension function is more important than the precise *definition*, the definition used in this thesis is given in full in appendix A. The reader is referred to [107] for a more detailed discussion of manifolds.

2.1.2 Decomposition of constraint problems

Many constraint solvers attempt to solve constraint problems by decomposing a large constraint problem into a number of smaller subproblems and solving them separately. In practice, this is a variant of the Divide-and-Conquer strategy employed in computer programming and is very effective. The decomposition allows the identification of subproblems that can be solved quickly and easily. The results of solving these subproblems can then have effects for the rest of the problem, simplifying other subproblems. Decomposition of constraint problems forms an important part of the research in this thesis. In particular, decomposing a complex constraint problem into a number of subproblems that can be solved simply using different solution techniques allows flexible and efficient solution.

Solvers such as D-Cubed [86], Erep [39], Connectivity Analysis [67], INCES [62] and IGCS [112] use decomposition to subproblems to aid solution. For example, Erep identifies *clusters* of geometric objects that can be defined relative to each other. The clusters are then combined recursively to give solutions to the whole problem. INCES identifies subproblems that cannot be solved using local propagation and then uses numerical solution to solve them.

For finite domain constraint satisfaction problems, Freuder and Hubbe [36] have presented an algorithm for extracting a particular subproblem from a constraint problem. This algorithm can be used to extract a subproblem that is known to be unsolvable and discard it, restricting the search for a solution to the remaining subproblems.

Latham and Middleditch [66, 67] present an algorithm called Connectivity Analysis that will decompose a constraint problem into a number of *well-balanced sets*. The well-balanced sets correspond to well-constrained subproblems. This allows the identification of under- and over-constrained subproblems as well as identifying precisely which objects need to be more or less constrained by adding or subtracting constraints respectively.

Connectivity Analysis also provides an ordering for the well-balanced sets so that the constraint satisfaction process can use information from previously solved well-balanced sets to aid solution of other well-balanced sets.

2.1.3 Hybrid constraint solvers

The decomposition of constraint problems means that a problem P is divided into a number of subproblems $P_1, \ldots, P_n$ which are solved using the same algorithm. The logical extension of this paradigm is to still decompose P into $P_1, \ldots, P_n$ but to solve the various P_i using different solution algorithms. This technique is variously called cooperation, combination or hybrid constraint solution. In this thesis, the use of multiple solvers to solve subproblems will be called *hybrid constraint solution*.

The use of multiple constraint solvers dates back to the introduction of constraints in Sketchpad [108] in 1963. There local propagation was used until it could proceed no further and then the problem was 'relaxed' using numerical techniques. Similar techniques followed until Prosser [91] suggested the use of hybrid algorithms for finite domain constraint satisfaction.

At that point, there existed several constraint satisfaction algorithms in use in the finite domain field. A collection of algorithms looked at the future effect of an action to investigate possible failure. Another collection of algorithms looked at past information to backtrack from current failure. Prosser presented algorithms that used both types of search. These hybrid algorithms proved very successful and were both more efficient [116] and less prone to extraordinary failure [105] than non-hybrid solvers.

Although Prosser's hybrid algorithms do not decompose constraint problems, the

concept of using multiple, co-operating constraint solvers to help solve the constraint problems can be usefully applied to other types of constraint problem.

The theory of hybrids has been examined by Baader and Schulz [6, 7], Monfroy [84], Monfroy *et al.* [83, 85] and Kirchner and Ringeisson [55]. Baader and Schulz discuss the combination of constraint solvers using a highly complex study of unification theory. The principal theories of Baader and Schulz's work are the notion of a free amalgamated product and a decomposition algorithm. The free amalgamated product is used to define a combined solution structure over which constraint problems are solved. The decomposition algorithm separates a problem into subproblems that can be solved by the individual solvers. These problems are presented in a different and more approachable form in this thesis.

Monfroy introduced BALI in [84]. BALI is a semantic definition of an environment for solver cooperation using the three paradigms of sequential, parallel and concurrent collaboration. BALI is a useful environment for describing solver collaboration but it tries to solve the whole problem using one solver, only resorting to other solvers on failure. BALI does not take into account the strengths of individual solvers and does not subdivide a constraint problem so that subproblems are solved using appropriate subsolvers.

BALI has been used to combine a Gröbner basis solver with a linear equation solver to produce COSAC [85]. COSAC tries to use the more efficient linear equation solver as much as possible and only resorts to the Gröbner basis solver as a last resort. COSAC is an improvement on Gröbner bases alone but is heavily dependent on the Gröbner basis solver which is necessarily slow. Monfroy and Ringeisson also propose a method of extending the scope of constraint solvers to process new types of constraint [82].

The hybrid constraint solver presented in this thesis uses constraint solvers to best effect by identifying the strengths of a constraint solver and then decomposing the constraint problem in such a way that solvers are used on problems that they are best suited to. Here we assume that the subdivision can be done sufficiently quickly that the efficiency of the constraint solver is still dominated by the efficiency of the slowest subsolver. This is discussed in more detail in chapter 3.

2.1.4 Solution spaces

Finite domain satisfaction techniques typically search through all of the possible combinations of object values in order to find combinations that are solutions. For

finite domain constraint problems, the number of possible combinations is large but finite. In infinite domain constraint problems, the number of possible combinations is typically infinite. However, the concept of a *solution space* - the set of possible combinations at a given time - forms a useful abstraction for studying constraint satisfaction processes.

The notion of solution spaces is equivalent to configuration spaces used in path planning. Lozano-Pérez [72] uses configuration spaces to help calculate constraints on the position of an object in space due to other objects. This allows the computer to arrange objects in space or to move objects without collisions. Problems are reduced from planning a path for a complex object to planning a path for a point and so are much simplified, though calculation of the configuration space is very time-consuming.

Wise [121] has built on Lozano-Pérez's work and used it to build svLis-m [1], an algorithm that builds multi-dimensional configuration spaces representing an object in every conceivable position and orientation. Once the configuration space has been created, it is simple to move objects around in the space detecting collisions. For example, the problem in figure 2.1 is to move a racing car in the space without colliding with any obstacles. The car can translate in the x and y directions.

The configuration space calculated for the car and obstacles is presented in figure 2.2. The current position of the car is described using a reference point on the car and compared with the configuration space map. If the (x, y) point indicating the position of the car is dark grey in the configuration space map then the car is inside an obstacle which is not allowed. If the point corresponds to a dark grey pixel, then the car is not in contact with any obstacle. If the point is a light grey pixel, then the car is touching an obstacle. Thus the problem of checking for a collision has been reduced to a very simple point membership test. Unfortunately creating the configuration space map in the first place took some 15 minutes. This example was taken from `http://www.bath.ac.uk/~ensab/G_mod/Svm/Html_ver/svm_home.html`.

2.2 Constraints in engineering design

Engineering design is currently undergoing a paradigm shift [47]. Classical design as espoused by Pahl and Beitz [88] typically involves an iterative model, whereby specifications and information are passed from one stage to another (figure 2.3, taken from [64]). At each stage a large quantity of information needs to be transferred from

[1] `http://www.bath.ac.uk/~ensab/G_mod/Svm/Html_ver/svm_home.html`

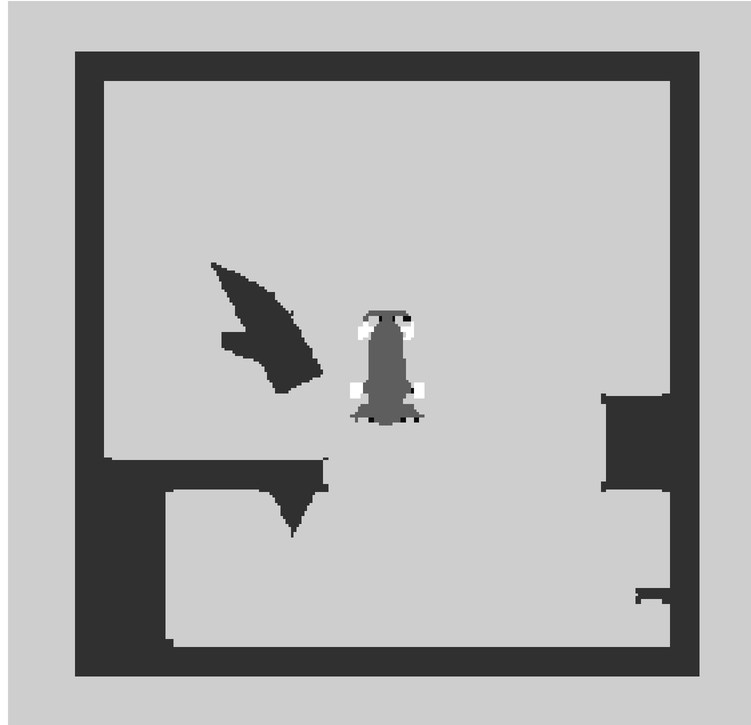


Figure 2.1: A Racing Car in a Space with Obstacles

one stage to the next, typically in the form of blueprints or design specifications. The potential for loss of design intent and information is very great.

Modern CAD packages are in common use by designers. Many designers use advanced design support tools such as knowledge based engineering, 3D CAD and finite element analysis. These support packages help to mitigate the loss of information by having consistent data structures and persistent and concurrent design development. However, they do not explicitly retain the designer's intent.

One paradigm shift identified by Hoffmann and Rossignac [47] is a move from the advanced support tools of CAD packages within a traditional design methodology to the use of constraint-based design paradigms [47]. Constraint-based design allows the design intent of a designer to be captured as constraint-based design is oriented more towards the design process. As a design is modified, the constraint solvers attempt to consistently maintain the designer's intent.

Anderl and Mendgen [3] discuss the use of constraints in modelling. They identify the importance of geometric constraints and engineering constraints [2]in the modelling process and discuss the definition, representation and solution of con-

[2]Engineering and functional constraints are used interchangeably in this thesis. There is no consensus in the literature as to which is the correct terminology.

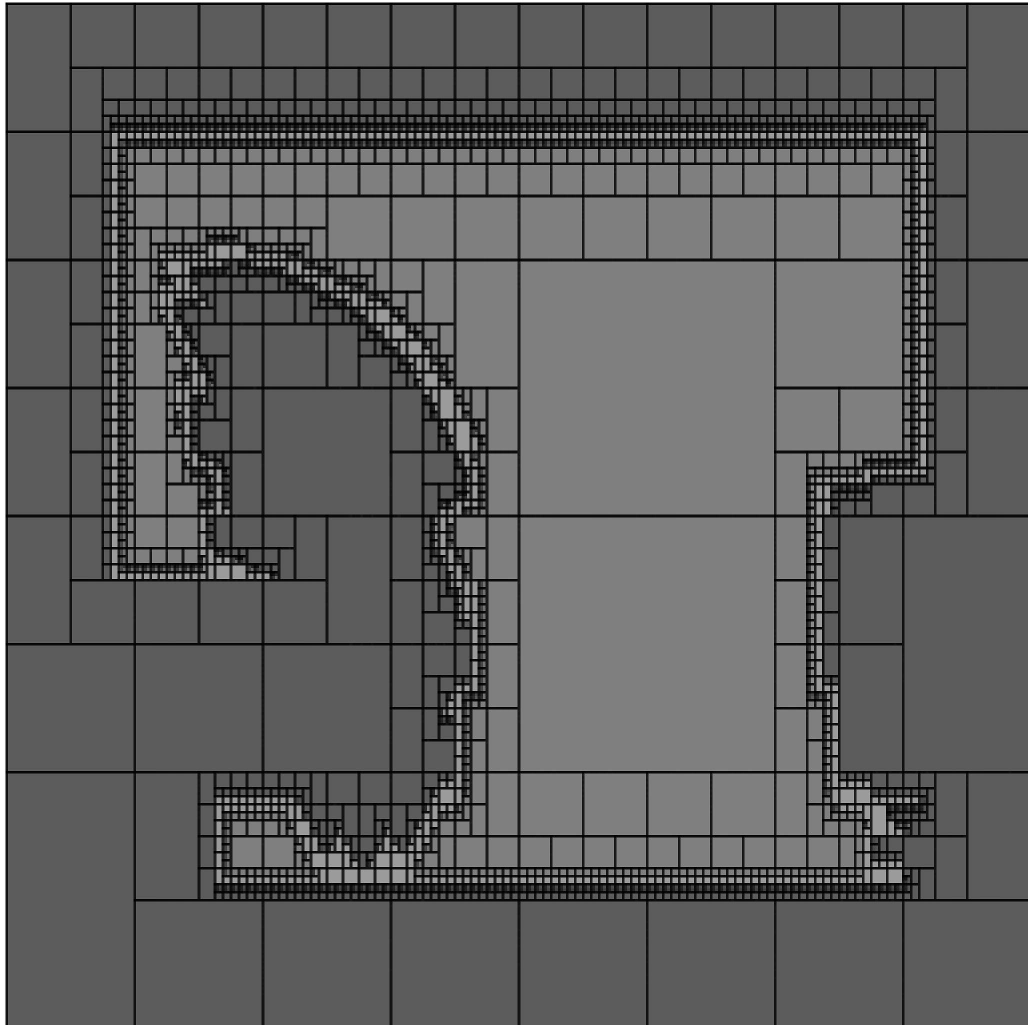


Figure 2.2: The Configuration Space Map for the Racing Car

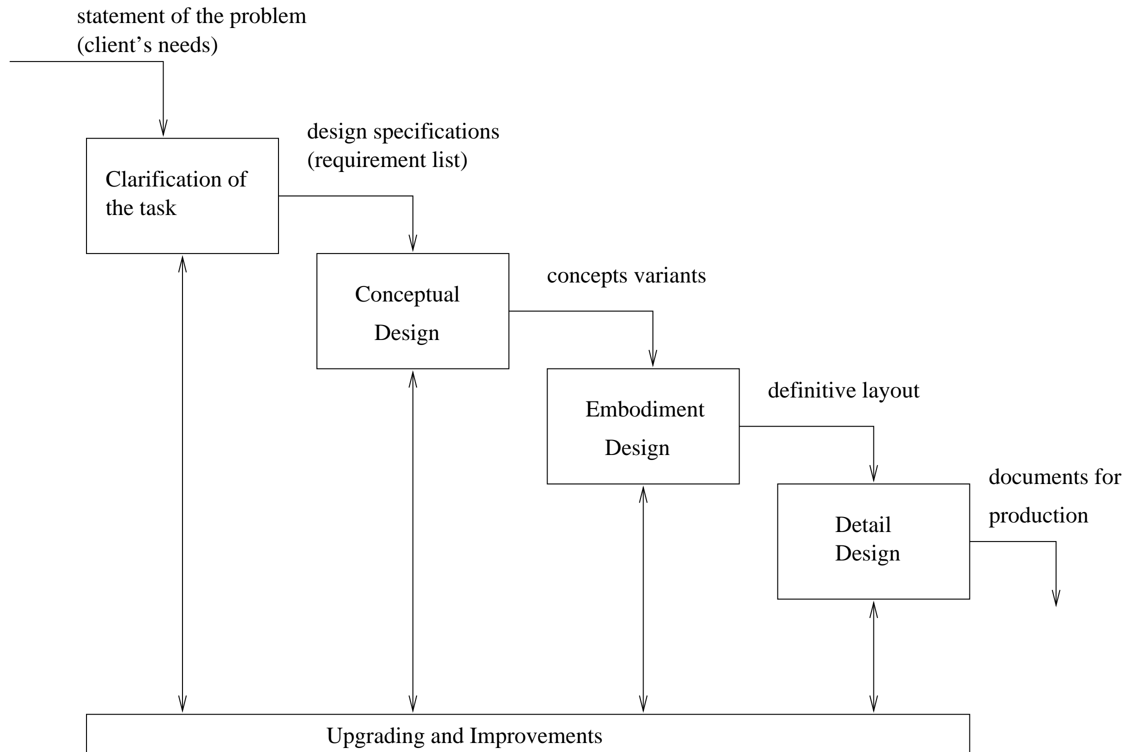


Figure 2.3: The Product Design Process

straint problems, following a similar structure to this thesis. Anderl and Mendgen also identify a number of typical applications of constraint-based design, such as feature-modelling and design with engineering constraints. The open issues they identify include the automatic generation of constraints and the evaluation of design alternatives as well as the constrainedness of problems.

Anderl and Mendgen conclude that

> Modelling with constraints is a modelling technique which contains a high potential for efficient working in all steps of the design cycle.

Serrano and Gossard [101] and Lamounier *et al* [62] concentrate more on the conceptual design phase and the use of engineering constraints rather than geometric constraints. These techniques are explored in more detail in section 2.3.4.

Sapossnek [97] also advocates constraint-based design. He states that design can be viewed as a constraint satisfaction process and notes that constraints can be on functionality, structure or manufacturability. He defines a constraint-based design system as one that explicitly represents and operates upon these constraints. Sapossnek identifies the separation of solution techniques from problem specification as being an important research issue. His concept of a general solver using multiple

techniques and separation of specification from solution is addressed in this thesis.

Dohmen [22] presents a survey of constraint satisfaction techniques for geometric modelling. Dohmen identifies geometric reasoning solvers using knowledge of the problem domain in the satisfaction process. This is a key issue in this thesis.

Constraint-based design allows the definition and solution of problems with tolerances, a problem of great interest to engineers [54, 93]. The parameterisation associated with tolerances is a natural use of constraint-based design and several authors allow the dimensioning and tolerance of designs [1, 69, 70].

Gorti and Sriram [45] present a framework for conceptual design. The framework they propose allows for incremental, evolving descriptions in an object-oriented environment. Gorti and Sriram also decouple different aspects of the overall problem to allow multiple reasoning methodologies - similar to the concept of hybrid solvers discussed in section 2.1.3. The framework allows geometric constraints and the investigation of alternative designs.

The use of constraints in design necessitates constraint solvers capable of satisfying the constraints. The current state-of-the-art in constraint solution is presented in the next section.

2.3 Constraint solvers

A constraint solver is an implementation of an algorithm that takes as input a constraint problem and produces as output a set of solutions to the constraint problem or the empty set if no solutions are found to the problem. Currently, there exist many different constraint solvers used to solve different types of problem. A large number of constraint solvers from a number of different fields have been studied. The constraint solvers investigated are presented in this section.

As the state-of-the-art was investigated, it became apparent that certain solvers were very good at solving particular types of problem. It is therefore reasonable to categorise solvers according to the type of problem they are best at solving. Lamounier [64] also categorised constraint solvers, but according to the type of solution algorithm used. The categorisation used by Lamounier is compatible with that used in this section. The advantages of the categorisation used in this thesis are that

1. It is hierarchical and can potentially be used to decompose a constraint problem automatically.

2. There is a natural mapping between types of constraint problem and types of solution technique.

The categorisation criteria identified in constraint solution are the use or not of domain specific knowledge; the type of constraint problem; and the constrainedness of the constraint problem. The categorisation of constraint solvers is presented in figure 2.4.

The use or not of domain specific knowledge forms the most fundamental distinction between solvers. Many constraint solvers restrict the type of problem they can solve so that they can take advantage of the structure of the restricted problems. Thus geometric constraint problem solvers can take advantage of the Euclidean space and rigid bodies in the problem. Alternatively, solvers that do not use domain specific knowledge, but instead handle constraints as a system of equations are called *general* and are discussed in section 2.3.1.

The remaining constraint solvers take advantage of domain specific knowledge and are called *domain specific*. The particular domain specific knowledge that the solvers use identifies the type of problem that they are best at handling.

Finite domain solvers take advantage of the finite nature of the problem size and typically use advanced search techniques in a finite solution space. These are discussed in section 2.3.2.

Geometric solvers take advantage of *geometric reasoning*. There exist many geometric constraint solvers currently and these can be further identified as those that solve over-, well- and under-constrained problems. Geometric solvers are investigated in section 2.3.3.

Functional (or engineering) constraint solvers are used to solve algebraic problems and are typically aimed at describing the functional aspects of a design at the conceptual design stage. Functional solvers take advantage of the structure of the equations to aid solution. The state-of-the-art is presented in section 2.3.4.

Maintenance and physical constraint solvers are used to simulate the physical environment and are introduced in section 2.3.5.

Chung and Schussel [19] compare variational and parametric solvers. They define a parametric solver as one that uses a predefined set of geometric constraints which are applied to the geometry by the engineer. They define a variational solver as one that makes no assumptions about the way in which geometric constraints are combined. Variational solvers typically use numerical or functional techniques to solve the system of equations, whilst parametric solvers correspond to geometric solvers in this thesis. They conclude that the use of one particular type of solver

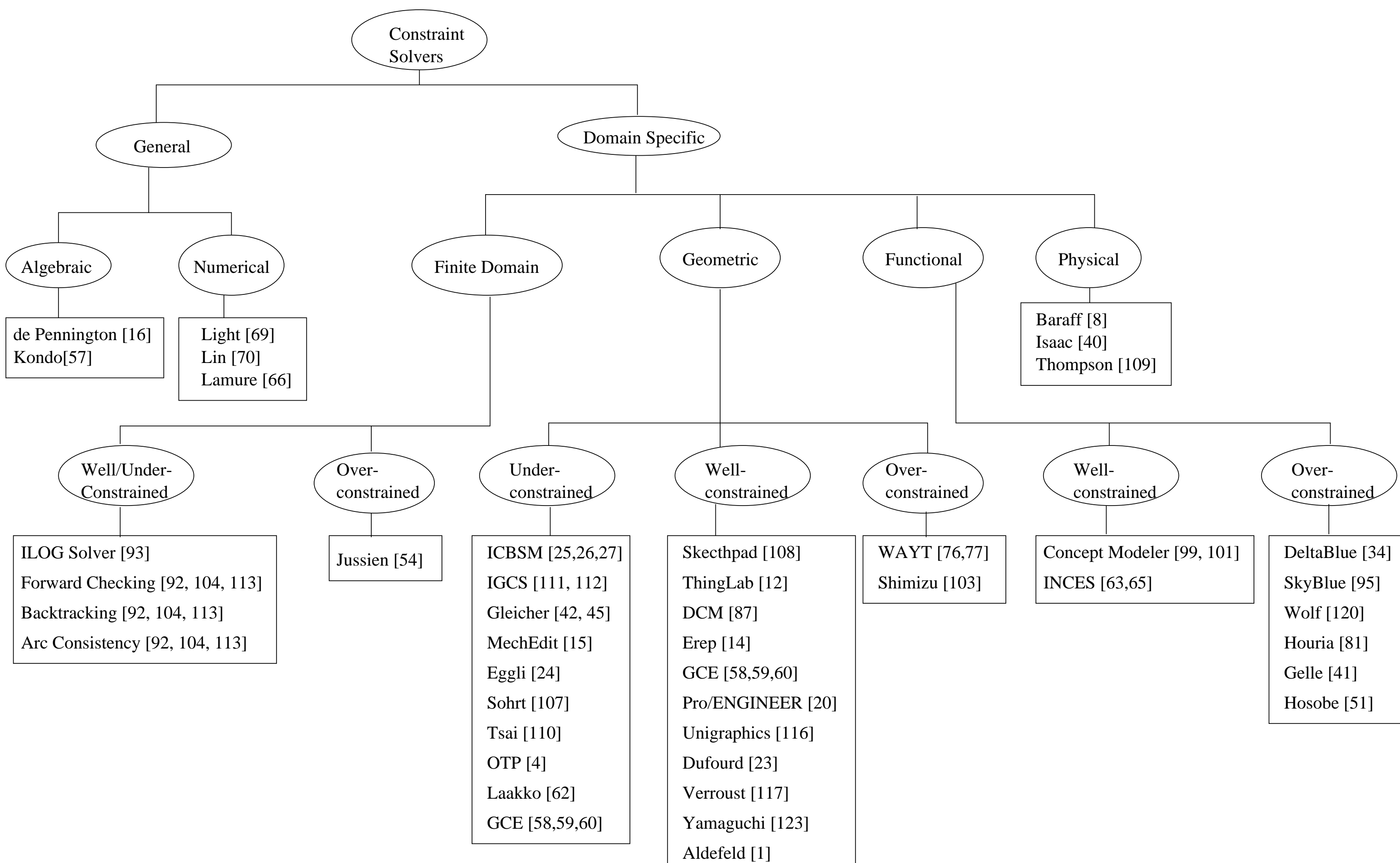


Figure 2.4: A Hierarchy of Constraint Solvers

depends on the design being created. Variational solvers tend to be more suitable for preliminary work, whilst parametric solvers are more suitable for fairly simple geometric design.

2.3.1 General constraint solvers

General constraint solvers handle constraint problems as systems of (typically) non-linear equations. The systems of equations are then solved using either numerical or symbolic techniques.

2.3.1.1 Numerical solvers

Numerical solvers take a system of equations and find a solution by using iterative techniques such as Newton-Raphson. The details of Newton-Raphson are not presented here as they are covered in detail elsewhere [69, 90]. Light and Gossard [69] and Lin *et al.* [70] use numerical techniques for variational geometry in computer-aided design systems. Several authors use numerical techniques as a backup to other approaches, for example MechEdit [15], INCES [62] and IGCS [112].

Numerical solution is general and solves a constraint problem as a whole entity. However, it suffers from a number of disadvantages:

1. Numerical solution is not robust and can fail to converge to a solution.

2. Numerical solution typically only finds one solution to a problem.

3. Numerical solution is computationally expensive, typically $\Omega(n^2)$ complexity, where n is the number of constraints,

4. Numerical solution may converge to a root but it may not be the expected solution to the problem.

Lamure and Michelucci [65] present a technique that resolves some of these problems. *Homotopy* avoids the convergence of Newton-Raphson to unpredictable solutions as it is much more predictable. Using the initial guess of the user corresponding to the user's initial sketch of the geometric system, homotopy interpolates between the system of polynomials describing the constraint problem and the system of polynomials describing the user's initial guess. The curve described by the interpolation can then be used to guide the solution of the constraint problem so that the roots

found are more predictable. Homotopy can be combined with decomposition techniques to speed up resolution. However, the price to pay for a more robust algorithm is that homotopy is, on average, some 10 to 20 times slower than Newton-Raphson.

2.3.1.2 Symbolic solvers

In the symbolic approach, general solvers use symbolic algebraic methods, predominantly Gröbner bases [17], to reduce a system of equations to a triangular system of polynomials that can be solved simply. Kondo [56] and Buchanan and de Pennington [16] have used Gröbner bases to solve systems of equations for geometric models.

Gröbner bases are particularly interesting as the final triangular system of equations is *complete* in the sense that it describes *all* solutions to the constraint problem. However, Gröbner bases are very computationally expensive and are therefore not appropriate for interactive applications.

2.3.2 Finite domain constraint solvers

Finite domain constraint problems include scheduling, resource management and some integer arithmetic. The domain specific knowledge used by finite domain constraint solvers is that there are only a finite number of possible configurations that need to be studied in order to look for solutions. The number of configurations may be very large but it is invariably finite. Thus it is possible to exhaustively examine all of the configurations and search for solutions.

Unfortunately, even very simple finite domain constraint problems, such as 3-SAT, are NP-complete. Consequently, it is not usually practical to check all configurations and knowledge of the structure of the problem is used to speed up the process. Tsang [114] provides a comprehensive description of the various strategies adopted for finite domain constraint satisfaction.

Although finite domain constraint problems are not often considered in the CAD and engineering design community, they form a useful testbed for studying ideas on the theory of constraint satisfaction for the simple reason that they are usually small and well-understood. For this reason, two finite domain satisfaction techniques will be described here in detail as representative of the general process. The descriptions of backtracking and forward-checking have been taken from Tsang [114], Smith [103] and Prosser [91].

2.3.2.1 Backtracking

Chronological backtracking is the simplest search algorithm used in finite domain constraint problems. A variable, v, is selected from the set of variables and a value, l, is selected from the domain of v. Variable v is then instantiated with the value l, so that $v := l$. The current values of all of the variables are then checked against the set of constraints in the problem. If a check fails then the value is inconsistent, the algorithm backtracks so that v is not instantiated and another value is tried. If all checks succeed, then another variable is chosen. If all of the possible values for v fail then the algorithm backtracks to the last variable successfully instantiated and chooses another value for it. This continues until all variables have been instantiated, in which case a solution has been found, or all possible configurations have been exhausted, in which case the problem has no solution.

2.3.2.2 Forward-checking

The forward-checking algorithm is a "look ahead" technique. A variable is instantiated with a value. Any values in the domains of other variables that conflict with this instantiation are removed from those domains and this process continues. If the domain of a variable becomes empty by this process then the instantiation is inconsistent and is changed. Forward-checking therefore prunes large parts of the search space quickly.

2.3.2.3 Other finite domain research

Research on finite domain constraint satisfaction problems is a well-developed field (see [81] and [60] for survey papers on finite domain constraint satisfaction) and has led to many studies of the performance of algorithms. Tsang *et al.* [115, 116] have compared a number of different algorithms and conclude that there is "no universally best choice of algorithm and heuristic combination." They recommend using a number of criteria on the type of problem to determine which particular solution algorithm to use. Although the criteria used in [115] are different, this principle is applied in this thesis to select algorithms to solve different types of constraint problem.

Freuder and Hubbe [35] present a control schema for solving constraint problems. Although Freuder and Hubbe's schema is primarily aimed at finite domain problems, it does have implications for the work in this thesis. The schema is presented below:

```
Place the initial problem on the Agenda
```

```
Until Agenda empty:
    Remove a problem P from Agenda
    If P has only instantiated variables
        then Exit with their values
        else
            Decompose P into a set of subproblems {Pi}
            Place each non-empty Pi onto the Agenda
Exit with no solution
```

For all constraint satisfaction processes, the decomposition technique is the key. Freuder and Hubbe use the schema to formulate descriptions of common algorithms such as backtracking and forward-checking. Freuder and Hubbe note that a general schema such as the one given "facilitates presentation and comparative analysis of ... algorithms and suggests new algorithmic possibilities". The work done in this thesis, particularly in chapter 6, also allows the description of algorithms in a common framework.

Finite domain problems may also be inconsistent. Jussien and Boizamault [53] present a solution technique that takes advantage of the constraint hierarchies developed by Borning *et al.* [11] by using an Assumption-based Truth Maintenance System to decide when to relax a constraint, which constraint(s) to relax and how to delete a constraint.

2.3.3 Geometric constraint solvers

The original geometric constraint solver was Sketchpad [108] which used local propagation and relaxation to solve constraint problems. ThingLab [12] extended Sketchpad by allowing information that was not purely graphical. ThingLab used Smalltalk and allowed constraints to apply to non-numeric objects such as text. ThingLab was a significant advance in geometric constraint modelling.

Geometric constraint solvers take advantage of domain specific knowledge about rigid bodies, Euclidean space and the actions of geometric objects within Euclidean space. Some solvers take advantage of local propagation techniques whilst some use ruler-and-compass construction to solve constraint problems. However, there are three identifiable strains of geometric constraint solver - those that handle under-constrained problems efficiently, those that handle well-constrained problems efficiently and those that handle over-constrained problems efficiently.

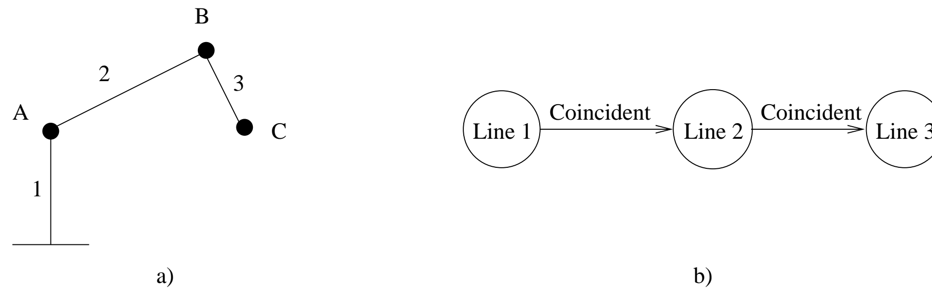


Figure 2.5: An Arm with Two Joints and the Relationship Graph for the Problem

Under-constrained solvers are usually incremental, building up a model step-by-step, frequently taking advantage of the user's interaction to help guide the solution process. Under-constrained solvers are discussed in section 2.3.3.1.

Well-constrained solvers are more usually specify-then-solve. Instead of building up a set of constraints gradually, the set of constraints is specified all at once. These are then solved using ruler-and-compass or rule-based methods to give solutions to the problem. Well-constrained solvers are investigated in section 2.3.3.2.

Over-constrained solvers also tend to be of the specify-then-solve variety. However, over-constrained solvers specialise in identifying inconsistency in constraint problems, probably caused by having too many constraints. The over-constrained solver will then select constraints that should be removed to make the system consistent. Over-constrained solvers are studied in section 2.3.3.3.

2.3.3.1 Under-constrained geometric constraint solvers

Fa *et al.* developed ICBSM, the Interactive Constraint-Based Solid Modeller [25–27, 31, 32], at the University of Leeds. ICBSM allows the user to build a constraint model in a virtual environment using direct manipulation of geometric objects. The key advances introduced by ICBSM are the use of Allowable Motion and Automatic Constraint Recognition. Allowable Motion is a local propagation technique that propagates changes of values of a geometric objects as it moves to other geometric objects that are constrained to move with it.

For example, consider the simple 2D arm shown in figure 2.5 (a). In ICBSM, this has the corresponding *relationship graph* in figure 2.5 (b). If line 3 is moved then, as it has been constrained to remain connected to line 2, it can only move in a circle around point B. However, if line 2 moves then, as line 3 is connected to line 2 and is free to move, line 3 rotates with line 2 around point A, as the movements of line 2 are propagated to line 3.

Automatic Constraint Recognition is the process whereby the user creates models within the virtual environment. As an object is moved around in the virtual environment, possible constraints are continually being detected and the user is notified of these. If the user wishes to activate one of these constraints then the object is simply released and the constraint is created automatically.

However, ICBSM is limited by the local propagation technique adopted and cannot handle cycles in the relationship graph. Tsai *et al.* have used locus analysis and other techniques to solve this problem, essentially turning ICBSM into the hybrid constraint solver IGCS, the Interactive Geometric Constraint System [112, 113].

ICBSM is also quite slow, with bottlenecks caused by the rendering of the model and the Automatic Constraint Recognition. Feng Gao *et al* [29] have helped to alleviate this problem. Maxfield has expanded ICBSM into a collaborative, distributed virtual engineering environment [78, 79] and Munlin has used ICBSM to build complex assemblies [30].

ICBSM was heavily influenced by the Degrees of Freedom Analysis approach used by Kramer in his Geometric Constraint Engine [2,9,57–59]. Degrees of Freedom Analysis does not fit comfortably into being a well-constrained or under-constrained approach because it is a hybrid of two techniques: *action analysis* and *locus analysis* and can handle well-constrained problems using locus analysis. Action analysis is a local propagation approach, equivalent to the Allowable Motion technique described above.

Locus analysis is used to "determine where in global space certain classes of partially constraint (geometric objects) must lie" [58]. Locus analysis works by examining the *loci* of geometric objects. The locus of an object is the set of possible positions that the object can take in space. If two objects are constrained to be coincident then the intersection of their loci satisfies the coincidence constraint.

For example, consider the problem described in figure 2.6 (taken from [58]). Here, line L_2 must remain fixed at point P. Correspondingly, the locus of point P_2 is the circle L_C. Also, the circle of variable radius C_2 is constrained to remain tangent to circle C and line L. The locus of the centre of C_2 is therefore curve L_P. Suppose a new constraint is added, that the centre of C_2 is coincident with point P_2. Then there are two possible configurations that satisfy this new constraint, as well as satisfying all of the previous constraints. These two configurations correspond to the centre of circle C_2 being at point P_3 or point P_4, found by intersecting L_C and L_P.

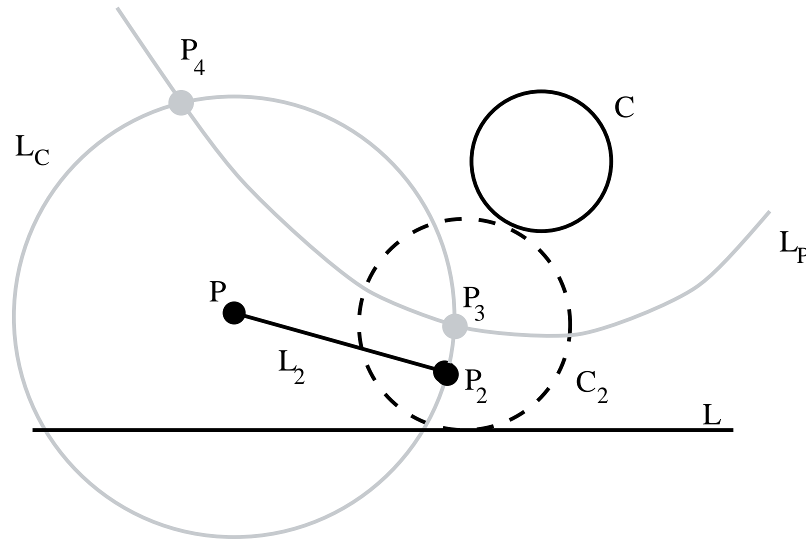


Figure 2.6: An Example of Locus Analysis

Kramer's approach is very powerful and the combination of the two techniques allows for much more flexibility than one technique alone. The great benefit of Degrees of Freedom Analysis and Allowable Motion is that they reduce a system of highly nonlinear, highly coupled equations into a set of simpler, discrete, high-level constraints.

Gleicher also investigates the use of direct manipulation in defining constraint problems [41–44]. Gleicher uses *differential manipulation* to couple user controls to graphical objects in a powerful manner that is closely linked to constraints. The approach provides abstractions that enable new interaction techniques.

The constraint solution technique used is an equation solving method. However, in order to solve the equations quickly and efficiently, differential manipulation controls the motion of the objects over time, adding a form of elasticity to the direct manipulation of objects. This domain specific knowledge helps to make differential manipulation practical.

Brunkhart [15] has created MechEdit, a geometric constraint system for solving systems of planar linkages. MechEdit uses local propagation and contraction in order to solve under-constrained systems of linkages quickly. Brunkhart has conducted experiments that indicate that his hybrid symbolic/numeric solver is significantly faster than a numeric solver alone, in some cases improving the number of refreshes per second from 0.02 to 222.

Eggli *et al.* [24] use 'Quick-Sketch' to infer 3D models given a freehand sketch with some constraints that can be inferred from the sketch. The geometric con-

straint satisfaction technique uses a degrees of freedom approach [59] to guide the construction steps for the model.

Sohrt and Brüderlin [106] also use interactive manipulation with constraints. The constraint solver that they use consists of two stages. First the geometric constraints are interpreted as prolog predicates and are translated using rewrite rules until the geometry can be determined. This is a planning stage. Then the symbolic solution created by the rewrite rules is numerically evaluated each time an object or constraint is altered. This allows fast interaction with the user when nothing changes. However, new objects or constraints require recalculation of the symbolic solution.

Tsai *et al.* [111] use local propagation techniques for incremental assembly. The constraint graph of a problem is analysed and this is used to decide on whether to reuse old information or to resolve the problem.

Arbab and Wang's approach, Operational Transformation Planning (OTP) [4], uses a "high-level understanding of the semantics of constraints and the geometric implications of operations for satisfaction planning". This use of domain specific knowledge allows OTP to satisfy a network of constraints incrementally.

Laakko and Mantyla [61] use SkyBlue [94], a local propagation solver to create geometric models in an incremental fashion. Their solver, EXTDesign, uses various subsolvers to handle cyclic subproblems. SkyBlue is discussed in more detail in section 2.3.4.1.

2.3.3.2 Well-constrained geometric constraint solvers

Well-constrained geometric constraint solvers typically operate using a specify-then-solve paradigm. In this paradigm a set of geometric objects and a set of geometric constraints are defined. The set of geometric constraints is sufficient for the constraint problem to be well-constrained. Then all of the constraints are used to find solutions to the problem.

DCubed uses this technique [73–75,86]. The DCM algorithm takes a set of 2D or 3D geometric objects and a set of distance and angle constraints. Using Hopcroft and Tarjan's subdivision algorithm [49], the constraint graph representing the constraint problem is divided into a number of split components. These split components can then be solved using simple ruler-and-compass construction techniques and the solutions to the subproblems are combined to form a solution to the whole problem.

For example, the pentagon in figure 2.7 is well-constrained and can be represented using the constraint graph in figure 2.8, where lines represent distance or angle

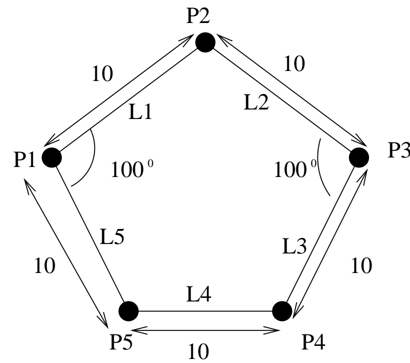


Figure 2.7: A Pentagon Defined by Distance and Angle Constraints

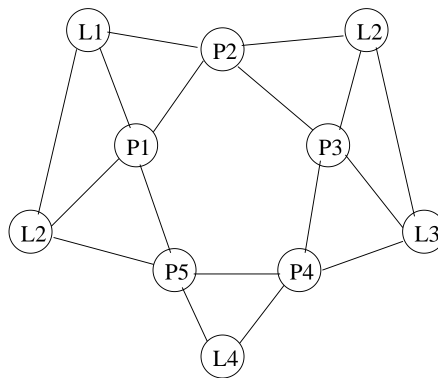


Figure 2.8: The Constraint Graph for the Pentagon

constraints. Then Tarjan's algorithm is used to identify the split components of the graph. These split components can be solved immediately and then recombined to give the picture in figure 2.7.

DCM claims to have an approximately linear speed of solution though Bouma *et al.* claim it is in fact quadratic [14]. DCM can be used for under- and over-constrained problems though it is not best suited for such cases as the algorithm attempts to find an acceptable solution to an under-constrained problem by changing as few geometries as possible. However, although it guarantees to find solutions to well-constrained problems, it cannot guarantee to find any solution to an under-constrained problem, even though an infinite number may exist. DCM deals with an over-constrained subproblem by placing it in a subgraph and then solving the rest of the problem.

Erep [37–39,48] is very similar to DCM. Erep studies the problem of under- and over-constrainedness in some detail. However, the key strengths of Erep lie in its ability to solve well-constrained problems and its ability to choose between solutions to a problem to find the solution the user desires.

Pabon *et al* [87] have extended Kramer's GCE engine to allow for feature and variational modelling. The aim of Pabon *et al.*'s research is similar to the aims in this thesis: to integrate different forms of constraint technique. However, Pabon *et al.* do not demonstrate any theoretical basis for their work, nor any discussion of domain specific knowledge, nor how solvers interact generally. These topics are covered in this thesis.

Commercial CAD packages, such as Pro/ENGINEER [20] and Unigraphics [117] allow well-constrained geometric constraint problems to be defined and solved.

Dufourd *et al.* [23] handle systems of geometric constraints using multiple constraint solvers. They describe geometric constraint systems in terms of predicates describing the constraints. By taking advantage of the invariance under displacement of constraint problems in a CAD environment, Dufourd *et al.* break such constraint problems into smaller ones which are easier to solve.

The smaller constraint problems are then solved using one of three *local solvers*. Two of the three local solvers use rule-based methods to solve the geometric constraint problems, whereas the third is a numerical technique. The first rule-based method assumes that there are no loops in the constraint problem and so uses local propagation to solve the constraint problem. The second rule-based method allows loops but involves a much more complete set of geometric construction rules. This second solver is not complete but Dufourd *et al.* claim that it is successful in most cases in achieving sophisticated constructions. Dufourd *et al.*'s constraint solver is similar in form to the hybrids discussed in this thesis. The domain specific knowledge that they take advantage of is the invariance under displacement of CAD models.

Verroust *et al.* [119] use an expert system to identify a sequence of computation. Yamaguchi and Kimura [125] develop a technique for simplifying the construction of consistent and sufficient constraint problems in order to make problems well-constrained. A constraint problem is consistent if it has a solution. A constraint problem is sufficient if it is well-constrained. Aldefeld [1] uses a rule-based method to solve well-constrained problems.

2.3.3.3 Over-constrained geometric constraint solvers

Constraint solvers that specialise in over-constrained problems tend to concentrate on resolving conflicts between constraints. This can involve using constraint hierarchies [11] or solving by other means.

Mäntylä has developed WAYT, Why-Are-You-There? [76,77]. WAYT is a mod-

elling environment for assembling products. WAYT uses hierarchical descriptions of models to capture design information and uses DeltaBlue [34] as the integral constraint solver used to solve the constraints. DeltaBlue can solve only linear equality constraints but can take advantage of constraint hierarchies to resolve conflicts. DeltaBlue and its brother SkyBlue are discussed in more detail in section 2.3.4.1.

Shimizu and Numao [102] use a different technique to resolve conflicts between constraints. They propose using an Automated Truth Maintenance System in order to distinguish between redundant and conflicting situations.

Connectivity Analysis [67], DCubed [86] and Erep [14] also identify and resolve over-constrained problems.

2.3.4 Functional constraint solvers

Functional (or engineering) constraint solvers solve constraint problems that describe the *functionality* of a design as compared to the *physicality* of the design. As such, they are primarily intended for use in the conceptual design phase when the functional description of the product is prepared. Generally, the functional description consists of a system of nonlinear equations and as such little domain specific knowledge can be applied to solving the system. If no domain specific knowledge can be applied, the system of equations must be solved by a general solver such as Newton-Raphson or Gröbner bases.

However, certain systems of equations have a structure that can be taken advantage of. For example, linear systems of equations can be solved by taking advantage of the sparseness of the matrix representing the system of equations and using LUD decomposition [21] to solve the problem in a relatively efficient fashion.

Serrano has created a system called Concept Modeler [98–101] that takes advantage of triangular systems of equations. Concept Modeler takes as input a system of equations, such as that given below (taken from [100]), and uses the constraints

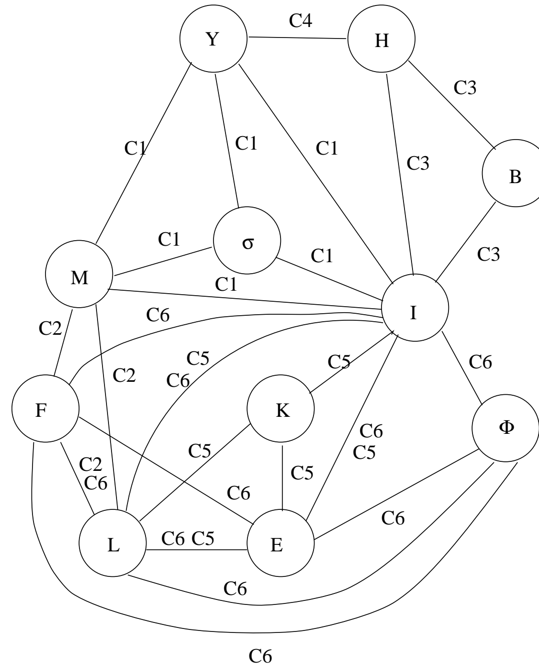


Figure 2.9: Equation Graph for Constraints $C1$ to $C6$

to build the *equation graph* given in figure 2.9.

$$\begin{aligned}
C1 &: \sigma - \frac{MY}{I} = 0, \\
C2 &: M - FL = 0, \\
C3 &: I - \frac{WH^3}{12} = 0, \\
C4 &: Y - \frac{H}{2} = 0, \\
C5 &: K - \frac{3EI}{L^3} = 0, \\
C6 &: \phi - \frac{FL^2}{EI} = 0.
\end{aligned}$$

The user then specifies which of the variables are known and which are not. In this case, variables M, σ, K, E and ϕ are known. Using graph matching techniques this allows the creation of the tree-like structure in figure 2.10 which is used to derive the sequence of constraint satisfaction for the problem.

Serrano then uses the values calculated for variables earlier in the constraint satisfaction sequence to give values for variables later in the sequence. However, the subgraph given by the variables I, L, F in figure 2.10 is *cyclic* and must be solved

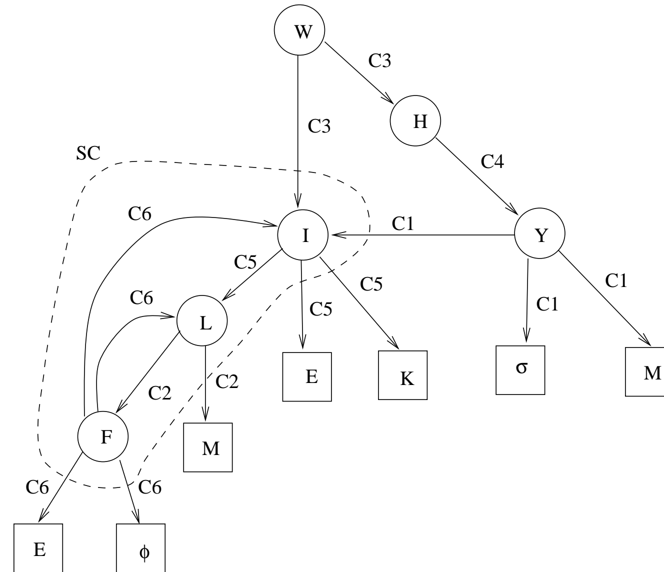


Figure 2.10: Tree-like Representation for Equation Graph

simultaneously. It is therefore collapsed into a strong component † SC which is solved separately. The sequence of satisfaction for figure 2.10 is $SC - Y - H - W$.

Serrano presents two methods of finding redundancies and conflicts within strong components. The first uses symbolic manipulation of the constraints within the strong component so that each constraint is described in terms of one variable and then constraints are successively eliminated by substitution into another constraint. For example, if a constraint problem consists of four constraints $\{f_1, f_2, f_3, f_4\}$:

$$f_1 : x_1 - x_3 = 0,$$
$$f_2 : x_2^3 - 1 = 0,$$
$$f_3 : x_1^2 + x_4 - A = 0,$$
$$f_4 : x_3^2 + x^4 - B = 0,$$

each constraint can be described simply in terms of a function of one variable. In this case, f_1 can be rewritten as $x_1 = x_3$, f_2 as $x_2 = 1$, f_3 as $x_4 = A - x_1^2$ and f_4 as $x_3 = \pm\sqrt{B - x_4}$. Substitution leads to the expression

$$x_1 = \pm\sqrt{B - (A - x_1^2)}.$$

If $A = B$ then the set of constraints $\{f_1, f_2, f_3, f_4\}$ is redundant. If $A \neq B$ then the set of constraints is conflicting and has no solution. However, this technique is, in general, insufficient. Most constraint problems cannot be solved using substitution

r
0
J nxn Δ x = -f

Figure 2.11: Typical result of Gaussian elimination on linearised constraint problem

techniques as constraints are not usually a simple function of a single variable. Gröbner bases could be used to identify redundancies in the strong components and to solve the strong components. However, the utility of Serrano's approach is then limited to the use of Gröbner bases.

The second method Serrano uses for finding redundancies and conflicts involves numerical techniques. First, the strong component is linearised by constructing its Jacobian matrix to give the following system of equations:

$$J\Delta x = -f,$$

where J is the Jacobian, Δx is the vector of $(x_i - x_0)$, where x_i are the unknowns and x_0 are the initial guesses, and f is the vector of the constraints evaluated at x_0.

Serrano notes that if the Jacobian is non-singular then the strong component is consistent and can be solved using the above matrix equation for each iteration of a Newton-Raphson method. The Jacobian can never be structurally singular (when the Jacobian is non-square [69]) as the strong component will always contain as many constraints as variables.

The Jacobian can be numerically singular if the determinant of the Jacobian is equal to 0. In this case, Gaussian elimination would lead to a matrix of the form in figure 2.11.

If an entry in $-f$ is zero for $i, r < i \leq n$, then the corresponding constraint to row i is redundant, otherwise the constraint is in conflict with the other constraints in the strong component.

Numerical inaccuracies in adopting this technique may mean that the determinant of the Jacobian is close to 0 and is approximated to 0. Consequently, non-singular matrices may be identified as being singular.

Serrano's approach is very similar to Light and Gossard's [69]. The chief difference is that Serrano only applies the numerical techniques described above to strong components that the initial algorithm identifies, whilst Light and Gossard apply their numerical approach to the entire constraint set. Serrano's approach

therefore does not use the more expensive numerical technique for acyclic subproblems. However, as will be proved later in this thesis, the decoupling of the cyclic and acyclic subproblems in Serrano's approach may lead to a failure to find solutions where Light and Gossard's algorithm will succeed.

Serrano states that this graph-based approach has proven to be more efficient than numeric and symbolic techniques. However, the speed of the system is dependent on there being relatively few cycles in the graph.

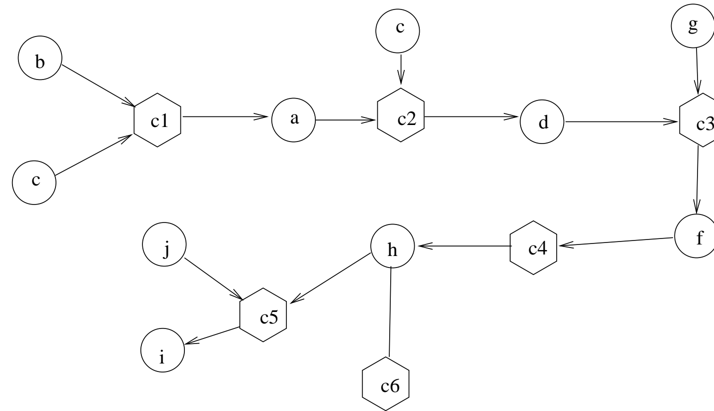
Lamounier [62–64] has improved on Serrano's algorithm by making it *incremental*. Each time a constraint is added to Serrano's system, the entire satisfaction sequence must be derived from scratch. For very large sets of equations this will take a long time. Lamounier has improved the algorithm by ordering the satisfaction process in such a way that when a new constraint is added, it only affects as small a part of the graph as possible. For example, adding constraint $c6$ to the graph given in figure 2.12 (taken from [63]) would only affect constraints $c5$ and $c4$ and the variables i, h, f and g rather than the whole problem.

Lamounier has also tried to describe more general constraint problems by allowing geometric constraints in INCES. However, to do this the geometric constraints are reduced to systems of equations and this loses all domain specific knowledge associated with the geometric constraints.

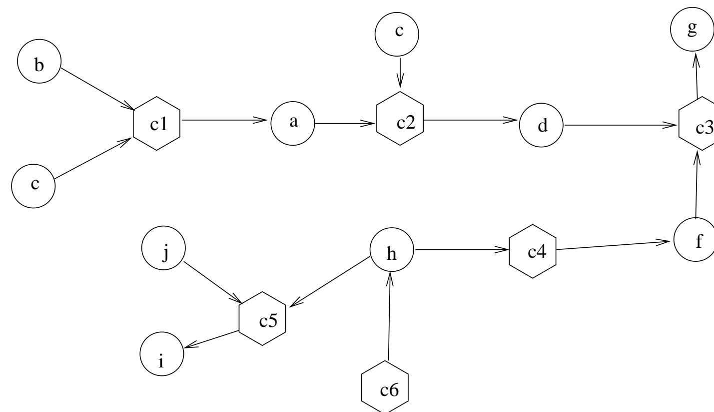
Serrano and Lamounier's work depend on *local propagation*. Local propagation is equivalent to triangular form in matrices or a tree structure in constraint graphs. Many solvers use local propagation as it is very efficient [15, 27, 62, 94, 101, 112]. However, local propagation cannot handle cycles or systems of equations that must be solved simultaneously. Serrano and Lamounier both deal with simultaneous subproblems by passing them to a numerical solver. One of the issues identified by this thesis is that this may lead to the whole solver being *inconsistent* in the sense that it may fail to find solutions when they exist.

2.3.4.1 Over-constrained functional constraint solvers

Over-constrained functional constraint solvers attempt to resolve the conflicts caused by having too many constraints for the number of variables. In order to do this Borning *et al.* [11] devised *constraint hierarchies*. Constraint hierarchies involve assigning each constraint a *strength*, indicating how important it is that the constraint be satisfied. The constraint problem is then solved using the constraint hierarchy to satisfy the most important constraints. The precise trade-off between strengths is determined by a *comparator*.



a) Before Inserting Constraint c6



b) After Inserting Constraint c6

Figure 2.12: Example of Incremental Insertion of Constraint. Arrows in Graph Indicate Order of Satisfaction

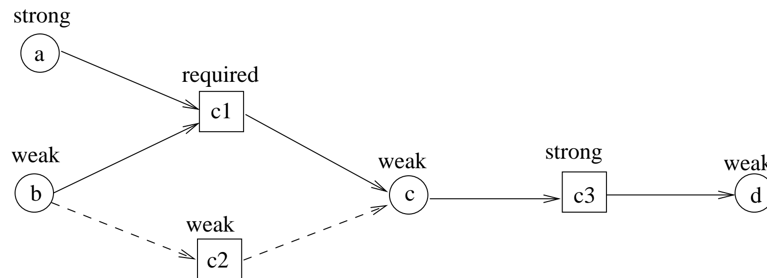


Figure 2.13: Example Constraint Graph for Hierarchical Constraint Problem

Constraint hierarchies have become very popular as they formalise the means by which conflicts in over-constrained systems can be resolved in a predictable fashion.

One of the first constraint solvers to take advantage of constraint hierarchies was DeltaBlue [34]. DeltaBlue is an incremental, local propagation constraint solver. It uses a bipartite constraint graph of constraints and variables to determine the flow of propagation from variable to variable and uses the strengths of constraints to determine which constraints need to be solved. Each constraint has a number of *methods* used to determine how a constraint is satisfied. Only one of these methods is active at any given time.

For example, consider the graph in figure 2.13. Constraints are squares and variables are circles. The direction of the arrows indicates which method is to be used, and dashed arrows indicate that a method is not active. Here constraint $c2$ has been sacrificed so that $c1$ is satisfied. In this case, once a and b are known, $c1$ can be used to calculate c. Once c is known, $c3$ can be used to calculate d.

DeltaBlue is fast as it is incremental: adding a new constraint will only affect a small part of the constraint graph. However, DeltaBlue is limited as it uses local propagation. Any cycle in the constraint graph cannot be solved. DeltaBlue also only allows single output variables.

SkyBlue [94], the successor to DeltaBlue, allows solution of cycles by calling external cycle solvers. SkyBlue also allows multi-output constraints. SkyBlue and DeltaBlue have been used in several other constraint solvers [71, 95, 96]. Hosobe *et al.* [50] propose a similar approach to SkyBlue.

Gelle and Smith [40] present a system for resolving over-constraint in a dynamic environment. They use a logic framework to determine solutions of the problem.

Wolf [123] uses constraint hierarchies to transform an over-constrained constraint hierarchy problem into a 'normal' constraint problem which can then be solved.

Houria [80] is another solver that uses constraint hierarchies and local propagation, but uses a global comparator and satisfies more constraints than other solvers.

2.3.5 Maintenance and physical constraint solvers

Maintenance simulation involves a virtual representation of a real environment in which an engineer or a mechanic can use virtual tools to assemble and disassemble complex machinery. Physical simulation involves a virtual representation of the real world, including gravity, friction, acceleration and forces. These two areas of study are closely linked as forces are involved in maintenance simulation to detect, for example, the amount of torque placed on a wrench to turn a screw.

Constraints do not play a particularly large part in physical modelling. However, there do exist some constraint methods for physically based modelling. Physical laws, such as gravity, friction and non-interpenetration of solid bodies, can be described using constraints, and such laws are very important for realistic engineering design. For example, Platt [89] presents a method based on constraint stabilisation and dynamic constraints that he uses to describe deformable models and for collisions between the models.

Project Isaac [118] is intended to combine accurate, efficient and robust techniques for collision and contact detection in order to simulate large and complex geometric models within a virtual environment. Others studying physical modelling include Baraff [8] and Witkin [122].

Whilst much research has been done on physical modelling, little has been done on investigating the support of maintenance analysis. Thompson [110] is currently investigating tools and techniques for supporting maintenance analysis within virtual environments. It is hoped that Thompson can extend the work done by Fa [25] and Munlin [30] so that dynamic maintenance analysis can be carried out in the constraint-based virtual environment of ICBSM.

2.4 Conclusions

This chapter has presented the state-of-the-art in constraint satisfaction. The theory underlying constraints was investigated, including the links with relational algebra as well as dimensions, decomposition and solution spaces.

Constraints are used in many different fields and constraint solvers tend to be biased towards a particular field. The fields identified in this thesis are general, finite domain, geometric, functional and physical. Other than the general constraint solvers, solvers within a particular field tend to take advantage of domain specific knowledge about that field. Domain specific knowledge allows more efficient solvers

to be created as well as solvers that can identify desired solutions more easily and also can be guaranteed to converge to solutions.

In engineering design, the most used types of solvers are general, geometric and functional. Finite domain and physical solvers are still significant in terms of engineering design but form a subsidiary part of this thesis. Geometric and general solvers are frequently joined to form hybrids [15, 87, 112]. Functional and general solvers are also occasionally joined as hybrids [94]. However, it is rare that two domain specific solvers are used in conjunction [62].

Hybrids are useful as they allow a constraint solver to solve more constraint problems than would be possible normally. In this thesis, hybrids will be described as being more *expressive*. Domain specific solvers tend to be very efficient but also quite restricted in the problems they can solve. This thesis investigates the use of hybrids of domain specific solvers.

However, in order to investigate hybrids of solvers, commonalities between different types of solvers must be found. To this end, an abstraction of the constraint satisfaction process is developed in the next four chapters. This abstraction allows common elements of constraint solvers to be identified and then exploited when the solvers are linked together.

The criteria identified for distinguishing constraint solvers as given in figure 2.4 allow the association of constraint solvers with problems that are particularly well suited to that constraint solver. This will allow the definition of a hybrid constraint solver in chapter 7 that will take best advantage of domain specific solvers.

Chapter 3

Solving Constraint Problems by Decomposition

This thesis investigates the definition, representation and solution of engineering design constraint problems. This chapter investigates existing engineering design constraint solvers and discusses the general method of solving constraint problems that these solvers use.

The principle behind most existing constraint solvers is that it is easier to solve lots of small problems rather than one large problem. This is the divide-and-conquer concept prevalent in computer science.

Freuder and Hubbe's control scheme for solving constraint problems [35] is an example of the divide-and-conquer approach to constraint solution and forms a useful basis for discussing most current constraint solvers. It is repeated here for convenience.

```
Place the initial problem on the Agenda
Until Agenda empty:
    Remove a problem P from Agenda
    If P has only instantiated variables
        then Exit with their values
        else
            Decompose P into a set of subproblems {Pi}
            Place each non-empty Pi onto the Agenda
Exit with no solution
```

A prime consideration of infinite-domain solvers, though often not explicitly stated, is that *ordering* a set of subproblems can significantly simplify their solution.

Given constraint problem P,
1. **Decompose** P **into a set of subproblems** $\{P_i\}$.
2. **Order the subproblems**.
3. **Solve the subproblems in the order given**.

Table 3.1: Control scheme for solving constraint problems

For example, consider the two problems P_1 and P_2:

$$P_1 = (\{(x, \mathbb{R}), (y, \mathbb{R}), (z, \mathbb{R})\}, \{x^2 + y^2 + z^9 = 43, 3x^2 + 4y^2 + \sin z = 256\}),$$
$$P_2 = (\{(z, \mathbb{R})\}, \{z = 4\}).$$

Solving P_1 first and then P_2 is extremely hard. However, if P_2 is solved first then P_1 is much simplified. The order of solution of the subproblems has significantly improved solution of the combined problem, $P = P_1 \cup P_2$.

Finite domain solvers also benefit from a careful ordering of subproblems as evidenced by the use of heuristics to help guide solution. Variable and value ordering can frequently make a hard finite domain constraint problem much simpler [103,114].

Freuder and Hubbe's control scheme can be adapted to incorporate the ordering of subproblems by dividing the solution process into three stages as shown in table 3.1.

Note that Freuder and Hubbe do not use constraint solvers *per se*, but decompose until variables are instantiated. Thus, the decomposition solves the constraint problem. Here decomposition is distinguished from solution so that existing constraint solvers can be applied to subproblems when it becomes possible to do so and because decomposition to the level of instantiation is not usually possible in infinite-domain problems.

Note also that Freuder and Hubbe continuously decompose the subproblems until they can be solved. In part this is because the decomposition is used to solve the subproblems. However, in part it is because the decomposition of subproblems can lead to other subproblems that can be decomposed further. The solution process presented in table 3.1 assumes that the decomposition of P into subproblems $\{P_i\}$ is comprehensive in the sense that any further decomposition of a P_i takes place as part of the solution phase. In this way, the general constraint solver does not reinvent the wheel and can reuse any existing solvers.

This approach means that the constraint solver described in table 3.1 should take into account the constraint solvers available. The constraint solver then decomposes

Given constraint problem P,
1. **Decompose P into a set of subproblem-solver pairs $\{(P_i, \mathcal{S}_i)\}$ such that P_i can be solved efficiently by one of the domain specific solvers in $\mathcal{S}_i$ or P_i can be solved by a domain general solver in $\mathcal{S}_i$.**
2. **Order the subproblem-solver pairs taking into account hybrid collaborations and domain specific knowledge.**
3. **Solve each subproblem using the solver selected and the hybrid collaboration chosen.**

Table 3.2: Control scheme for solving constraint problems using domain specific knowledge and hybrid constraint solvers

the constraint problem according to the solvers available, orders the subproblems and then initiates the solvers on the subproblems in the order decided. Clearly the *decomposition strategy*, the *ordering strategy* and the *available solvers* become the key elements of the constraint solver.

As noted in chapter 1, the use of domain specific knowledge and hybrid collaboration have been identified as being powerful techniques to take advantage of. Consequently, it is of particular interest to identify how these two concepts fit into the above constraint solver framework. In fact, the use of domain specific knowledge and hybrid constraint solvers forms a natural extension of the solution process in table 3.1.

The use of domain specific knowledge manifests itself in the availability of domain specific solvers. The constraint solver should decompose the constraint problem into subproblems that can be solved by the domain specific solvers advantageously. Consequently the constraint solver must know the strengths of the domain specific solvers available and how these can be identified in the constraint problem.

Hybrid constraint solvers are useful because they help to make explicit the ordering of subproblems and also allow solvers to pass solutions amongst themselves in a controlled fashion. Hybrid solvers are therefore particularly useful in stages 2 and 3 of table 3.1. The use of hybrid collaborations in stage 3 will be studied extensively in chapter 7.

The specialisation of table 3.1 to take into account domain specific knowledge and hybrid constraint solvers can therefore be described as shown in table 3.2.

This solution process has great potential for success. As discussed above, the key elements of the process are the decomposition strategy, the ordering strategy and the solvers available. Each of these elements is discussed in detail in the following

sections.

Section 3.1 presents four examples discussing currently existing constraint solvers in terms of the processes discussed in table 3.1 and table 3.2. Section 3.2 discusses the decomposition strategy and gives some examples of current decomposition strategies. In section 3.3, the ordering strategy is covered in detail, giving advantages and disadvantages of ordering strategies.

Section 3.4 discusses solution of the subproblems using domain specific solvers and the advantages and disadvantages thereof. Section 3.5 presents conclusions from this chapter.

3.1 Examples of current constraint solvers

This section presents four examples of current constraint solvers, discussed in terms of the processes described in the previous section. The first example uses the DCM solver introduced in [86]. The second example demonstrates the use of a hybrid of a domain specific solver and a domain general solver using the process in table 3.2. This example uses the INCES solver introduced in [62]. The third example discusses IGCS [112] and the fourth example discusses Connectivity Analysis [67], which is a hybrid of many domain specific solvers.

3.1.1 DCM

The two-dimensional constraint solver used by DCubed, DCM [86], solves constraint problems consisting of two-dimensional points and lines and distance and angle constraints. The algorithm works by decomposing the constraint problem into triconnected components† . The triconnected components will usually consist of triangles in a constraint graph constructed of either *real* or *virtual* edges. Then, triangles of constraints and entities consisting entirely of real edges are solved. This fixes the three entities in the triangle relative to each other and allows further triangles to be fixed and solved relative to these three entities. The DCM algorithm can be described in three stages:

1. Decompose the constraint problem P into a number of subproblems $\{P_i\}$, where each P_i is a triconnected component. The decomposition strategy creates virtual edges when a complex subgraph is split into two other subgraphs along an articulation pair† . The edge between the two vertices that form the

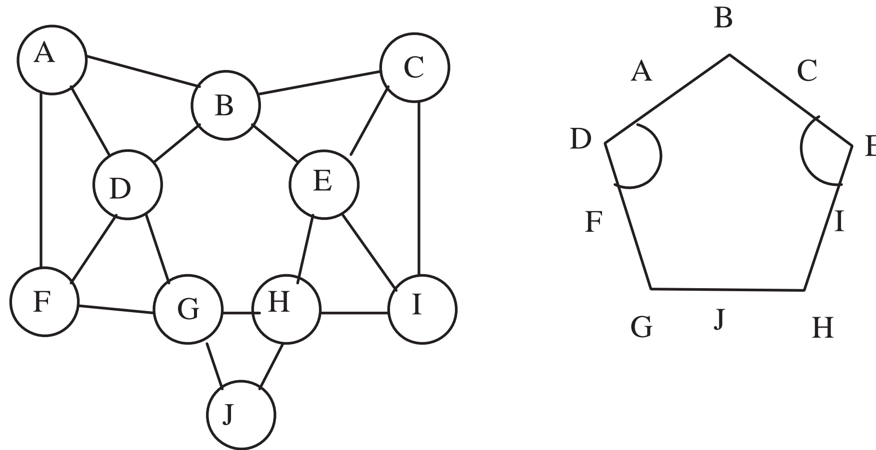


Figure 3.1: Constraint problem P describing a pentagon

articulation pair is part of one subgraph and is repeated as a virtual edge in the other subgraph.

2. Order the subproblems so that triangle (A, B, C) is solved before triangle (A, B, D) if there is an edge (A, B) common to both triangles that is real in (A, B, C) and virtual in (A, B, D). This forms the ordering strategy of DCM.

3. Each triangle is solved in turn, in the order dictated by the ordering strategy. Solvers find solutions to three simultaneous equations in three unknowns using special case or numerical techniques. Since the number of possible triangles is small and finite, special case techniques can be used efficiently.

For example, consider the constraint problem P in figure 3.1. The decomposition strategy in [86] gives the set of subproblems depicted in figure 3.2, where virtual edges are represented using dotted lines.

The partial order generated for these triangles is

$$\triangle DFG < \triangle ADF < \triangle ABD, \tag{3.1}$$

$$\triangle EHI < \triangle EIC < \triangle BEC. \tag{3.2}$$

The special case solvers would then be applied to the triangles in turn, maintaining this order. The special case solvers are effectively domain specific solvers. For example, assuming points D and G are fixed, a position for line F can be calculated using $\triangle DFG$. Since D and F are now fixed, a position for line A can be calculated using $\triangle ADF$. Finally, a position for B can be calculated using $\triangle ABD$. Eventually relative positions for B, G and H would be determined, as shown in figure 3.3.

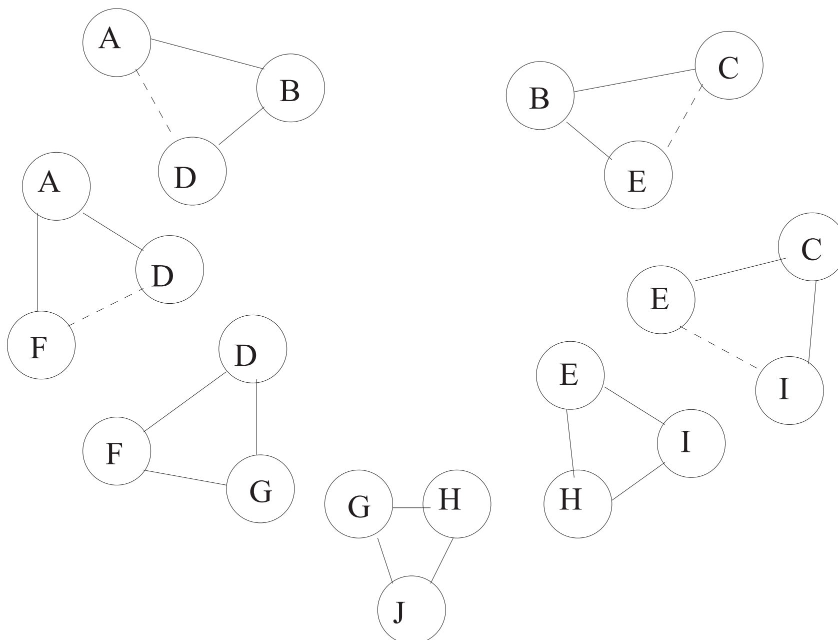


Figure 3.2: Decomposed subproblems of problem P

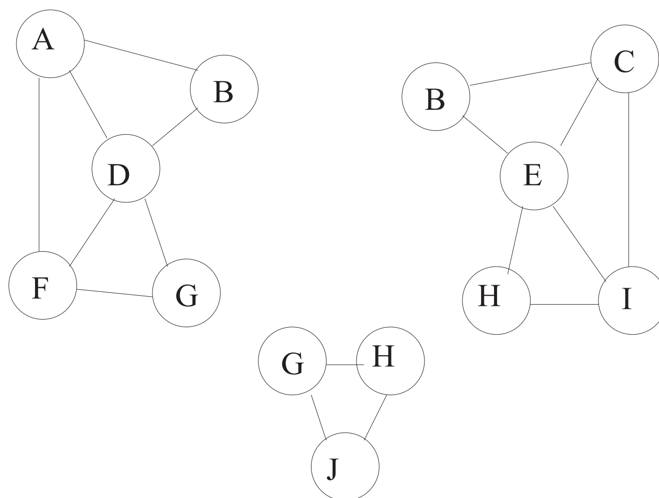


Figure 3.3: Recombined subproblems of constraint problem P

These three components can then be placed relative to each other, fixing B, G and H up to rigid body freedom.

Note the close correspondence here with Erep. Erep would solve P in exactly the same way, forming triangles DFG, ADF and ABD into a single cluster and triangles EHI, EIC and BEC into another cluster and then fixing these two clusters with triangle GHJ in similar fashion to that above. In fact, Erep and DCM have recently been acknowledged to be very similar [14].

3.1.2 INCES

The functional constraint solver INCES [62], developed by Lamounier at the University of Leeds, solves constraint problems consisting of variables on the real line and equations. The INCES algorithm solves such problems by examining the constraint graph of the problem and identifying strongly connected components† . Strongly connected components are equivalent to subproblems that must be solved simultaneously. Such subproblems, once identified, are renamed as a single entity in the constraint graph so that the constraint graph becomes acyclic.

The acyclic constraint graph is then solved using local propagation techniques so that entities that are fixed are solved first and entities that are connected are solved next, until a strongly connected subproblem is encountered.

The strongly connected subproblem is solved using a domain general solver. In the current implementation of INCES this is a Newton-Raphson technique. The result of the Newton-Raphson solver is passed downstream to other entities in the acyclic graph by local propagation and the process continues.

This is an example of the process described in table 3.2. This can be seen more easily if the INCES algorithm is described in three stages:

1. Decompose constraint problem P into a number of subproblems $\{P_i\}$, where each P_i is either a strongly connected component or is a connected, acyclic constraint graph.

2. Order the subproblems so that fixed subproblems are solved first, followed by subproblems that are connected to them, and so on.

3. Solve the subproblems in the order dictated by the ordering strategy. Solve acyclic subproblems using the local propagation solver and strongly connected subproblems using the Newton-Raphson solver.

C1: $s - \frac{MY}{I} = 0$
C2: $M - FL = 0$
C3: $I - \frac{WH^3}{12} = 0$
C4: $Y - \frac{H}{2} = 0$
C5: $K - \frac{3EI}{L^3} = 0$
C6: $P - \frac{FL^2}{EI} = 0$
C7: $P = 10$
C8: $M = 20$
C9: $E = 30$
C10: $K = 40$
C11: $s = 50$

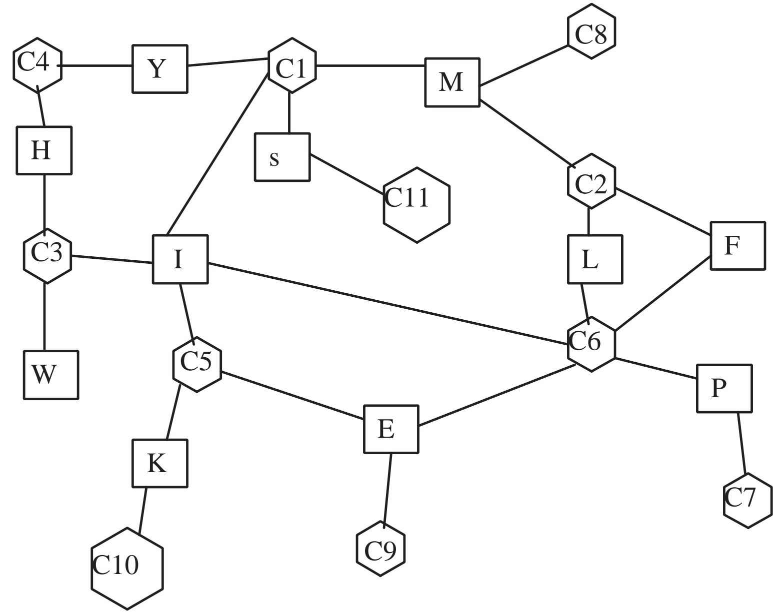


Figure 3.4: Constraint/Entity graph of figure 2.9

For example, consider the constraint problem in figure 3.4, where each entity has domain $\mathbb{R}$. INCES decomposes this to constraint problems P_1, P_2, P_3 and P_4 as shown in figure 3.5. The order of solution is given by the partial order:

$$\begin{aligned} P_1 &< P_3, \\ P_3 &< P_4, \\ P_2 &< P_4. \end{aligned}$$

The ordering strategy used by INCES is to solve subproblems that fix entities first and then use the results from these subproblems to solve connected subproblems. In figure 3.5, problems P_1 and P_2 are fixed and so are solved first. Problem P_4 cannot be solved before P_3 as it is dependent on P_3. Consequently, the ordering above is formed.

Problem P_3 is strongly connected and so is solved using Newton-Raphson. Problems P_1, P_2 and P_4 are solved using local propagation. Thus INCES is a hybrid of the domain specific local propagation algorithm and the domain general numerical solver.

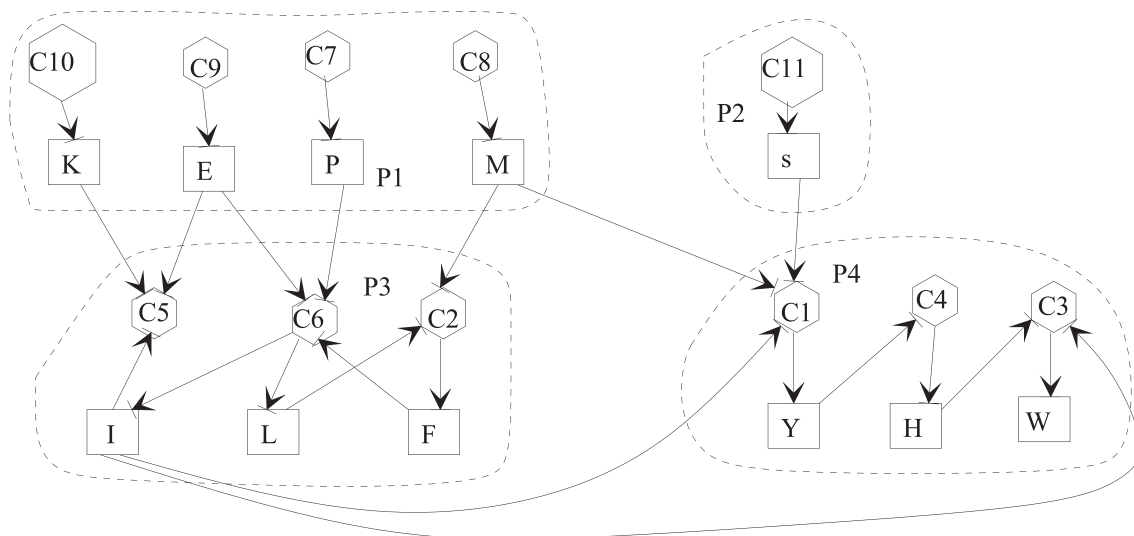


Figure 3.5: Decomposed subproblems of figure 3.4

3.1.3 IGCS

Like INCES, IGCS [112] was developed at the University of Leeds. IGCS was designed by Tsai as a geometric constraint solver that would serve as a successor to Fa's ICBSM [27]. ICBSM was limited in that it could not solve geometric constraint problems with loops and so IGCS was built with this in mind.

IGCS consists of three separate solution techniques for a geometric constraint problem: Allowable Motion, Locus Analysis and inverse operation. The Allowable Motion method is equivalent to ICBSM's Allowable Motion. This technique satisfies a new constraint by manipulating an object using its Allowable Motion (how it can translate and rotate in space). The Allowable Motion can then be used to maintain constraints when the model is manipulated.

Locus Analysis is as described by Kramer [59]. The Allowable Motion of two geometric objects that have a constraint between them are examined. Each geometric object has a locus of possible positions it can occupy without breaking its current constraints. The intersection of the two loci is the set of possible positions the two geometric objects can occupy and satisfy the new constraint as well as all of the old ones.

The final solution technique is the inverse operation method. This technique tries to satisfy a new constraint between objects A and B by examining the Allowable Motion and loci of objects connected to A and B. If the new constraint cannot be satisfied by manipulating A and B alone, then the inverse operation method attempts to satisfy the constraint by manipulating objects connected to A and B

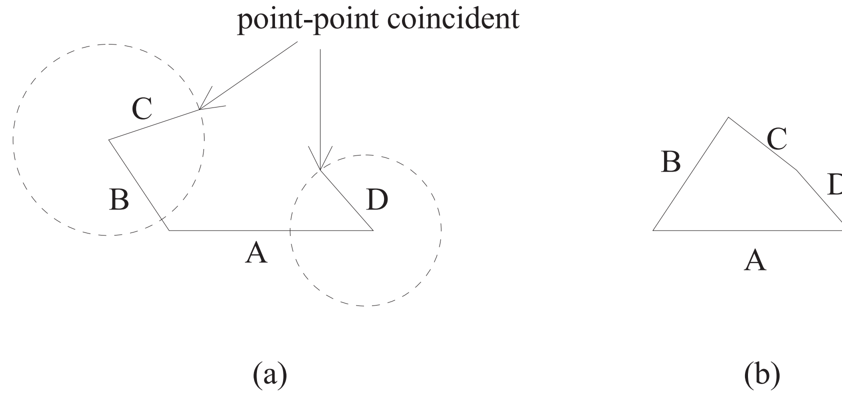


Figure 3.6: The inverse operation method in IGCS (from [112])

that may give A and B more freedom.

For example, consider figure 3.6 (a). A point-point coincidence constraint is added between the end-points of C and D. This cannot be satisfied by moving C and D alone. However, line B is free to move and so the inverse operation method will satisfy the new constraint by rotating B to a position where C and D can be rotated to solve the constraint (see figure 3.6 (b)).

The algorithm adopted by IGCS is that it tries Allowable Motion to satisfy a new constraint first. If this fails, then it tries locus analysis and if locus analysis fails, then it tries the inverse operation method. Again, this is an example of the decomposition framework:

1. Decompose a constraint problem into subproblems consisting of individual constraints and the objects associated with them.
2. Order the subproblems according to the user's interaction.
3. Solve the subproblems by applying Allowable Motion, then Locus Analysis and then the inverse operation method in order.

Note that IGCS can solve loops in the constraint graph by using the locus analysis and inverse operation methods. Note also that IGCS is an incremental solver and can deal with single constraints being added very quickly.

3.1.4 Connectivity Analysis

Connectivity Analysis [67] was presented by Latham and Middleditch as a means of solving geometric constraint problems. Of the three algorithms presented by Latham and Middleditch, one discusses the identification of over- and under-constrainedness

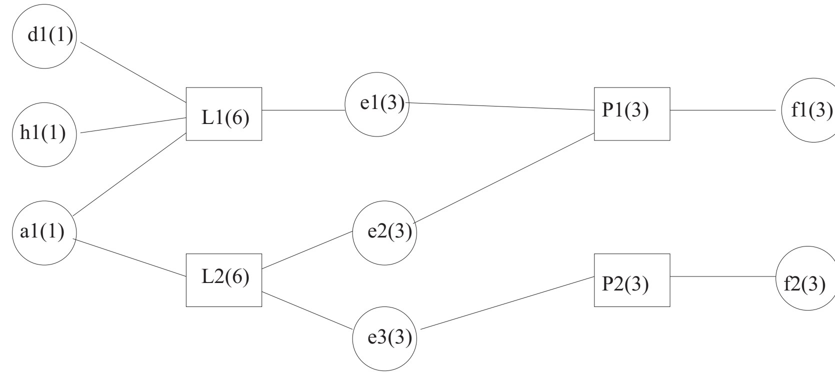


Figure 3.7: A connectivity graph for constraint problem P

of constraint problems and making over- and under-constrained problems well-constrained. The second algorithm uses constraint priorities to choose constraints to remove to solve over-constraint. The third algorithm subdivides a large set of constraints into small subproblems that can be resolved independently. It is this final algorithm that has the most direct bearing on this thesis.

Connectivity Analysis describes a constraint problem P using a bipartite connectivity graph (see figure 3.7 taken from [67]). Each entity has a number associated with it which is the dimension of the entity. Each constraint also has its dimension associated with it.

The entities in the problem are:

$L1$ and $L2$ are each line segments with dimension 6,

$P1$ and $P2$ are each points with dimension 3.

The constraints are:

$f1$ and $f2$ are constraints to fix the location of $P1$ and $P2$,

$e1, e2$ and $e3$ are endpoint constraints to fix the endpoints of $L1$ and $L2$,

$d1$ fixes the length of $L1$,

$a1$ forces $L1$ and $L2$ to be orthogonal,

$h1$ forces $L1$ to be horizontal.

Each edge of the connectivity graph has a weight associated with it, such that the sum of the weights of all the edges incident to a node is not more than the dimension of the corresponding entity or constraint. Each edge is also directed. If the weight of the edge is zero, then the direction of the edge is from an entity node to

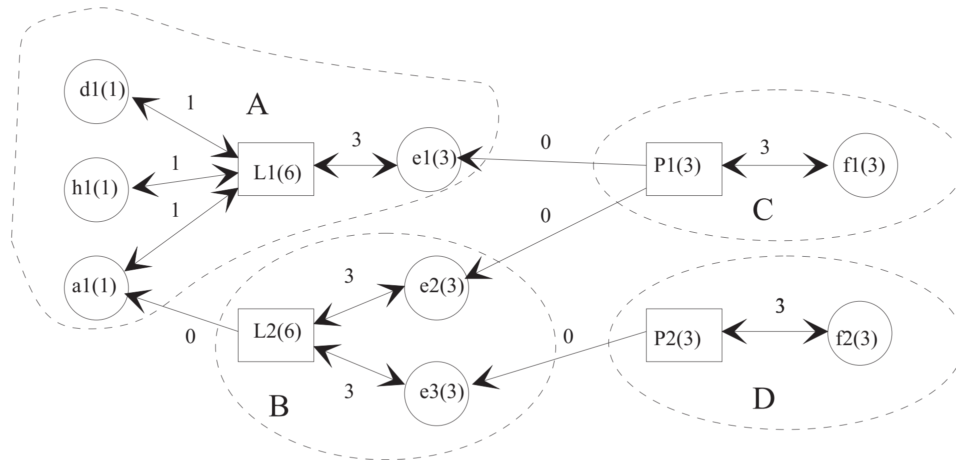


Figure 3.8: Residual sets for constraint problem P

a constraint node. If the weight is non-zero, the edge is directed in both directions.

Connectivity Analysis then proceeds by identifying a *maximal weighting* of the connectivity graph. A maximal weighting is a weighting where the sum of weights on the edges is no smaller than the sum of weights in any other weighting. At this point, Connectivity Analysis detects *residual sets* which are strongly connected sets that are not strict subgraphs of any other strongly connected sets. Figure 3.8 shows the residual sets of figure 3.7. Circles represent constraints and rectangles represent entities.

The Connectivity Analysis algorithm is presented below:

```
BEGIN
  Partition the constraint graph into residual sets
  Compute the partial order for the residual sets
  FOREACH residual set in order
    Solve the constraints in the set, treating external entities
    as constants
  ENDFOR
END
```

The Connectivity Analysis algorithm is dominated by the time taken to identify the residual sets, which is nonlinear. The partial order of residual sets is that residual set Y is solved before residual set Z, $Y < Z$, if and only if:

$$\exists y \in Y, z \in Z \text{ such that there exists a path } y \rightsquigarrow z \in E$$

where E is the set of edges in the connectivity graph. Consequently, in figure 3.8,

the partial order of residual sets is:

$$D < B < A,$$
$$C < B < A.$$

The algorithm for Connectivity Analysis is essentially a specific case of that given in table 3.2. The decomposition strategy is the identification of residual sets. The ordering is given above and the solution is by solvers applied to the residual sets in the order given. Latham and Middleditch do not discuss the specific solvers applied to the residual sets in much detail but they do note that "individual balanced sets can be solved using special or general purpose algorithms" and that "special purpose algorithms are usually more efficient than general purpose algorithms". In other words, Connectivity Analysis should use domain specific solvers to solve residual sets.

Although Latham and Middleditch do not discuss hybrid constraint solvers, the partial ordering used to order the residual sets lends itself naturally to hybrid collaboration. For example, in figure 3.8 residual sets D and C can be solved in parallel and then B and then A sequentially.

If implemented in such a fashion, Connectivity Analysis would be a hybrid of domain specific solvers. Note, however, that the decomposition strategy used in Connectivity Analysis does *not* allocate solvers to subproblems.

Recently Latham has generalised Connectivity Analysis to *type analysis* [68]. Type analysis associates a *type* with each constraint and decomposes a constraint problem into subproblems that can usually be satisfied independently of other problems. Here "usually" means not considering degenerate cases. Type analysis finds complete constraint sets with respect to a certain type T. A constraint set is *complete* if the imposed set of entities is complete with respect to T.

A set of entities is complete with respect to T if

1. A new type T constraint is compatible with some sequence of entities in that set.

2. No sequence of entities in that set can have their allowable motion reduced by a new type T constraint.

DCM [86], GCE [58] and Erep [14] all exploit distance-complete sets, i.e. subproblems that are well-constrained if only distance constraints are considered. Type

analysis detects other complete sets and so is more general. However, the time complexity of type analysis is $O(n^3)$, where n is the number of constraints in a problem. Correspondingly, type analysis is not suitable for the type of scaleable, interactive application of interest to the Virtual Working Environment group at Leeds. Note that type analysis also does not consider the allocation of solver to subproblem, although the use of types helps to narrow the choice of solvers appropriate to a particular subproblem.

3.2 Decomposition strategies

A decomposition strategy De is a function that takes a constraint problem and a set of available solvers and produces a set of subproblems with solvers assigned to them. It is formally defined here.

Definition 3.1 (Decomposition Strategy) *A decomposition strategy is a function*

$$De : (P, \mathcal{S}) \longrightarrow \{(P_i, \mathcal{S}_i)\},$$

where P is a constraint problem, P_i are subproblems of P, $\mathcal{S}$ is a set of constraint solvers and $\mathcal{S}_i \subseteq \mathcal{S}$. □

A decomposition strategy can be simple or complex. One of the most basic decomposition strategies that is employed by virtually every constraint solver is :

$$De_{connected\ components} : (P, \mathcal{S}) \longrightarrow \{(P_i, \mathcal{S})\},$$

where the $\{P_i\}$ are the connected components of the Constraint/Entity graph of problem P. Elements of a connected component do not affect any elements of a different connected component, by the definition of imposed sets (see definition 4.9). $De_{connected\ components}$ applies the same set of solvers to the connected components of P.

Decomposition strategies are important because they divide a large and complex problem into a number of smaller problems that can be more easily dealt with. However, there are a number of difficulties associated with decomposition. These are discussed in more detail in section 3.2.4.

The remainder of this section presents some examples of decomposition strategies in section 3.2.1. Section 3.2.2 discusses decomposition strategies to take advantage of

domain specific solvers. Section 3.2.3 notes the advantages of decomposition strategies, whilst section 3.2.4 acknowledges the limitations and dangers of this approach. Section 3.2.5 discusses the use of decomposition strategies in incremental constraint solvers. Section 3.2.6 draws some conclusions from the use of decomposition strategies.

3.2.1 Examples of decomposition strategies

Most constraint solvers use decomposition strategies in order to find solutions to constraint problems. Note that a further decomposition strategy can be applied to each of the results of a decomposition strategy and so decomposition strategies can be *composed*. For example, Serrano's DESIGNPAK [100] decomposes problems into strongly-connected components† and non-strongly-connected components. Call this decomposition strategy $De_{strong\ components}$. However, prior to applying $De_{strong\ components}$, DESIGNPAK has already decomposed the constraint problem into connected components using $De_{connected\ components}$.

Thus the DESIGNPAK decomposition strategy is

$$De_{strong\ components} \circ De_{connected\ components},$$

where the $\circ$ notation is used to denote composition of decomposition strategies:

$$(De_2 \circ De_1)(P, \mathcal{S}) = \bigcup_i \{De_2(P_i, \mathcal{S}_i)\},$$

where

$$De_1(P, \mathcal{S}) = \{(P_i, \mathcal{S}_i)\}.$$

Erep [14] uses a decomposition strategy that is very similar in outcome to DCM but somewhat different in application. Erep identifies *clusters* of entities and constraints. A cluster is a part of a constraint problem that can be fully described as a rigid body and can be manipulated as a rigid body. Erep has two distinct phases to constraint solution. First it builds clusters by positioning an entity relative to two fixed entities using two constraints. This procedure continues until no more entities can be placed relative to any pair of entities in the cluster. At this point, the cluster is a rigid body. Clusters can then be placed relative to each other using manipulation of the rigid bodies. Three clusters $Cl1, Cl2$ and $Cl3$ can be placed

relative to each other if there exist three entities, $x \in Cl1$, $y \in Cl2$ and $z \in Cl3$ and three constraints, between x and y, y and z and z and x.

The decomposition strategy used by Erep decomposes a constraint problem into a number of clusters, which are solved using the cluster building technique, and sets of three clusters that can be positioned relative to each other.

The decomposition strategies used by INCES, IGCS, DCM and Connectivity Analysis have been discussed in detail in section 3.1.

3.2.2 Decomposition to domain specific subproblems

In order to take advantage of the power of domain specific solvers, a constraint solver must decompose a constraint problem into subproblems that can be solved efficiently by the domain specific solvers available. Such subproblems are referred to here as domain specific subproblems. In order to perform this decomposition, the constraint solver must know the strengths of the domain specific solvers and how the appropriate domain specific subproblems can be identified within the original constraint problem. This is very difficult to do thoroughly.

In order to find domain specific subproblems, it is first necessary to identify the domain specific knowledge being used by the domain specific solvers. This helps to identify types of problem that the domain specific solvers can solve efficiently. This in turn leads to methods of pulling out domain specific subproblems from the original constraint problem.

For example, the domain specific knowledge that Erep and DCM use is ruler-and-compass construction. More specifically, Erep and DCM use the domain specific knowledge that distance and angle constraints between points and lines can be solved quickly and efficiently because a point is fixed in space by two distance constraints relative to two other fixed points and a number of similar rules. Erep and DCM exploit such domain specific knowledge by using the constraint graph of the constraint problem to identify clusters of entities that form rigid bodies. Clusters correspond to constraint graphs with no articulation pairs[†] and both DCM and Erep exploit constraint graphs with this structure in order to identify ruler-and-compass subproblems.

IGCS, ICBSM and GCE use domain specific knowledge related to rigid bodies. They use the knowledge that rigid bodies are not deformable and can only translate, rotate and scale in space once created. Rigid body subproblems are characterised by 3D and 2D objects, such as lines, planes, cuboids and spheres with high level

against, coincidence and *concentric* constraints.

Finite domain subproblems are characterised by entities and constraints with finite domains. Scheduling subproblems are characterised by entities with a time domain and constraints such as *before, after* or *concurrently*.

INCES uses the domain specific knowledge that functional constraint problems that have a tree graph structure can be solved using local propagation.

A decomposition strategy to take advantage of these domain specific subproblems will be dependent on the solvers available and the problems to be solved. The advantages and limitations of such decomposition strategies are discussed in the following sections.

3.2.3 Advantages of decomposition strategies

There are several advantages to using decomposition strategies in general and to domain specific decomposition strategies in particular.

1. Smaller problems are simpler to deal with.
2. Decomposing and solving is usually faster than dealing with problems as a whole.
3. Decomposition reuses existing constraint solvers.
4. Domain specific decomposition takes advantage of very efficient solvers.

The final point is the key issue of interest in this thesis. The advantages are covered in more detail below.

1. When solving large and complex problems, it is difficult to comprehend the whole problem all at once. Small parts of the problem are easier to deal with and understand, both for the human operator and the computer. It is significantly easier to spot patterns and problems in small problems than large ones.
2. In a similar fashion, it is much easier, and therefore quicker, to find solutions to small problems than large problems. However, actually decomposing the constraint problem may take significantly longer than finding the solutions to the subproblems if a complex decomposition strategy is adopted. Consequently, it may be faster to solve the constraint problem as a whole.

3. Since a constraint problem is decomposed to subproblems that are consequently solved using specialist solvers, the decomposition strategy can be designed in such a way as to utilise existing solvers. This means that the power and expressibility of old solvers can be increased at little cost in developing new code, beyond developing the decomposition strategy.

4. Domain specific decomposition is a particular type of decomposition strategy that tries to take advantage of powerful domain specific solvers. As noted in chapter 2, domain specific solvers are very efficient at solving some constraint problems. Consequently they should be reused and adapted if possible. Domain specific decomposition provides a means of doing this. Domain specific decomposition is a particular example of the divide and conquer strategy prevalent in computer science.

3.2.4 Limitations of decomposition strategies

In fact most of the advantages of decomposition can also be interpreted as disadvantages. Limitations of decomposition and domain specific decomposition include:

1. Small parts of a greater whole frequently provide no information about the greater whole.

2. Decomposition can fail to find solutions where some exist.

3. Decomposition can be slower than solving a problem as a whole.

4. Domain specific decomposition may lead to subproblems that cannot be decomposed further and cannot be solved by any existing domain specific solver.

5. It may be extremely hard to identify the domain specific knowledge a domain specific solver uses.

6. It may be extremely hard to identify domain specific subproblems within a constraint problem.

7. Recombination of solutions to subproblems into solutions to the combined problem is hard.

These limitations are discussed in more detail below:

1. Studying small parts of a problem does not necessarily give any insight to the whole. At one extreme, this has given rise to chaos theory. In terms of constraint problems, this means that decomposing to subproblems and then studying the subproblems does not necessarily lead to insight into the problem as a whole. This point is discussed in more detail in chapter 6 and in section 7.4.1.

2. One immediate effect of the difficulty in applying knowledge of small parts of a problem to the larger whole is that solving independent subproblems may not lead to solutions of the whole problem. This is because one of the underlying assumptions about decomposition is that there is little or no interdependency between subproblems. The classic manifestation of this in infinite-domain constraint problems is the existence or not of cycles in the constraint graph of a problem. Constraint graphs that can be structured to form a tree usually have subproblems that can be solved independently and in a certain order. Constraint graphs that contain cycles have constraints that are dependent on each other and must be solved simultaneously. It is sometimes possible to develop decomposition strategies whereby dependent subproblems can be put into one subproblem which is solved as a whole. This behaviour is apparent in Serrano's DESIGNPAK [100] and Lamounier's INCES [62]. However, this only really sweeps the problem under the carpet, as the dependent subproblem to be solved simultaneously now forms the bulk of the effort necessary to solve the constraint problem.

3. The more complex a decomposition strategy and the smaller the subproblems it produces, the faster solvers can find solutions to the subproblems. However, the bulk of the effort of the general constraint solver has simply been shifted from the solvers to the decomposition strategy and little has been gained. Conceptually, this is like the graph in figure 3.9. As more effort goes into the decomposition strategy, less effort needs to be put into the solution of the subproblems. At a certain point however, the decomposition strategy takes more effort than the solution of the subproblems. It is at the trade-off point, where equal effort is made by the decomposition strategy and the solvers, that the most efficient general constraint solver should exist.

   No effort is made here to discuss the shape of the curves nor where the trade-off point may be found, as this is beyond the scope of this thesis. It is simply noted that such a point will probably exist.

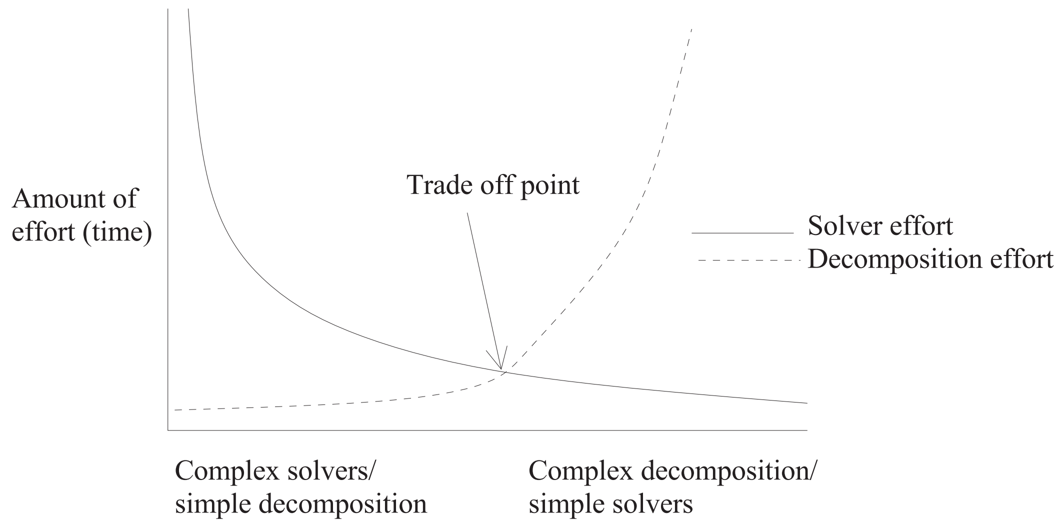


Figure 3.9: Graph of trade-off between complex decomposition and complex solvers

4. Decomposing to domain specific subproblems may lead to a situation whereby a subproblem is highly interdependent and can be decomposed no further. However, the subproblem is not suitable for any particular domain specific solver and so cannot be assigned to a domain specific solver. Such a situation is dependent on the decomposition strategy used and may be countered by having a backup, domain general, solver that can be used to solve the subproblem. However, if the non-decomposable subproblem is large or a significant proportion of the whole problem then solution of the constraint problem as a whole will be dominated by solution of the subproblem by the domain general solver, which will typically be very slow. This situation is evident in, for example, COSAC [85], where constraint problems with few linear equations will be solved by the slow Gröbner basis solver.

5. Domain specific knowledge is a fairly nebulous concept. Some examples of the use of domain specific knowledge are presented in this thesis but these are by no means comprehensive. However, it will not always be easy or even possible to identify the domain specific knowledge employed by a particular solver. For example, what is the domain specific knowledge used by genetic algorithms? If the domain specific knowledge used by a solver cannot be specified, then it will be extremely hard to characterise the type of problems that a particular solver can deal with. For example, what type of problems are genetic algorithms particularly well-suited to solving?

6. Given characterisations of the type of problems domain specific solvers can

handle, it may still be extremely hard to identify domain specific subproblems within a constraint problem. Latham identifies an aspect of this problem as the difference between identifying graph properties and real properties [68]. For example, the identification of ruler-and-compass constructible subproblems is difficult without performing the decompositions used by Erep and DCM. This problem is discussed in more detail in section 8.

7. A side effect of decomposition strategies is the need for recombination of the solutions to subproblems into solutions of the combined problem. In effect, the recombination is a constraint problem in its own right and is typically hard to do. This problem is discussed in more detail in chapter 3.4.

3.2.5 Incremental issues in decomposition strategies

An incremental constraint solver adds a set of constraints and entities to a previously solved constraint problem and tries to find solutions to the augmented constraint problem by reusing the existing solution as much as possible.

Most of the work carried out in incremental constraint solution takes place in the decomposition strategy of the incremental solver. Recall that a decomposition strategy takes a constraint problem and a set of constraint solvers and splits the constraint problem into a number of subproblems which have constraint solvers associated with them.

The decomposition strategy of an incremental solver will start with no constraints and no entities. A set of constraints and entities is then added and these are decomposed into subproblem-solver pairs. When a new set of constraints and entities is added, a new set of subproblem-solver pairs is constructed. However, the incremental solver will try to reuse the existing decomposition as much as possible.

INCES [62] adopts this incremental approach. The decomposition strategy in INCES splits the constraint problem into connected components that are either strongly connected or acyclic. If a new entity is added then it effectively forms its own (disconnected) constraint subproblem. If a new constraints is added then one of several possibilities occur:

1. The constraint is connected to entities in a previously acyclic subproblem that remains acyclic when the constraint is added. In this case the constraint is simply added to the acyclic subproblem which is resolved.

2. The constraint is connected to a previously acyclic subproblem that becomes

strongly connected when the constraint is added. In this case, the constraints and entities in the new strongly connected subproblem are put into a new strongly connected subproblem and the remaining constraints and entities form a new acyclic subproblem.

3. The constraint is connected to a previously strongly connected subproblem that remains strongly connected when the constraint is added. In this case the constraint is added to the strongly connected subproblem.

4. The constraint is connected to a previously strongly connected subproblem that is no longer strongly connected when the constraint is added. In this case, a new acyclic subproblem consisting of only the new constraint is created and added to the subproblems found by the decomposition strategy.

Thus, when a new constraint is added, very little new decomposition need be done.

IGCS [112] is also an incremental solver. However, the incremental nature of IGCS comes from the fact that it tries to solve a newly added constraint by manipulating the model without breaking any of the existing constraints. IGCS works by trying to satisfy the new constraint by first manipulating the objects identified by the new constraint directly using Allowable Motion and locus analysis. If that does not work then the inverse operation method applies the same techniques to objects that are upstream in the constraint graph. IGCS has a very simple decomposition strategy and the incremental decomposition strategy is exactly the same. All of the incremental work is done in the solution stage.

DCM [86] (and by implication Erep [14]) is *not* incremental. DCM decomposes constraint problems to triconnected components that represent rigid bodies. Thus DCM can only solve problems that consist of rigid bodies. By its nature, incremental solution adds constraints and entities one by one. Thus, adding a new entity leads to an under-constrained problem and adding new constraints gradually makes it more and more constrained until it becomes well-constrained.

Since rigid bodies are always well-constrained up to rigid body motion, DCM can only solve well-constrained constraint problems. However, there are two ways that it may be possible to make DCM at least partially incremental.

Incremental constraint solution adds a number of constraints and entities to a previously solved constraint problem. DCM can only solve well-constrained constraint problems. Thus, if a number of constraints and entities were added to a well-constrained problem to give another well-constrained problem, then DCM would be able to solve both the original problem as well as the new problem.

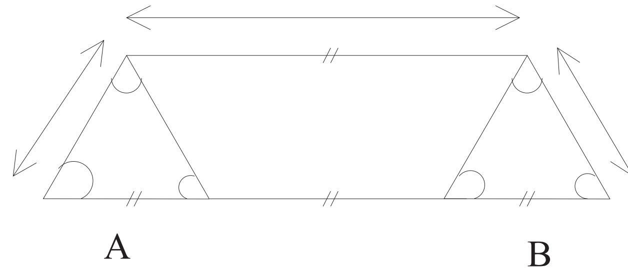


Figure 3.10: An rigid body composed of two triangles

For example, triangle A in figure 3.10 is well-constrained and DCM can solve the constraint problem defining the triangle. Similarly, the whole figure is well-constrained and so DCM can solve the associated constraint problem. Thus, DCM could solve for triangle A first and then, when the remaining elements in figure 3.10 are added, DCM can solve for the figure as a whole, incrementally.

This incremental DCM could reuse the decomposition found for triangle A when solving for the whole problem and thus save some effort.

However, this approach is difficult because the user will have to add a number of constraints and entities before being able to resolve the problem and getting feedback from the solver. Though Erep is not incremental in the sense described here, the method of defining constraint problems is the same and the author's experience with Erep underlines the difficult in defining constraint problems using this process.

Also, in order to make the whole figure well-constrained, a large number of constraints and entities need to be added and the subproblem that is triangle A becomes a very small part of the whole problem. Consequently there has been little gain in keeping the decomposition of triangle A.

The second method of making DCM incremental is of particular interest in this thesis. DCM is very good at finding solutions to well-constrained geometric constraint problems. However, it cannot solve under-constrained geometric constraint problems and so cannot find a solution when a single constraint is added to an under-constrained problem. On the other hand, IGCS *can* find solutions when a single constraint is added to an under-constrained problem. Gradually, the under-constrained problem has more and more constraints added to it and becomes well-constrained. At this point, IGCS is not particularly well-suited to solving the problem, but DCM is.

The two techniques therefore complement each other very well and an obvious question is whether it is possible to combine the two techniques in such a way as to get the benefits of both. Investigating this possibility is one of the main objectives

of this thesis.

3.2.6 Conclusions

Decomposition has proved a powerful means of tackling problems in computer science. As demonstrated by the constraint solvers in section 3.1, decomposition allows constraint problems to be solved quickly and efficiently. Generally, the advantages of decomposition far outweigh the disadvantages.

In particular, decomposing a constraint problem can allow problems that would take exponential time to solve using a general constraint solver to be solved in polynomial time.

The limitations of decomposition strategies are serious however. Whilst this thesis does not attempt to solve all of these problems, it does tackle some of them. In particular, chapters 6 and 7 examine the difficulties of using decomposition strategies to create and solve subproblems. Chapters 6 and 7 identify the key areas that decomposition strategies must address in order to find solutions to the constraint problem.

Chapter 8 examines the use of a simpler decomposition strategy with more complex constraint solvers.

3.3 Ordering strategies

When a constraint problem has been decomposed to a number of subproblems, it remains to solve the subproblems and recombine the solutions to the subproblems into solutions to the whole problem. However, the *order* in which the subproblems are solved in is frequently important. For example local propagation techniques such as INCES [62], ICBSM [27] and SkyBlue [94] decompose a constraint problem into a triangular format:

Example 3.1 (Ordering strategies) Consider constraint problem P,

$$P = (\{(x, \mathbb{R}), (y, \mathbb{R}), (z, \mathbb{R})\}, \{x = 1, x + y = 2, x + y + z = 3\}).$$

Let the subproblems be:

$$P_1 = (\{(x, \mathbb{R})\}, \{x = 1\}),$$
$$P_2 = (\{(x, \mathbb{R}), (y, \mathbb{R})\}, \{x = 1, x + y = 2\}),$$
$$P_3 = (\{(x, \mathbb{R}), (y, \mathbb{R}), (z, \mathbb{R})\}, \{x = 1, x + y = 2, x + y + z = 3\}).$$

The order of solution of P_1, P_2, P_3 has significant repercussions on the ease of solving P. Solving in the order (P_3, P_2, P_1) using local propagation is much harder than solving in the order (P_1, P_2, P_3). $\square$

Consequently, most constraint solvers use an *ordering strategy* on the subproblems once they have been decomposed. The ordering strategy produces a partial order $<$ on subproblems, such that $<$ is irreflexive, antisymmetric and transitive.

Ordering strategies are dependent on the constraint solver used and the problems being solved. Consequently, this section does not attempt to present a comprehensive study of ordering strategies. Section 3.3.1 gives examples of ordering strategies in use in current constraint solvers. Section 3.3.2 discusses the use of ordering strategies in general constraint solvers and how hybrid constraint solvers and domain specific knowledge can be used to help formulate an ordering strategy. Section 3.3.3 discusses the use of ordering strategies in incremental solvers. Section 3.3.4 draws conclusions from the use of ordering strategies.

3.3.1 Examples of ordering strategies

Fa's ICBSM [27] forces an ordering strategy on the user by means of the Relationship Graph (see section 5.3.2). The Relationship Graph is a directed constraint graph that forces one constraint to be solved before another and passes allowable motion from one entity to another. Consider for example the construct in figure 3.11 with the associated Relationship Graph.

In constructing the figure, line 1 is fixed at one end and its allowable motion calculated to be rotation around the fixed end. Line 2 is then attached and its allowable motion calculated to be rotation around the other end point of line 1. Similarly line 3 when attached is allowed to rotate about the end point of line 2. Equally, direct manipulation of line 1 is propagated to line 2, following the directions in the Relationship Graph. The directed edges in the Relationship Graph form the ordering strategy of ICBSM. In ICBSM, an edge is directed from A to B if the user directs that A is the reference and B the target entities of a constraint. Consequently

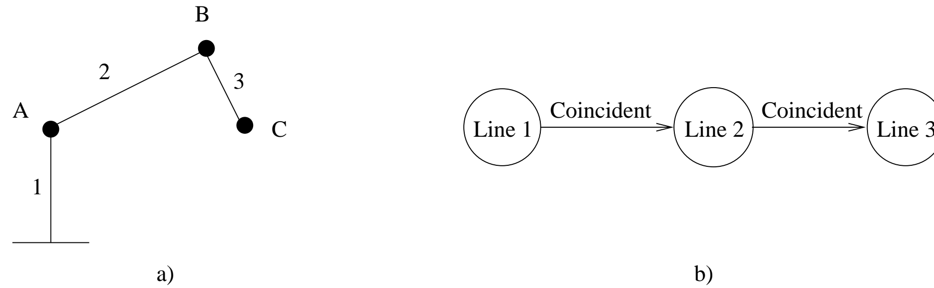


Figure 3.11: An Arm with Two Joints and the Relationship Graph for the Problem

the ordering strategy in ICBSM is dictated by the user and this tends to place an additional burden on the user.

Serrano's DESIGNPAK [100] uses an ordering strategy of proceeding from known to unknown entities. The general principle that Serrano advocates is "the manipulation of symbols (knowledge) in order to derive (infer, conclude) new facts from existing (known) facts". Effectively, this means solving "simple" (in some sense) subproblems and using the information derived from solving the simple subproblems to solve the more difficult subproblems later. This principle manifests itself by having fixing constraints of the form '$x = 10$' that fully define an entity and make it 'known'.

The ordering strategies employed by IGCS, INCES, DCM and Connectivity Analysis have been discussed in detail in section 3.1.

3.3.2 Ordering strategies for a constraint solver

A constraint solver using domain specific knowledge and hybrid constraint collaboration should also take advantage of ordering strategies in order to improve the efficiency of solution.

The ordering strategies given in sections 3.3.1 and 3.1 can be summarised as

1. User interaction,

2. Find solutions to easy problems before harder ones,

3. Simultaneous subproblems should be as small as possible.

All of the above strategies can be usefully employed in any ordering strategy. In particular, the strategies become particularly useful if domain specific knowledge is employed.

User interaction is a useful ordering strategy because it allows the user to state preferences for solutions, albeit indirectly. Unfortunately, user interaction places an additional burden on the user in an already complicated procedure. The interaction in ICBSM forces the user to consider not only which shapes and which constraints are necessary to construct a model, but also in which order to add the constraints. Since different orders of construction might lead to different and unintuitive solutions, this is clearly inadvisable.

This problem is somewhat offset by using rules to determine the order of construction and solution and to determine which solution the user intended. For example, DCM [86], Erep [14] and ICBSM [27] try to interpret the user's intent in order to determine which of many solutions the user requires.

It is sensible to solve easy problems before harder ones because solution of the easier problems might simplify the harder ones significantly. However, the definition of 'easy' and 'hard' are domain specific and consequently domain specific knowledge must be brought to bear in order to identify the 'easy' and 'hard' problems. Consequently, the ordering strategy must employ a great deal of domain specific knowledge.

INCES identifies 'fix' constraints that fix a variable with a certain value. Such constraints are trivial to solve and so form the 'easy' subproblems. Fixing the value of the variable allows other constraints to be solved more easily. DCM and Erep use much more sophisticated knowledge to identify 'easy' subproblems. The 'easy' subproblems in DCM and Erep are the triangles identified by the cluster building algorithm. Such triangles can be solved using simple calculations and so are considered to be the simplest class of problems that DCM and Erep identify. Note that finding the triangles and then identifying the order in which they should be solved is a complex process and uses the domain specific knowledge that a point or line can be fixed relative to two other points or lines using two constraints.

Tsai has identified the need to make simultaneous subproblems as small as possible. Since simultaneous subproblems will be the most time-consuming to solve, it is advantageous to keep them as small as possible. Note that it will not usually be possible to limit the size of simultaneous subproblems and so this technique has only limited potential.

3.3.3 Incremental issues in ordering strategies

Once an incremental solver has decomposed a constraint problem to subproblem-solver pairs, the incremental solver will usually apply an ordering strategy to the subproblem-solver pairs as normal. However, there are a couple of issues that an incremental ordering strategy should take into account.

The first is that the ordering strategy should not try and calculate the order from scratch but should reuse the ordering previously calculated. This should save some effort.

Secondly, the ordering strategy should be aware that a newly inserted set of constraints and entities may only affect a small part of the constraint problem and thus solvers need only be applied to a small part of the constraint problem in order to find solutions to the new constraint problem.

For example, INCES orders subproblems so that solution proceeds from the first subproblem to the last. Thus if the decomposition gives:

$$(P_1, \mathcal{S}_1), (P_2, \mathcal{S}_2), (P_3, \mathcal{S}_3), (P_4, \mathcal{S}_4), (P_5, \mathcal{S}_5),$$

P_1 is solved before P_2 before P_3, etc.

If a new constraint is added and is incorporated into P_3, then the nature of INCES is such that the solutions of P_1 and P_2 are still valid solutions to the new constraint problem and need not be resolved. Thus, the ordering strategy needs to note that only P_3, P_4 and P_5 need to be resolved.

3.3.4 Conclusions

The main advantage of an ordering strategy is that it can be used to help guide solution of a constraint problem and hence speed up solution of the constraint problem. With respect to hybrid constraint solvers, an ordering strategy can be used to describe an appropriate collaboration. For example, suppose that an ordering strategy produces an ordering $<$ on subproblem-solver pairs

$$\{(P_1, \mathcal{S}_1), (P_2, \mathcal{S}_2), (P_3, \mathcal{S}_3), (P_4, \mathcal{S}_4)\}$$

such that

$$(P_1, \mathcal{S}_1) < (P_2, \mathcal{S}_2) < (P_3, \mathcal{S}_3),$$
$$(P_1, \mathcal{S}_1) < (P_4, \mathcal{S}_4)$$

then the subproblems can be solved in a serial fashion in the order

$$(P_1, \mathcal{S}_1),\ (P_2, \mathcal{S}_2),\ (P_3, \mathcal{S}_3),\ (P_4, \mathcal{S}_4).$$

The limitations of ordering strategies are more dependent on their implementation than the concept. For example, the Relationship Graph ordering used in ICBSM forces the burden of deciding in which order to build a model onto the user and is an extra complication from a system that is trying to simplify the process.

Inadequate ordering strategies may lead to orderings of subproblems that do not simplify solution of the constraint problem, but instead complicate it, as seen in example 3.1. Care should be taken when selecting an ordering strategy to ensure that it is not too sensitive to the structure of the constraint problem and is applied correctly.

3.4 Solution of subproblems

Under the constraint solver framework introduced in this chapter, a constraint problem P is decomposed to a set of subproblem-solver pairs $\{(P_i, \mathcal{S}_i)\}$. The set of subproblem-solver pairs is then ordered using an ordering strategy to give a partial ordering $<$ on the subproblem-solver pairs. It remains to solve the subproblems and to use the solutions of the subproblems to find solutions to the original problem P.

The set $\mathcal{S}_i$ associated with subproblem P_i in the set of subproblem-solver pairs is a set of solvers that have been identified by the decomposition strategy as being suitable candidates for solving P_i. Usually $\mathcal{S}_i$ will contain only one candidate and consequently choosing the solver to apply is trivial. However, it is possible that a constraint solver will identify a number of potential subsolvers that can be applied equally well. If this is the case, then a subsolver needs to be selected at this point for application.

The solution and recombination of subproblems using hybrid collaboration will be discussed in detail in chapter 7.

Section 3.4.1 gives examples of constraint subsolvers and the solution of subproblems. The use of domain specific subsolvers to solve the subproblems is discussed in section 3.4.2. Section 3.4.3 discusses the solution of subproblems in incremental solvers. Conclusions are drawn from the solution of subproblems in section 3.4.4.

3.4.1 Examples of solution of subproblems

In ICBSM [27], subproblems are not decomposed *per se*. For each constraint there is a subproblem consisting of that constraint and the imposed entities of the constraint. Consider, for example, constraint aCb, where the ordering imposed by the Relationship Graph is $[a, b]$. Consequently, a is the *reference* of the constraint and b is the *target*. Solution of the subproblem $(\{(a, \mathbb{R}^6), (b, \mathbb{R}^6)\}, \{aCb\})$ is by examining the allowable motion of the target entity and using that allowable motion to try and satisfy the constraint. If b can be translated or rotated to satisfy constraint C, then the new allowable motion of b is dependent on C and the previous allowable motion of b. The solver used to solve the subproblem finds the allowable motion of b and the translation or rotation necessary to satisfy the new constraint. The solver then applies the translation or rotation to b and then updates the allowable motion of b.

DESIGNPAK [100] decomposes a constraint problem to subproblems that are either cyclic or acyclic. Cyclic subproblems are solved by a numerical technique such as Newton-Raphson. Acyclic subproblems are solved using local propagation. Consequently, DESIGNPAK consists of two constraint subsolvers, Newton-Raphson and local propagation. Since both subsolvers produce only one solution, recombination of solutions is trivial.

COSAC [85] effectively decomposes constraint problems into subproblems that consist of linear equations, and so can be solved using simple Gaussian elimination or similar subsolvers, and subproblems that cannot be solved using a linear equation solver, which are solved using Gröbner bases. Generally, the Gröbner basis subsolver will dominate the time to solve a constraint problem as most constraint problems will not consist of many linear equations. Whilst COSAC is mostly a Gröbner basis solver, it is the first hybrid solver designed using Monfroy's hybrid collaboration language [84].

3.4.2 Solving using domain specific knowledge

In order to take advantage of domain specific knowledge, domain specific solvers should be used to solve the subproblems produced by the decomposition strategy. As noted by Latham and Middleditch [67], "Special purpose algorithms are usually more efficient and reliable than general purpose algorithms, and they provide geometrically meaningful information that facilitates useful user feedback and aids selection from multiple solutions.". The special case solvers used by DCM and the allowable motion used in ICBSM are both examples of domain specific solvers. The

local propagation technique used in INCES is not domain specific as such. However, it does require a specific structure of subproblem to be successful. In fact, local propagation is generally used as a domain specific algorithm, as it is used in ICBSM specifically for acyclic geometric problems and in INCES to solve acyclic equation problems.

The hard work of identifying the domain specific knowledge used by a domain specific solver to produce subproblems appropriate for that domain specific solver is carried out by the decomposition strategy. Domain specific solvers are applied to the subproblems to produce solutions quickly and efficiently.

3.4.3 Incremental issues in solving subproblems

The use of subsolvers in an incremental constraint solver is usually the least of the three phases. Nearly always, the bulk of the work in an incremental solver is carried out by the decomposition and ordering strategies. The subsolvers generally act in the same fashion as described in this section and find solutions to the subproblems.

There are exceptions to this rule of course. IGCS, in particular, puts most of the effort of incremental solution into solving the constraints as they are added. The algorithms used by IGCS are described in more detail in section 3.1.3.

3.4.4 Conclusions

Since a subproblem must be solved in order to find solutions to the whole problem, the advantages and limitations of solving subproblems are related to the advantages and disadvantages of using domain specific solvers and hybrid constraint solvers.

However, the advantages and disadvantages of domain specific solvers and hybrid constraint solvers are not obvious. The use of domain specific solvers is discussed in section 7.1. The use of hybrid constraint solvers is discussed in section 7.2.

3.5 Conclusions

This chapter has presented a discussion of the general nature of constraint solvers and in particular the creation of constraint solvers that explicit domain specific knowledge and hybrid constraint solvers. The general outline of a constraint solver can be described simply as a sequence of three strategies:

1. A decomposition strategy takes as input a constraint problem and produces as output a set of subproblems and constraint solvers assigned to them. The

decomposition strategy is significant as it splits a large, complicated constraint problem into a number of smaller, more manageable subproblems. As noted by Latham [68], "the key to efficient constraint solution is to partition a large set of constraints into smaller sets that can be solved independently".

2. An ordering strategy takes as input a set of subproblem-solver pairs and produces as output a partial order in which the subproblems should be solved. The ordering strategy is not always necessary. However, the ordering strategy does allow a constraint solver to influence the sequence of solution of subproblems. Since the result of subproblems can be used to simplify solution of later subproblems, an ordering strategy is an important tool to aid solution of the whole constraint problem.

3. A solution strategy takes the partially ordered set of subproblem-solver pairs and initiates a solver on the subproblem, maintaining the order given by the ordering strategy. Solutions are passed from subproblem to subproblem in order to take advantage of known solutions.

This outline lends itself naturally to the use of domain specific knowledge and hybrid constraint solvers. The decomposition strategy can make use of domain specific knowledge inherent in domain specific solvers and ensure that the subproblems are selected to take best advantage of domain specific solvers available.

The ordering strategy can then be used to order subproblems in order to take advantage of domain specific solvers hybrid collaboration strategies.

The solution strategy then initiates the domain specific solvers on the appropriate subproblems and the hybrid collaboration paradigms are used to make explicit the recombination of solutions to the subproblems.

The advantages of using a decomposition strategy are that large and complex constraint problems are subdivided into a number of smaller and more manageable subproblems. This divide-and-conquer approach is frequently faster than trying to deal with a problem as a whole. Decomposing to subproblems amenable to domain specific solution also speeds up the solution process.

Conversely, decomposition strategies are subject to consistency problems as well as difficulties identifying domain specific subproblems. Identifying domain specific subproblems is also hard, because it is frequently difficult to identify the domain specific knowledge used by a domain specific solver. Once the domain specific knowledge is identified, it is hard to find domain specific subproblems that explicate that domain specific knowledge without solving the whole problem.

Ordering strategies allow solvers to help guide solution of a constraint problem in order to improve the speed of the constraint solver. Ordering strategies also lead naturally to the use of hybrid collaborations. Unfortunately, ordering strategies can be very sensitive to the ordering chosen and care needs to be taken when selecting a strategy.

Many examples of constraint solution by decomposition were given, such as DCM [86], INCES [62], IGCS [112], Connectivity Analysis [67] and Erep [14]. The strategies adopted by each such algorithm were explored in detail.

Several issues and questions were raised by study of the decomposition framework in this chapter:

- Is it possible to lose solutions by decomposing and recombining?
- What effect does decomposition have on incremental techniques?
- Is it more efficient to decompose and recombine or solve as a whole?
- Is it possible to have a fast decomposition strategy and a fast solution of subproblems or must one always dominate?

In order to investigate these questions thoroughly, it was necessary to study the constraint solution process in more detail. For this study, the decomposition of a constraint problem is not necessary and it merely complicates the procedure. The constraint solution process is split into three stages: constraint definition, constraint representation and constraint solution.

Chapter 4 addresses constraint definition and identifies the key terms and elements that comprise a constraint problem. Chapter 5 studies the use of constraint representation schemes which are methods of describing and storing constraint problems. Chapter 6 uses the definition of constraint problems from chapter 4 to build a framework for constraint solution that can be used to study the above questions.

Chapter 4

Constraint Definition

Constraints are used in many different situations. People frequently refer to constraints on their financial position, social life or personal habits. In one week in June 1997, the BIDS Uncover Service [1] found references in the following journals mentioning the word 'constraints'.

Omega.
Soil Biology and Biochemistry.
Duke Mathematical Journal.
Applied geochemistry : journal of the international Association of Geochemistry and Cosmochemistry.
Physical review. e. statistical physics, plasmas, fluids, and related interdisciplinary topics.
Tectonophysics.
Physical review d: particles, fields, gravitation, and cosmology.
International journal of control.
Journal of symbolic computation.
Journal of East Asian linguistics.
Ecology.
Pharmaceutical research.
Geochimica et cosmochimica acta.
The serials librarian.
Eclogae geologicae Helvetiae.
Physical review B: Condensed matter.

Such a wide variety of subject areas and uses means that there are many different definitions of a constraint, with different means of describing and dealing with them.

Even in the somewhat more limited context of this thesis, such diversity is evident. In fact, most papers on the finite domain constraint satisfaction problem

[1] `http://www.bids.ac.uk`

define the problem using a different definition and terminology. The fact that none of these definitions are contradictory leads to the concept of *a* finite domain constraint problem.

Other types of constraints, such as geometric constraints or scheduling constraints, however, do not, at first glance, appear to resemble the finite domain definition at all.

Example 4.1 (Definitions of Constraint Problems) From [91], a definition of a finite domain constraint problem:

> The *binary constraint satisfaction problem* (bcsp) involves a set of variables $\{V_1, V_2, \dots\}$ ... a set of binary constraints $\{C_{1,1}, \dots, C_{n,m}\}$ where the constraint $C_{i,j}$ is a relation between V_i and V_j and if $C_{i,j}$ is null then there is no constraint acting from V_i to V_j.

From [102], a definition of a geometric constraint problem:

> *Topological constraint.* A topological constraint defines the topology of a primitive solid itself by specifying the connection between the geometric elements. ...
>
> *Structural constraint.* A structural constraint gives the primitive solid (for example prisms and cylinders) the character of a particular feature. Groove, square hole and step features have the same topological structure of "4-sided prism" ...
>
> *Dimensional constraint.* Dimensional constraints define the size and location of a feature. There are two constraints: the distance and angle between geometric elements. These correspond to dimensions in drawings.

From [124], a definition of a scheduling problem:

> The process of driver scheduling is the construction of a set of legal shifts ... which together cover all the blocks in a particular vehicle schedule ... Blocks may be considered as being divided into units of work which start and finish at relief opportunities pass agree change-over points. ...
>
> Driver scheduling is subject to a set of rules which is specific to an organisation. ... Typically, there are restrictions on the total time worked, on the length of time that may be worked without a meal break, on the total spreadover (duration between beginning and end of a shift), etc.

□

It is in fact possible to define constraints in such a way that all the different types of constraint are merely specialised cases of the general definition. This chapter presents a formal definition of the general constraint problem. The set theoretic definition used is, in fact, not new. Finite domain constraint problems are frequently defined in terms of relations [91, 103] and the definitions used there, carried to their logical extreme are equivalent to the definitions in this chapter.

However, there has been very little research into the unification of the various different types of constraint problem. The definition presented here is capable of describing all types of constraint problem and hence, to an extent, unifying the different definitions currently used.

As well as defining the general constraint problem, this chapter presents the notion of *dimension*, which is used to capture the *size* of a set. Dimensions form a key part of the theories espoused later in this thesis. Although used in many forms in the literature (for example, dimension is equivalent to degrees of freedom as used by Kramer [59] and Fa [26]), no formal definition has been given in the literature. Section 4.5 presents a formal definition of dimensions and discusses the concept in some detail.

This chapter defines the terms *entity*, *constraint*, *constraint problem*, *constraint solver* and *dimension*. In order to demonstrate the effectiveness of the set theoretic approach, examples of both finite and infinite domain problems will be given. However, the chief benefit of this chapter is that the set theoretic approach simplifies the results and proofs presented in the rest of this thesis.

4.1 Entities

A constraint problem is a compromise between objects giving freedom of action and objects taking it away.

Example 4.2 (Components of constraint problems) A finite domain constraint problem consists of variables and constraints. The variables provide freedom as they can take many values and they increase the possible choices available. The constraints restrict freedom as they reduce the number of values a variable is allowed to have.

A geometric constraint problem consists of geometric entities and geometric constraints. The geometric entities provide freedom as they can take many positions

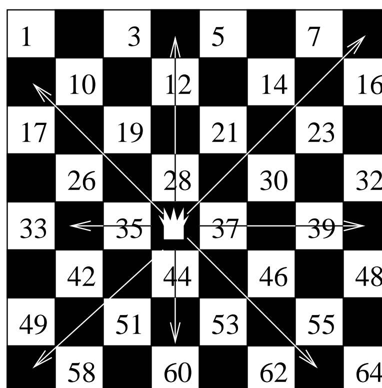


Figure 4.1: Placing a queen on a chessboard

in space. The geometric constraints restrict freedom as they reduce the number of positions a geometric entity is allowed to take. □

Entities are the objects providing freedom. In example 4.2, entities are the variables in the finite domain problem and the geometric entities in the geometric constraint problem.

Definition 4.1 (Entities) *An* **entity** *E is a pair $(label, D)$, where label is a unique identifier for the entity and D is the* **domain** *of E, where D is any set. A* **value** *for an entity is an instance in the domain of the entity. An* **assignation** *for an entity is a formula of the form $E = \{v\}$, determining that entity E is to be assigned the set with the single value v, where v is in domain D. The notation $E = \mathcal{S}$ denotes the fact that E is assigned the set $\mathcal{S}$ of values for E.* □

Where the meaning is unambiguous or the domain unimportant, an entity will sometimes be referred to in this thesis by its label alone.

Throughout this thesis, two examples will be studied in detail. One is a finite domain problem, that of queens being placed on a chessboard. The other is a geometric problem, that of constructing a triangle. These examples will help to demonstrate the ideas espoused in the thesis.

Example 4.3 (Finite domain) A queen can take 64 positions when placed on an empty chessboard. If the squares on the board are numbered 1 to 64, left to right, top to bottom, then the queen can take a value from 1 to 64. Thus the domain of the queen is $\{1, 2, 3, \ldots, 64\}$ (see figure 4.1). Note, however, that there are many ways of describing this domain. For example, by using rows and columns, the queen can take positions $\{(1,1), (1,2), \ldots, (8,8)\}$. In fact, for reasons that are apparent in finite

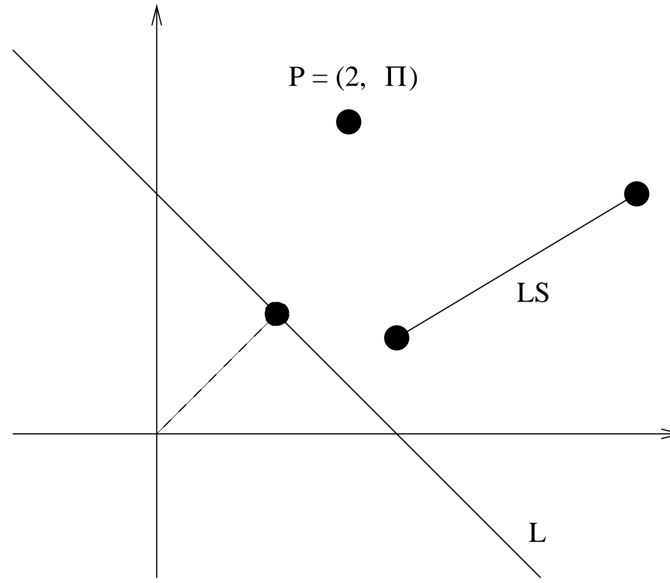


Figure 4.2: A point, a line and a line segment on a plane

domain research, the latter method is a much more efficient means of describing the chessboard.

A value for the queen is any member of the domain, for example, $(1,3)$. An assignation to this value would be of the form, $Queen = \{(1,3)\}$. If the queen can be positioned on any square in a row, then an assignation of the form $Queen = \{(2,1),(2,2),\ldots,(2,8)\}$ is appropriate. □

Example 4.4 (Geometric domain) A point, $(P, \mathbb{R}^2)$, on a plane has domain $\mathbb{R}^2$. A value for a point is any pair of numbers (x, y), such that $x, y \in \mathbb{R}$, for example, $(2, \pi)$. An assignation for P would be $P = \{(2, \pi)\}$ (figure 4.2).

An infinite line, $(L, \mathbb{R}^2)$, in a plane also has domain $\mathbb{R}^2$, as it is uniquely determined by the point on the line closest to the origin, provided that the line does not pass through the origin. A value for L is any pair $(x, y), x, y \in \mathbb{R}$ and an assignation is $L = \{(x, y)\}$.

A line segment, $(LS, \mathbb{R}^4)$, on a plane has domain $\mathbb{R}^4$, as it is uniquely determined by its two end points. As each end point has domain $\mathbb{R}^2$ and can move independently of the other, LS has domain $\mathbb{R}^4$. □

4.2 Constraints

Where entities provide freedom, constraints restrict it. Constraints are described in many different ways but these all reduce to the concept of relations in set theory. In

example 4.2, the constraints are the relations in the finite domain constraint problem and the geometric constraints in the geometric constraint problem.

Definition 4.2 (Relations) *A* **relation***, R, between a pair of entities a, b with domains D_1, D_2 is a subset of the Cartesian product*[†] *of the domains, $D_1 \times D_2$. For $(a, b) \in R$, the notation aRb is used. An n-***ary relation***, S, on a set of variables $a_1, \ldots, a_n$ with domains $D_1, \ldots, D_n$ is a subset of the Cartesian product $D_1 \times \cdots \times D_n$. For $(a_1, \ldots, a_n) \in S$, the notation $S(a_1, \ldots, a_n)$ is used.* □

Since the sets defining a relation are typically infinite, it is not usually possible to explicitly list the members. Thus implicit means, such as set construction notation are normally used. In this thesis, the standard notation for relations will be used, whereby a relation is referred to using the symbol aRb rather than the list of tuples defining the relation.

A useful definition of a relation is in terms of a test function. Each relation $S(a_1, \ldots, a_n)$ has a boolean test function $f : D_1 \times \cdots \times D_n \longrightarrow \{0, 1\}$ associated with it, where $f(x_1, \ldots, x_n) = 1 \Leftrightarrow (x_1, \ldots, x_n) \in S$. This definition is consistent with Fraïssé [33].

Definition 4.3 (Constraints) *A* **constraint** *(C, D_C) on a set of entities E is a restriction of the possible values that the entities in E can simultaneously take. C is a label representing the constraint and D_C is a relation on E. A tuple of values $(v_1, \ldots, v_n)$ for entities $\{(x_1, D_1), \ldots, (x_n, D_n)\}$* **satisfies** *constraint (C, D_C) if*

$$(v_1, \ldots, v_n) \in D_C.$$

□

Associated with every constraint is a *constraint test procedure*. A constraint test procedure (CTP) for a constraint (C, D_C) with respect to the set of entities $\{(x_1, D_1) \ldots, (x_n, D_n)\}$ is the boolean function $f_C : D_1 \times \cdots \times D_n \longrightarrow \{0, 1\}$ associated with C such that $f_C(v_1, \ldots, v_n) = 1 \Leftrightarrow (v_1, \ldots, v_n) \in D_C$.

In fact, it is normally very difficult to define constraints explicitly as relations and they are normally described using only the CTP which also doubles as the label for the constraint. This convention will be adopted in the remainder of this thesis and so, where it is unambiguous, constraints will be denoted only by the function describing them.

Example 4.5 (Finite domain) One of the simplest constraints possible is the equality constraint between two entities. Simply put, the equality constraint means that whatever value one entity takes, the second entity must also take. The CTP for the equality constraint for two queens on a chessboard is:

$$
\begin{aligned}
CTP_= : f(Q1, Q2) &= 1 \text{ if } Q1 = Q2, \\
&= 0 \text{ if } Q1 \neq Q2.
\end{aligned}
$$

Function f has value 1 if and only if both queens are in the same position.

The infamous n-queens problem [103], however, utilises the opposite of the equality constraint, the disequality constraint. The disequality constraint between two queens is satisfied if the two queens are *not* on the same square. The CTP for the disequality constraint is:

$$
\begin{aligned}
CTP_{\neq} : f(Q1, Q2) &= 1 \text{ if } Q1 \neq Q2, \\
&= 0 \text{ if } Q1 = Q2.
\end{aligned}
$$

The n-queens problem involves positioning n queens on an $n \times n$ chessboard, so that no queen attacks another. The constraints involved are usually split into the following types:

1. No queen is on the same square as another;
2. No queen attacks another horizontally;
3. No queen attacks another vertically;
4. No queen attacks another diagonally.

Intelligent definition of the domains of the queens typically removes constraint 1 and one of 2 or 3. However, it is still necessary to describe the remaining two sets of constraints.

For example, when $n = 4$, the 4-queens problem consists of the four entities $\{(Q1, D_4), (Q2, D_4), (Q3, D_4), (Q4, D_4)\}$ with domains $D_4 = \{1, 2, 3, 4\}$ where the value of Qi represents the position of the queen in row i. Note that it is not necessary to say which row each queen is in as they are in separate rows already. $Q1$ is in row 1, $Q2$ is in row 2, etc.

The constraints necessary to prevent each queen being attacked vertically are:

$$C1 : Q1 \neq Q2,$$
$$C2 : Q1 \neq Q3,$$
$$C3 : Q1 \neq Q4,$$
$$C4 : Q2 \neq Q3,$$
$$C5 : Q2 \neq Q4,$$
$$C6 : Q3 \neq Q4.$$

The constraints necessary to prevent each queen attacking along a diagonal are:

$$C7 : Q1 \neq 1 \Rightarrow Q2 \neq Q1 - 1,$$
$$C8 : Q1 \neq 4 \Rightarrow Q2 \neq Q1 + 1,$$
$$C9 : Q1 \neq 3, 4 \Rightarrow Q3 \neq Q1 + 2,$$
$$C10 : Q1 \neq 1, 2 \Rightarrow Q3 \neq Q1 - 2,$$
$$C11 : Q1 = 1 \Rightarrow Q4 \neq 4,$$
$$C12 : Q1 = 4 \Rightarrow Q4 \neq 1,$$
$$C13 : Q2 \neq 1 \Rightarrow Q3 \neq Q2 - 1,$$
$$C14 : Q2 \neq 4 \Rightarrow Q3 \neq Q2 + 1,$$
$$C15 : Q2 \neq 3, 4 \Rightarrow Q4 \neq Q2 + 2,$$
$$C16 : Q2 \neq 1, 2 \Rightarrow Q4 \neq Q2 - 2,$$
$$C17 : Q3 \neq 1 \Rightarrow Q4 \neq Q3 - 1,$$
$$C18 : Q3 \neq 4 \Rightarrow Q4 \neq Q3 + 1.$$

$C7$, for example, can be described as a set as follows:

$$\begin{aligned} C7 \equiv \{&(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),\\ &(3,1),(3,3),(3,4),(4,1),(4,2),(4,4)\}. \end{aligned}$$

Thus the only combinations of values for $(Q1, Q2)$ not allowed by $C7$ are $(2, 1)$, $(3, 2)$ and $(4, 3)$. □

Example 4.6 (Geometric problem) The simplest geometric constraint is also an equality constraint, but is usually referred to as a coincident constraint. For example, two points $(P_1, \mathbb{R}^2)$ and $(P_2, \mathbb{R}^2)$ are coincident if and only if $P_1 = P_2$ (see

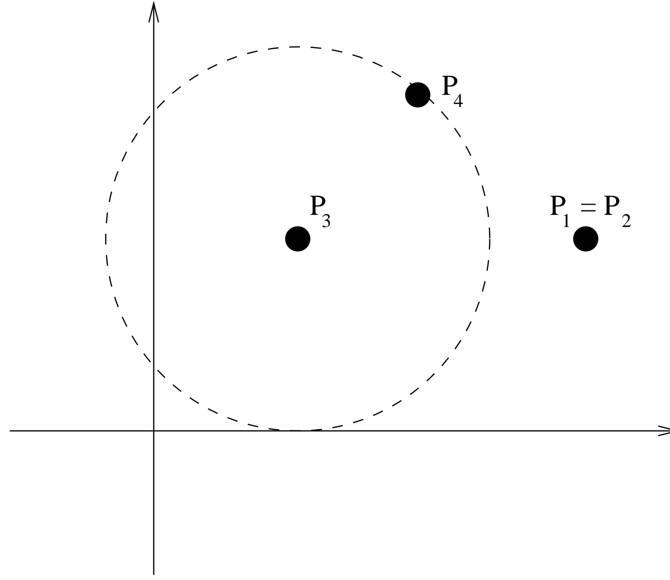


Figure 4.3: An equality constraint and a distance constraint

figure 4.3).

An extension of this is the *distance* constraint. Two points $(P_3, \mathbb{R}^2)$ and $(P_4, \mathbb{R}^2)$ satisfy a distance constraint $d(P_3, P_4) = a$ if and only if $||P_3 - P_4|| = a$ (see figure 4.3). Here $||\cdot||$ is the normal metric in Euclidean space

$$||(x_1, y_1) - (x_2, y_2)|| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Note that the equality constraint is a degenerate case of the distance constraint with $a = 0$.

The equality constraint restricts the freedom of two independent points from $\mathbb{R}^4$ to $\mathbb{R}^2$, as the two points are no longer independent. The distance constraint restricts the freedom of the second point relative to the first to the 1D locus of a circle, radius a, around the first point. Since the first point is free in $\mathbb{R}^2$, it has domain $\mathbb{R}^2$. Since the second point relative to it has domain $\mathbb{R}$, the construction as a whole has domain $\mathbb{R}^3$.

Although the coincident constraint looks similar to the equality constraint of the finite domain problem in example 4.5, the distance constraint does not. However, the distance constraint is a relation as it can be written in the following form:

$$d(P_3, P_4) = a \quad \equiv \quad \left\{((x, y), (z, w)) | \sqrt{(x - z)^2 + (y - w)^2} = a\right\}.$$

Thus both finite domain and geometric constraints can be described as relations. □

4.3 Constraint problems

A constraint problem is the fundamental structure in this thesis. A constraint problem consists of a set of entities and a set of constraints on those entities. A solution to a constraint problem is a set of values for the entities that satisfies the constraints.

Definition 4.4 (Constraint problems) *A* **constraint problem** *is a pair* (Φ, Ψ), *where* Φ *is a set of entities and* Ψ *is a set of constraints.* □

Finite domain constraint problems are sometimes defined as a triple of variables, domains and constraints [114]. This is equivalent to the above definition.

Example 4.7 (Finite domain) The pair

$$F = (\{(Q1, D_4), (Q2, D_4), (Q3, D_4), (Q4, D_4)\}, \{C1, C2, C3, \ldots, C18\})$$

describes the 4-queens constraint problem. □

Example 4.8 (Geometric problem) Let $LS1, LS2, LS3$ be three line segments with end points $LS1_a, LS1_b, LS2_a, LS2_b, LS3_a, LS3_b$ respectively. Then the constraint problem,

$$\begin{aligned} G = \; & (\{(LS1, \mathbb{R}^4), (LS2, \mathbb{R}^4), (LS3, \mathbb{R}^4), (LS1_a, \mathbb{R}^2), (LS1_b, \mathbb{R}^2), (LS2_a, \mathbb{R}^2), \\ & (LS2_b, \mathbb{R}^2), (LS3_a, \mathbb{R}^2), (LS3_b, \mathbb{R}^2)\}, \\ & \{LS1_a = (0,0), LS1_b = LS2_a, LS2_b = LS3_a, LS3_b = LS1_a, \\ & d(LS1_a, LS1_b) = 3, d(LS2_a, LS2_b) = 5, d(LS3_a, LS3_b) = 4, \\ & endpoint(LS1, LS1_a), endpoint(LS1, LS1_b), \\ & endpoint(LS2, LS2_a), endpoint(LS2, LS2_b), \\ & endpoint(LS3, LS3_a), endpoint(LS3, LS3_b)\}), \end{aligned}$$

describes the right angled triangle in a plane shown in figure 4.4. Here the constraint $endpoint(LS, pt)$ is used to indicate that the endpoint of a line segment LS is coincident with a point pt. □

A solution to a constraint problem is a set of values that the entities in the problem take simultaneously such that all of the constraints in the problem are satisfied. This is formalised below.

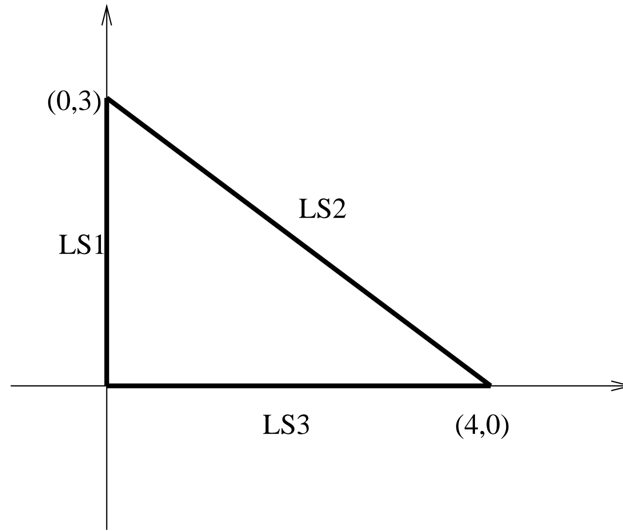


Figure 4.4: A solution to constraint problem G

Definition 4.5 (Solutions) *A* **configuration** *of problem P is a set of assignations for all of the entities in Φ. Configuration $\{x_1 = \{y_1\}, \ldots, x_n = \{y_n\}\}$ is a* **solution** *to a constraint problem $P = (\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}, \Psi)$ if and only if*

$$(y_1, \ldots, y_n) \in \bigcap_{C \in \Psi} C,$$

or, equivalently,

$$\bigwedge_{C \in \Psi} f_C(y_1, \ldots, y_n) = 1,$$

where f_C is the CTP for constraint C. □

Note that the intersection of the constraints requires that the tuples defining the constraints are all in terms of the same entities. In general this is not so, as constraints are described only in terms of the entities significant to them. Thus, the above definition requires *enhanced constraints* which are used to translate constraints defined in terms of the entities significant to them to constraints defined in terms of the whole set of entities in the constraint problem. Enhanced constraints are defined in the following paragraphs.

Definition 4.6 (Notation for Many Configurations) *The notation*

$$\{x_1 = D_1, x_2 = D_2, \ldots, x_n = D_n\}$$

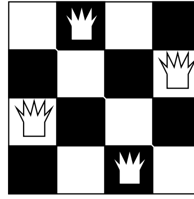


Figure 4.5: A solution to constraint problem F

is used to represent the set

$$\{(x_1 = \{y_1\}, x_2 = \{y_2\}, \dots, x_n = \{y_n\}) | y_1 \in D_1, y_2 \in D_2, \dots, y_n \in D_n\}.$$

□

Example 4.9 (Finite domain) A configuration for F is

$$\{Q1 = \{1\}, Q2 = \{2\}, Q3 = \{3\}, Q4 = \{4\}\}.$$

However, this is not a solution to F as $Q1 = \{1\}, Q2 = \{2\}$ breaks constraint $C8$. A solution for F is the configuration

$$\{Q1 = \{2\}, Q2 = \{4\}, Q3 = \{1\}, Q4 = \{3\}\}$$

shown in figure 4.5, as this satisfies all of the constraints in F.
□

Example 4.10 (Geometric problem) A configuration for G is

$$\begin{aligned}\{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,4,0)\}, LS3 = \{(4,0,0,0)\},\\ LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},\\ LS2_b = \{(4,0)\}, LS3_a = \{(4,0)\}, LS3_b = \{(0,0)\}\}.\end{aligned}$$

This configuration is a solution to G as it satisfies all of the constraints in G, as shown in figure 4.4. In fact, there are an infinite number of other solutions to G, found by rotating triangle A about the origin. Constraint problem G is, in fact, under-constrained, defined in section 4.5.2.

If the constraint $LS2_a = (0,3)$ is added to G to form G', then only two configurations of G' are solutions. These two solutions are the triangle in figure 4.4 and its reflection in the y-axis. G' is said to be well-constrained (see section 4.5.2). □

A constraint C is a subset of the Cartesian Product of the domains of the entities describing C. Each constraint is typically described in a local sense, in that each constraint is usually only described by a subset of the total set of entities E. For example, the constraint $C7$ in problem F of example 4.7 is defined only in terms of entities $Q1$ and $Q2$. Thus, $C7$ is defined as a subset of the Cartesian Product of $Q1$ and $Q2$. However, a constraint problem will have more than one constraint usually, and each constraint will have a different set of entities relevant to it.

According to definition 4.5, the solutions to two constraints are the intersection of the two constraints. However, since the entities relevant to each constraint are different, there is no way of taking the intersection of the two constraints. For example, constraint $C7$ in problem F is defined using $Q1$ and $Q2$, but constraint $C9$ is defined using $Q1$ and $Q3$. The intersection of the two constraints is meaningless.

Consider problem P consisting of the three finite domain entities, $(x, D), (y, D)$ and (z, D) with domains $D = \{0, 1\}$ and two constraints A and B defined by

$$\begin{aligned} xAy &\Leftrightarrow x = y, \\ yBz &\Leftrightarrow y \neq z. \end{aligned}$$

If A and B are enumerated explicitly, then the solutions to P are:

$$\begin{aligned} A &= \{(0,0),(1,1)\}, \\ B &= \{(1,0),(0,1)\}. \end{aligned}$$

However, the intersection of these sets is the empty set, $\emptyset$. Clearly, there are solutions for P: $\{x = 1, y = 1, z = 0\}$ is one, but simply taking the intersection of the two constraints is not enough to identify them. However, if constraint C is defined in a global sense, with respect to E, rather than just the set of entities that affect it, then it is possible to take the intersection of the constraints to find solutions.

Definition 4.7 (Enhanced constraints) *If the set of entities not relevant to C is E_C, then the* **enhanced constraint C with respect to E**, *$C|^E$, is*

$$C|^E = C \times D_{E_C},$$

where D_{E_C} is the Cartesian Product of the domains of E_C. □

For the example above then,

$$\begin{aligned} E_A &= \{z\}, \\ E_B &= \{x\}, \end{aligned}$$

and the enhanced constraints are

$$\begin{aligned} A|^E &= A \times D_{E_A} \\ &= A \times D_z \\ &= \{(0,0),(1,1)\} \times \{0,1\} \\ &= \{(0,0,0),(0,0,1),(1,1,0),(1,1,1)\}, \\ B|^E &= B \times D_{E_B} \\ &= B \times D_x \\ &= \{(1,0),(0,1)\} \times \{0,1\} \\ &= \{(1,0,0),(1,0,1),(0,1,0),(0,1,1)\}. \end{aligned}$$

However, the intersection of $A|^E$ and $B|^E$ is still $\emptyset$. The reason for this is that $A|^E$ and $B|^E$ are arranged in different orders. $A|^E$ is in the order x, y, z, whilst $B|^E$ is in the order y, z, x. In order for the intersection to make sense, they must both have the same ordering of variables. In order to ensure this, we enforce an ordering, $<$ on the Cartesian Products, so that, in the example above, with ordering $<\equiv x < y < z$,

$$\begin{aligned} A|^E_< &= \{\{(0,0),(1,1)\} \times \{0,1\}\} \\ &= \{(0,0,0),(0,0,1),(1,1,0),(1,1,1)\}, \\ B|^E_< &= \{\{0,1\} \times \{(1,0),(0,1)\}\} \\ &= \{(0,1,0),(0,0,1),(1,1,0),(1,0,1)\}. \end{aligned}$$

Clearly the intersection of $A|^E_<$ and $B|^E_<$ is sensible and the result,

$$\{(1,1,0),(0,0,1)\},$$

is the set of all solutions to P and using $<$, the assignations for x, y, z can simply be read off. For simplicity, it is assumed that all constraints in P are enhanced constraints with respect to E. The ordering on Cartesian products will be assumed from now on and omitted for clarity.

If a constraint is presented as an enhanced constraint, then it is important to be able to identify the entities that are significant to that constraint. The entities that are significant to a constraint and whose values affect a constraint are called the *imposed entities*.

Definition 4.8 (Imposed entities) *Assuming that constraints are described in terms of tuples with an ordering* $(v_1, v_2, v_3, \ldots, v_n)$ *corresponding to variables* $\{(x_1, D_1), (x_2, D_2), (x_3, D_3), \ldots, (x_n, D_n)\}$ *respectively, then constraint* C *is* **imposed** *on entity* (x_i, D_i) *if and only if*

$$\begin{gathered} \exists (v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n) \in C \text{ such that} \\ \exists u \in D_i, \text{ such that } (v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n) \notin C. \end{gathered}$$

□

The intuitive explanation of this is that the constraint C is imposed on x_i if and only if varying the value of x_i may violate the constraint.

In fact, it is somewhat clearer to examine the negative statement. A constraint C is *not* imposed on an entity (x_i, D_i) if and only if

$$\begin{gathered} (v_1 \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n) \in C \Rightarrow \\ \forall u \in D_i, (v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n) \in C. \end{gathered}$$

Thus, C is not imposed on x_i if the value of x_i does not really affect C. In this case, we need not include x_i in our description of C as it is essentially superfluous.

In the example above, constraint $A|^E$ is imposed on x as $(1, 1, 1)$ is in $A|^E$, but $(0, 1, 1)$ is not. However, $A|^E$ is not imposed on z as $(0, 0, z)$ and $(1, 1, z)$ are in $A|^E$, whether z is 0 or 1.

Normally constraints are described only in terms of the entities they are imposed on. There is thus a subset of $\{x_1, \ldots, x_n\}$ associated with each constraint consisting of the entities imposed on by the constraint. This subset is denoted by the symbol ξ.

Definition 4.9 (Imposed sets) *The set* $\xi(C)$ *of imposed entities for a constraint* C *is called the* **imposed set of** C. □

In the example above, the imposed set of $A|^E$, $\xi(A|^E)$, is $\{x, y\}$ and the imposed set of $B|^E$ is $\{y, z\}$.

The size of $\xi(C)$ for a constraint C, $|\xi(C)|$, is usually referred to as the *arity* of C (see, for example [103]). A constraint is called unary if $|\xi| = 1$, binary if $|\xi| = 2$, etc. Intuitively, a unary constraint only affects and is affected by one entity, binary constraints affect and are affected by two entities, etc. Unary constraints are simply a restriction on the domain of the imposed entity and as such are usually dealt with by pre-processing.

Since a constraint C is normally only described in terms of the imposed set of C, $\xi(C)$, it is important that $\xi(C)$ is known for each C. Since the ordering for $\xi(C)$ is also important, this too is associated with each constraint.

This thesis studies the use of *subproblems* of constraint problems. A subproblem is a constraint problem that is part of a larger constraint problem.

Definition 4.10 (Subproblems of constraint problems) *Constraint problem* $P_1 = (\Phi_1, \Psi_1)$ *is a* **subproblem** *of constraint problem* $P = (\Phi, \Psi)$ *if*

1. $\Phi_1 \subseteq \Phi$.
2. $\Psi_1 \subseteq \Psi$.
3. *For each* $c \in \Psi_1$, $\xi(c) \subseteq \Phi_1$.

□

Notation will be abused in this thesis so that set operations can be performed on subproblems. In particular:

$$P_1 \cup P_2 = (\Phi_1 \cup \Phi_2, \Psi_1 \cup \Psi_2).$$

4.4 Constraint solvers

A constraint solver is an algorithm or technique that takes as input a constraint problem and produces as output a set of solutions that satisfy that constraint problem. This set can be empty, consist of one, all, some or the best solution, depending on the algorithm used.

Example 4.11 (Finite domain) Constraint solvers for the n-queens problem include forward checking (FC) and backtracking (BT). These algorithms have been described in section 2.3.2. □

Example 4.12 (Geometric problem) Constraint solvers for geometric problems include D-Cubed [86], Erep [37], ICBSM [27] and IGCS [112]. These algorithms have been described in sections 2.3.3.1 and 2.3.3.2. □

4.5 Dimensions

Degrees of freedom analysis [59] uses the degrees of freedom associated with geometric objects to help guide the solution. The degrees of freedom describe the motion of a rigid body to translate, rotate and scale in space. As the solution of a problem progresses, the overall degrees of freedom decrease. When the solver terminates, there may be no degrees of freedom left, in which case the solution space describes a rigid body, or there are degrees of freedom left in which case the terminal solution space is said to be *under-constrained*. An under-constrained solution space contains an infinite number of configurations.

Example 4.13 (Geometric problem) The triangle described in problem G in example 4.8 is underconstrained as it is allowed to rotate about the origin and has one degree of freedom. The triangle described in problem G' has no degrees of freedom left, although it has two possible solutions.

□

Middleditch and Latham [67] use concepts similar to Kramer, which they refer to as over-constrained, well-constrained and under-constrained geometric constraint problems. The issue of whether a constraint problem is over-, under- or well-constrained is referred to as the *constrainedness* of the constraint problem in this thesis. The identification of constrainedness of a problem is very important as it affects the choice of solver used to solve the problem. For example, ICBSM is optimised for solving underconstrained problems, whilst D-Cubed is primarily aimed at solving well-constrained problems and Skyblue at solving over-constrained problems.

This section presents a formal definition of dimensions and constrainedness. Dimensions form the extension of the concept of degrees of freedom to the general constraint problem. With dimensions it is possible to define constrainedness strictly, as shown in section 4.5.2. Since certain constraint solvers are better suited to under-constrained problems and others are better suited to well-constrained problems, it is important to be able to identify the constrainedness of a constraint problem.

4.5.1 Definition of dimensions

Degrees of freedom are defined as the translation, rotation and scaling a geometric entity is allowed to perform in $\mathbb{R}^3$. In a sense, this is equivalent to describing the *size* of the domain of the entity.

For example, consider a cube in $\mathbb{R}^3$. The cube can translate anywhere in $\mathbb{R}^3$, so has a translational domain of $\mathbb{R}^3$ and 3 translational degrees of freedom. The cube can also rotate in $\mathbb{R}^3$ and has rotational domain of $\mathbb{R}^3$ and 3 rotational degrees of freedom. Since it is a cube, all sides must have the same length. However, the cube is free to be as large as desired and so can be scaled by any factor. It therefore has a scaling domain of $\mathbb{R}$ and 1 scaling degree of freedom. Overall the cube has domain $\mathbb{R}^7$ and 7 degrees of freedom.

Unfortunately, simply counting the power of $\mathbb{R}$ in the domain of an entity is not sufficient for the general constraint problem.

Consider, for example, an entity x which takes values in $\mathbb{Z}^2$. The size of the domain of this entity is unspecified. Following the example of the cube above, since $\mathbb{Z} \subseteq \mathbb{R}$ and $\mathbb{Z}^2 \subseteq \mathbb{R}^2$, a size of 2 would not be unreasonable. However $\mathbb{Z}$ is a *countable* set as is $\mathbb{Z}^2$, and so $\mathbb{Z}$ and $\mathbb{Z}^2$ are in fact *equivalent* in a certain sense and should have the same size. In this case, the domain of x would have size 1.

This ambiguity means that, for the general constraint problem, a stricter definition of the size of a domain is needed. This is the *dimension* of a set. In fact, this is difficult to determine. Ideally the dimension function should be a function

$$dim : domain \longrightarrow \mathbb{N}$$

with the following properties

$$dim(\mathbb{R}) = dim(\mathbb{Q}) = dim(\mathbb{Z}) = dim(\mathbb{N}) = 1$$
$$dim(\emptyset) = dim(A) = 0,$$

where A is an arbitrary finite set.

These properties agree with the definitions of degrees of freedom and constrainedness used elsewhere. It is also important for the *dim* function to have the following additional properties:

$$dim(D_1 \times D_2) = dim(D_1) + dim(D_2), \tag{4.1}$$
$$dim(A \cup B) = \max(dim(A), dim(B)), \tag{4.2}$$
$$dim(A \cap B) \leq \min(dim(A), dim(B)), \tag{4.3}$$
$$dim(A \setminus B) \leq dim(A). \tag{4.4}$$

These properties mean that, for example, if a set A is a subset of another set B,

then the dimension of A is no bigger than the dimension of B.

If the domain is a linear space then the definition of dimension should be the same as the definition of dimension for linear spaces, i.e. the size of the smallest linearly independent spanning set.

As was noted above it is very difficult to define *dim* in such a way that property 4.1 is satisfied. This is precisely because $\mathbb{Z}$ and $\mathbb{Z}^2$ are equivalent in a mathematical sense and so $dim(\mathbb{Z}^2) = 1$, not 2 as required by property 4.1.

Manifolds can be used to deal with this problem, providing it is assumed that the sets used in constraint problems are within a metric space. In practice this is not a serious restriction. Manifolds are discussed in brief in appendix A. The reader is referred to [107] for more discussion of manifolds. Dimensions as used in this thesis are therefore defined as follows:

Definition 4.11 (Dimensions) *A set S in a metric space M has dimension n, $dim(S) = n$, if and only if n is the smallest number such that S is an n-manifold in M.* □

This definition gets over the problem discussed above to do with countable sets. However, property 4.1 must be slightly altered to

$$dim(D_1 \times D_2) = dim(D_1) + dim(D_2) \text{ iff}$$
$$D_1 \times D_2 \text{ is not homeomorphic to a proper subset of } \mathbb{R}^{(dim(D_1)+dim(D_2))}$$

Notice that both entities and constraints are characterised by sets. However, the dimension of entities and constraints as commonly used are slightly different.

Definition 4.12 (Dimension of entities) *The dimension of an entity (E, D_E), $dim(E)$ is given by*

$$dim(E) = dim(D_E).$$

□

Definition 4.13 (Dimension of constraints) *The dimension of a constraint (C, D_C), $dim(C)$ is given by*

$$dim(C) = dim(D_{x_1} \times D_{x_n}) - dim(D_C),$$

where $\xi(C) = \{(x_1, D_{x_1}), \ldots, (x_n, D_{x_n})\}$. □

4.5.2 Constrainedness

The *constrainedness* of a problem is related to the number of constraints in the problem. Since constraints are sets also, they too have a dimension. Whilst entities provide extra dimension for a problem, constraints consume dimensions.

Example 4.14 (Geometric problem) Problem G consists of 9 entities and 13 constraints. The three line segment entities each have dimension 4 as the domain of line segments is $\mathbb{R}^4$. The six point entities each have dimension 2 as the domain of the points is $\mathbb{R}^2$. The 13 constraints consume different numbers of dimensions. Fixed point constraints such as $LS1_a = 0$, when described as enhanced constraints, have dimension 10 and consume 2 dimensions from $LS1$, as once the constraint is satisfied it has one end point fixed. By tradition, the dimension of a constraint is described as the number of dimensions it consumes. Equality constraints, such as $LS1_b = LS2_a$ similarly consume 2 dimensions. Distance constraints, such as $d(LS1_a, LS1_b) = 3$ consume only 1 dimension. The endpoint constraints each consume 2 dimensions as they restrict the freedom of a point relative to a line segment so that the point must move with the line segment.

Thus G creates a total of 24 dimensions but consumes only 23. The remaining single degree of freedom accounts for the infinite number of solutions to G.

Problem G' creates 24 dimensions but consumes 25. This does not correspond to the fact that solutions to G' do exist, though there are a finite number of them. □

Given the definition of dimensions in section 4.5.1, it is now possible to define the terms *well-constrained*, *under-constrained* and *over-constrained*. These terms are very important and are used to give an indication of the number of solutions to the constraint problem. The definitions presented here are adapted from the definitions used by Middleditch and Latham [66].

A constraint problem is well-constrained if the dimension consumed by *every* set of constraints is exactly equal to the dimension created by the set of entities in the imposed sets of the constraints. A well-constrained problem would be expected to have a finite number of solutions, though it is possible for a well-constrained problem to have no solutions or an infinite number of solutions. By definition 4.11, finite domain constraints and entities always have a dimension of 0 and so finite domain constraint problems are always well-constrained. Finite domain constraint problems can have no solutions and they are then sometimes referred to as over-constrained. This should not be confused with the definition below.

Definition 4.14 (Well-constrained constraint problem) *Constraint problem* $P = (\Phi, \Psi)$ *is* **well-constrained** *if*

$$\begin{aligned} \forall C &\subseteq \Psi, \\ E &= \{e \in \Phi | e \in \xi(c), c \in C\}, \\ \sum_{e \in E} (dim(e)) &= \sum_{c \in C} (dim(c)) \,. \end{aligned}$$

□

A constraint problem is under-unconstrained if the dimension created by any set of entities is greater than the dimension consumed by the set of constraints imposed on by the entities. An under-constrained problem would be expected to have an infinite number of solutions, though it is possible for an under-constrained problem to have no solutions.

Definition 4.15 (Under-constrained constraint problem) *Constraint problem* $P = (\Phi, \Psi)$ *is* **under-constrained** *if*

$$\begin{aligned} \exists E &\subseteq \Phi, \\ C &= \{c \in \Psi | e \in \xi(c),\ e \in E\}, \\ \sum_{e \in E} (dim(e)) &> \sum_{c \in C} (dim(c)) \,. \end{aligned}$$

□

A constraint problem is over-constrained if the dimension consumed by any set of constraints is greater than the dimension created by the set of entities imposed on the constraints. An over-constrained problem would be expected to have no solutions, though it may have any number.

Definition 4.16 (Over-constrained constraint problem) *Constraint problem* $P = (\Phi, \Psi)$ *is* **over-constrained** *if*

$$\begin{aligned} \exists C &\subseteq \Psi, \\ E &= \{e \in \Phi | e \in \xi(c), c \in C\}, \\ \sum_{c \in C} (dim(c)) &> \sum_{e \in E} (dim(e)) \,. \end{aligned}$$

□

According to these definitions, as noted by Middleditch, it is possible for a constraint problem to be both over-constrained and under-constrained at the same time. This is because a part of the constraint problem can be over-specified and therefore be over-constrained whilst another part of the constraint problem can be under-specified and therefore under-constrained.

Example 4.15 (Constrainedness of problems) In example 4.8, problem G is under-constrained as,

$$\begin{aligned}
E &= \{(LS1, \mathbb{R}^4), (LS2, \mathbb{R}^4), (LS3, \mathbb{R}^4), (LS1_a, \mathbb{R}^2), (LS1_b, \mathbb{R}^2), \\
&\quad (LS2_a, \mathbb{R}^2), (LS2_b, \mathbb{R}^2), (LS3_a, \mathbb{R}^2), (LS3_b, \mathbb{R}^2)\}, \\
C &= \{LS1_a = (0,0), LS1_b = LS2_a, LS2_b = LS3_a, LS3_b = LS1_a, \\
&\quad d(LS1_a, LS1_b) = 3, d(LS2_a, LS2_b) = 5, d(LS3_a, LS3_b) = 4, \\
&\quad endpoint(LS1, LS1_a), endpoint(LS1, LS1_b), \\
&\quad endpoint(LS2, LS2_a), endpoint(LS2, LS2_b), \\
&\quad endpoint(LS3, LS3_a), endpoint(LS3, LS3_b)\}. \\
\sum_{e \in E} (dim(e)) &= dim(LS1) + dim(LS2) + dim(LS3) + dim(LS1_a) + \\
&\quad dim(LS1_b) + dim(LS2_a) + dim(LS2_b) + dim(LS3_a) + \\
&\quad dim(LS3_b) \\
&= 4 + 4 + 4 + 2 + 2 + 2 + 2 + 2 + 2 \\
&= 24. \\
\sum_{c \in C} (dim(c)) &= dim(LS1_a = (0,0)) + dim(LS1_b = LS2_a) + dim(LS2_b = LS3_a) \\
&\quad + dim(LS3_b = LS1_a) + dim(d(LS1_a, LS1_b) = 3) + \\
&\quad dim(d(LS2_a, LS2_b) = 5) + dim(d(LS3_a, LS3_b) = 4) + \\
&\quad dim(endpoint(LS1, LS1_a)) + dim(endpoint(LS1, LS1_b)) + \\
&\quad dim(endpoint(LS2, LS2_a)) + dim(endpoint(LS2, LS2_b)) + \\
&\quad dim(endpoint(LS3, LS3_a)) + dim(endpoint(LS3, LS3_b)) \\
&= 2 + 2 + 2 + 2 + 1 + 1 + 1 + 2 + 2 + 2 + 2 + 2 + 2 \\
&= 23 \\
&< \sum_{e \in E} (dim(e)) .
\end{aligned}$$

Thus problem G should have an infinite number of solutions and this is true as the triangle formed can be rotated about the origin. In fact the difference between

24 and 23 indicates that there should be one degree of freedom in the solution, which is provided by rotation about the origin.

Problem G' is over-constrained as

$$
\begin{aligned}
E &= \{(LS1, \mathbb{R}^4), (LS2, \mathbb{R}^4), (LS3, \mathbb{R}^4), (LS1_a, \mathbb{R}^2), (LS1_b, \mathbb{R}^2), \\
&\quad (LS2_a, \mathbb{R}^2), (LS2_b, \mathbb{R}^2), (LS3_a, \mathbb{R}^2), (LS3_b, \mathbb{R}^2)\}, \\
C &= \{LS1_a = (0,0), LS1_b = LS2_a, LS2_b = LS3_a, LS3_b = LS1_a, \\
&\quad d(LS1_a, LS1_b) = 3, d(LS2_a, LS2_b) = 5, d(LS3_a, LS3_b) = 4, \\
&\quad LS2_a = (0,3), endpoint(LS1, LS1_a), endpoint(LS1, LS1_b), \\
&\quad endpoint(LS2, LS2_a), endpoint(LS2, LS2_b), \\
&\quad endpoint(LS3, LS3_a), endpoint(LS3, LS3_b)\}. \\
\sum_{e \in E} (dim(e)) &= dim(LS1) + dim(LS2) + dim(LS3) + dim(LS1_a) + \\
&\quad dim(LS1_b) + dim(LS2_a) + dim(LS2_b) + dim(LS3_a) + \\
&\quad dim(LS3_b) \\
&= 4 + 4 + 4 + 2 + 2 + 2 + 2 + 2 + 2 \\
&= 24. \\
\sum_{c \in C} (dim(c)) &= dim(LS1_a = (0,0)) + dim(LS1_b = LS2_a) + dim(LS2_b = LS3_a) \\
&\quad + dim(LS3_b = LS1_a) + dim(d(LS1_a, LS1_b) = 3) + \\
&\quad dim(d(LS2_a, LS2_b) = 5) + dim(d(LS3_a, LS3_b) = 4) + \\
&\quad dim(LS2_a = (0,3)) + \\
&\quad dim(endpoint(LS1, LS1_a)) + dim(endpoint(LS1, LS1_b)) + \\
&\quad dim(endpoint(LS2, LS2_a)) + dim(endpoint(LS2, LS2_b)) + \\
&\quad dim(endpoint(LS3, LS3_a)) + dim(endpoint(LS3, LS3_b)) \\
&= 2 + 2 + 2 + 2 + 1 + 1 + 1 + 2 + 2 + 2 + 2 + 2 + 2 + 2 \\
&= 25 \\
&> \sum_{e \in E} (dim(e)).
\end{aligned}
$$

However, problem G' has a finite number of solutions, when over-constrained problems typically have none. This is because some of the constraints in G' are *redundant*, in that they are simply repeating information that can be deduced from other constraints. More specifically, the constraint $d(LS1_a, LS1_b) = 3$ is redundant as it can be deduced from the constraints $LS2_a = (0,3)$, $LS1_a = (0,0)$ and $LS2_a =$

$LS1_b$. In fact, this is the only redundant constraint in the example. If the constraint $d(LS1_a, LS1_b) = 3$ is removed from G', then $\sigma_{c \in C} = 24$, and the problem becomes well-constrained. □

A constraint is redundant if it adds no information beyond what can be deduced from other constraints.

Identification of the constrainedness of a constraint problem is a non-trivial task. Middleditch and Latham present an algorithm for doing this in [66]. This algorithm is fast and incremental and operates on a bipartite graph† representation scheme (see chapter 5).

4.6 Conclusions

This chapter has presented a formal definition of constraint problems. The set theoretic approach adopted means that the definition used is rich enough to describe all kinds of constraint problem, as demonstrated by the finite and infinite-domain examples running through the chapter.

Entities are defined in terms of the values that they are allowed to take. This means that entities are not restricted to objects that have numeric values, but can also include objects such as strings or enumerated values. This general description means that entities can describe many different types of objects.

Constraints restrict the values that entities can take. The definition of constraints as relations allows constraints from many different fields to be described. Although constraints are normally described locally, in terms of a few entities, the definition of *enhanced constraints* allows description of *configurations* and *solutions* of sets of constraints.

Constraint problems consist of a set of entities and a set of constraints. Sets of many configurations form a kind of *search space* that *constraint solvers* use to find solutions to constraint problems.

The *dimension* of a set forms an important guide as to how close a constraint solver is to finding a solution to the constraint problem. If there are an infinite number of configurations under consideration then the constraint solver is probably not near finding a solution to the problem. If there are a finite number of configurations under consideration then the constraint solver is probably near to finding a solution. As constraint solution progresses, the dimension of the search space under consideration will gradually shrink until (hopefully) only a finite number of configurations remain.

Dimensions of constraint problems also form an important method of categorising constraint problems in terms of *constrainedness*. The precise definition of dimensions is complex due to the existence of space filling curves and mappings between countable sets. However, dimensions as defined in this thesis will become increasingly important in later chapters.

As dimensions categorise constraint problems in terms of constrainedness, it is apparent that certain constraint solvers are better at dealing with problems of a certain constrainedness. For example, ICBSM [27] is targeted more at solving under-constrained problems, i.e. those with an infinite number of solutions, whilst DCM [86] is targeted more at solving well-constrained problems, i.e. those with a finite number of solutions. This categorisation will be exploited in chapter 3.

Middleditch and Latham's Connectivity Analysis [67] is an important means of identifying well-, over- and under-constrained subproblems within a constraint problem. Connectivity Analysis identifies *balanced sets* of constraints and entities that are effectively well-constrained. Connectivity Analysis also identifies over- and under-constrained subproblems and the possible means of making them well-constrained. Connectivity Analysis will be used in chapter 3 to identify subproblems of a constraint problem according to constrainedness in order to exploit the categorisation of solvers according to constrainedness given in chapter 2.

Not only does the set-theoretic approach provide a unifying structure for diverse constraint problems, but it also makes it possible to identify the key structures necessary for constraint representation and makes the abstraction of the constraint solution process logical and transparent, as will become clear in the next two chapters.

Chapter 5

Constraint Representation

Constraint problems are meant to be solved. Solution may involve finding one or a number of configurations that satisfy the constraints in the problem. It is important to realise that human beings deal with constraint problems all the time: simply travelling from A to B involves negotiating a path subject to the constraints of available time, transport, routes, traffic and weather. A problem of this nature is very difficult to describe and solve on a computer. Paradoxically, the movements necessary for a Puma robot arm (see figure 5.1, taken from [79]) to reach an object may be difficult for a human but simple to describe and solve on a computer. The significance of this is that not all constraint problems can or should be solved on a computer and that it should never be assumed that any given problem will be best solved by a computer. For the purposes of this thesis, however, it is assumed that there are interesting constraint problems to be solved that are better solved on a computer.

Once a constraint problem is defined using the building blocks of the previous chapter, it is necessary to represent the problem in some way so that a computer can understand the problem and then solve it. Different constraint solvers use different representation schemes and these schemes are not obviously equivalent. In fact, some schemes seem to be only appropriate for specific types of problem. For example, finite domain problem solvers use a representation scheme that appears to be totally unsuitable for infinite domain problems and Lamounier's Equation Graph [62] is very different from Fa's Relationship Graph [112].

As the title suggests, this thesis is primarily interested in solving a *general* constraint problem, one not restricted to particular types of problem. In order to *solve* a general problem, it is first necessary, then, to *represent* the problem on a computer.

To this end, this chapter presents research to find *generic* methods of describing

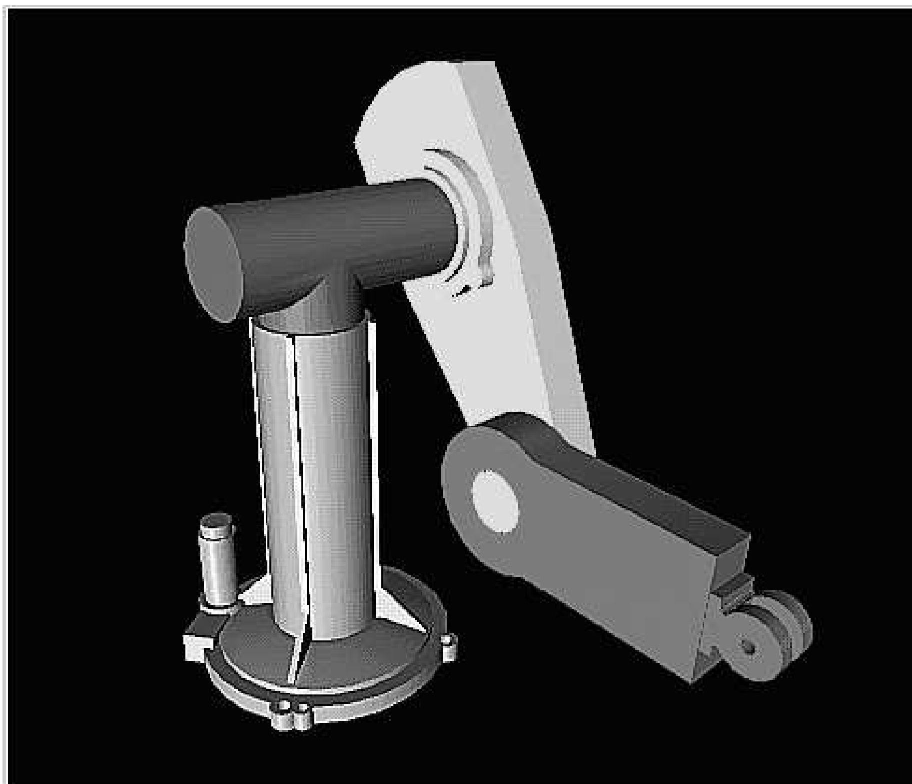


Figure 5.1: A Puma Robot Arm

constraint problems. Section 5.1 discusses the problems of representing the basic constituents of constraint problems, entities and constraints. Section 5.2 presents a formal definition of constraint representation schemes and the properties necessary for a valid constraint representation scheme. Constraint representation schemes will be used to represent constraint problems. Section 5.2 also introduces the notion of a *generic* constraint representation scheme. Section 5.3 discusses CRS schemes currently in use, including an algebraic representation which is generic. Other schemes studied are Relationship Graphs [27, 112], undirected graphs [37, 86, 103], bipartite graphs [62, 66, 100] and hypergraphs [100]. This section also introduces the Constraint/Entity graph, a bipartite graph representation that includes all of the properties necessary for a constraint representation scheme.

Section 5.4 presents *reductions*, a technique developed to compare constraint representation schemes and to prove if a scheme is generic. Reductions have been used to compare the constraint representation schemes in section 5.3 and identify generic constraint representation schemes. Using reductions it is possible to form a hierarchy of constraint representation schemes in terms of expressiveness. Figure 5.2 presents the hierarchy of constraint representation schemes in section 5.3. It is worth

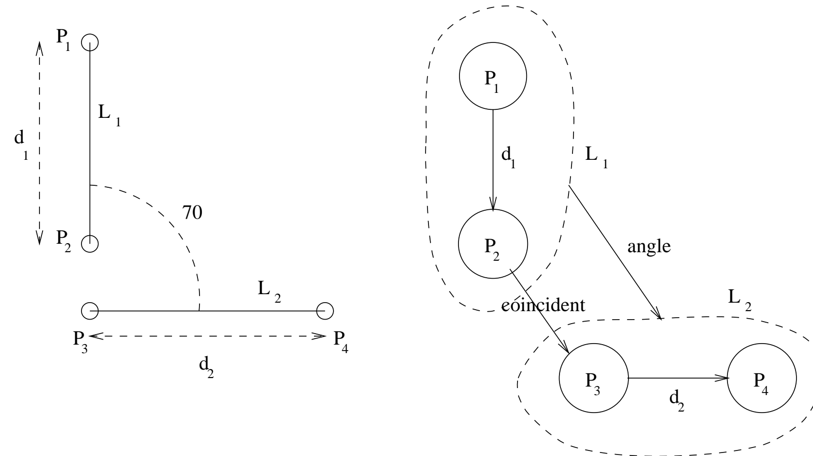


Figure 5.2: A Hierarchy of Constraint Representation Schemes

noting that more than one generic representation scheme exists.

The significance of figure 5.2 is discussed in section 5.5 and conclusions drawn from this chapter.

5.1 Representing entities and constraints

The representation of entities and constraints on a computer is fundamentally the problem of representing sets on a computer, since entities and constraints are both effectively sets. Sets are difficult to represent on a computer because they are usually *infinite* and thus impossible to represent explicitly on a finite machine. Some success has been achieved by representing *finite* entities and constraints and this has led to much research being focused on finite constraint problems.

5.1.1 Finite-domain entities and constraints

Finite entities are usually described by simply listing the domain of the entities explicitly. Thus an entity E that can take values $\{1, 2, 3, 4\}$ will typically be represented using a data structure such as a one dimensional array. If the domain is fragmented or non-sequential, such as the set $\{1, 4, 6, 10, 11, 12\}$, then it will be necessary to store the domain in such a way that it is possible to quickly and efficiently check whether the entity is allowed a certain value in the domain. This will entail the use of a sorted array or hash table or perhaps a linked list. Finite domain algorithms will frequently use a large array which is simply left blank at locations not in the domain.

Finite constraints can always be expressed as binary constraints, as noted in [114]. The key component of a constraint is to know which of the tuples in the Cartesian product satisfy the constraint and which do not. It must therefore be possible to say quickly whether a particular tuple satisfies the constraint or not. In random finite domain problems, this is usually a matter of constructing a two dimensional array so that an entry $(1, 2)$ is 1 if the tuple $(1, 2)$ satisfies the constraint and 0 if the tuple does not satisfy the constraint.

The array data structure is useful because it allows fast lookup of entries, is simple to construct and also allows *random constraints* to be constructed easily and with certain properties [104, 105].

However the array will frequently be *sparse*, in the sense that it will have many more 0s than 1s. For example, only ten tuples satisfy the constraint $x = y$ for the entities x, y with domains $\{1, 2, \ldots, 10\}$. Correspondingly, a 10×10 array is constructed which is 90% full of 0s. Sparse matrix techniques would help this problem.

5.1.2 Infinite-domain entities and constraints

Whilst it is possible to explicitly enumerate the domains of finite entities and constraints, it is impossible to do so for infinite entities and constraints. It is therefore important to identify *implicit* means of describing these structures.

Since sets are defined using a test function (see section 4.2), the simplest method of describing an infinite set is to code a function that implements the test function. It is therefore possible to query whether a particular tuple is in the set and is therefore in the domain of the entity or satisfies the constraint.

More sophisticated techniques involve taking advantage of the *type* of constraint or entity, i.e. by using domain-specific knowledge appropriate to the constraint or entity. For example, it is possible to capture the movements of rigid bodies in space by describing them in terms of *degrees of freedom* [59]. Using degrees of freedom reduces an infinite domain to a finite one and greatly simplifies description of the problem space.

5.2 Representing constraint problems

A *constraint representation scheme* (CRS) is a method of describing a constraint problem, typically using graph† techniques. A CRS should be such that, given a

problem in the CRS, no more information is needed to examine, understand and attempt to solve that problem, other than that provided by the scheme. A problem in a valid CRS will therefore always be *well-posed* in the sense that there is sufficient information to solve the problem given that a solution exists. Thus, objects representing constraints would be linked to the actual constraints themselves in some way and objects representing entities would be linked to the actual entities. Having examined several CRSs in the literature with respect to the very formal framework for the constraint problem defined in the previous chapter, it is apparent that there exists a common set of properties that a CRS should have.

Definition 5.1 (Constraint representation schemes) *A data structure for describing constraint problems is a* **Constraint Representation Scheme**. *It satisfies all of the following properties:*

1. *There should be an identifiable set of entities, together with their domains;*
2. *There should be an identifiable set of constraints, with a constraint test procedure, imposed set and an ordering associated with each constraint;*
3. *There is a 'connection' between a subset of entities if and only if there is a constraint imposed on the subset, i.e. a connection corresponds to the imposed set of a constraint;*
4. *There should be a one-to-one correspondence between the constraints and the connections in the representation;*
5. *There should be a definition of what a solution in the representation looks like.*

□

The definition of 'connection' will vary from scheme to scheme. For example in the constraint graph representation of Owen [86], a connection is just an edge, whilst in the Constraint/Entity graph representation (section 5.3.5), a connection is the set of edges incident to a particular constraint vertex.

A *generic constraint representation scheme* is a representation scheme capable of describing all constraint problems. Generic representation schemes are important as they allow a general purpose constraint management system to store and represent general constraint problems.

Definition 5.2 (Generic representation schemes) *A* **generic** *CRS is a CRS in which* every *constraint problem can be described, where a constraint problem is as defined in section 4.3. The algebraic CRS presented in section 5.3.1 is generic as all constraints can be described in terms of relations. A generic constraint representation scheme will be useful as it can be used to describe all problems that will be encountered.* □

5.3 Example constraint representation schemes

This section provides sample CRSs and introduces the Constraint/Entity graph.

5.3.1 Algebraic representation

The algebraic representation of a constraint problem is the pair $(\mathcal{E}, \mathcal{C})$, where $\mathcal{E} = \{(x_1, D_1), \ldots, (x_n, D_n)\}$ is the set of entities with domains and the set

$$\mathcal{C} = \{(c_1, f_{c_1}, \xi_{c_1}, <_{c_1}), \ldots, (c_m, f_{c_m}, \xi_{c_m}, <_{c_m})\}$$

is the set of constraints, where f_{c_i} is the Constraint Test Procedure for constraint c_i, ξ_{c_i} is the imposed set of constraint c_i and $<_{c_i}$ is an ordering on ξ_{c_i}.

A connection between elements of $\mathcal{F} \subseteq \mathcal{E}$ exists iff $\exists c_i \in \mathcal{C}$ such that $\xi_{c_i} = \mathcal{F}$. The n-tuple $(v_1, \ldots, v_n)$ is a solution of an algebraic representation iff

$$\forall c_i \in \mathcal{C}, \quad \left(f'_{c_i}(v_1, \ldots, v_n) = 1\right) \wedge (\forall j,\ v_j \in D_j),$$

where

$$\begin{aligned} f'_{c_i}(v_1, \ldots, v_n) &= f_{c_i}(v_{i_1}, \ldots, v_{i_m}), \\ \xi_{c_i} &= \{x_{i_1}, \ldots, x_{i_m}\}, \\ \text{and} \qquad & x_{i_1} <_{c_i} x_{i_2} <_{c_i} \cdots <_{c_i} x_{i_m}. \end{aligned}$$

From here, notation will be abused so that f_{c_i} is equivalent to f'_{c_i}.

Given the definition of generic CRSs in section 5.2, the algebraic representation scheme above is generic.

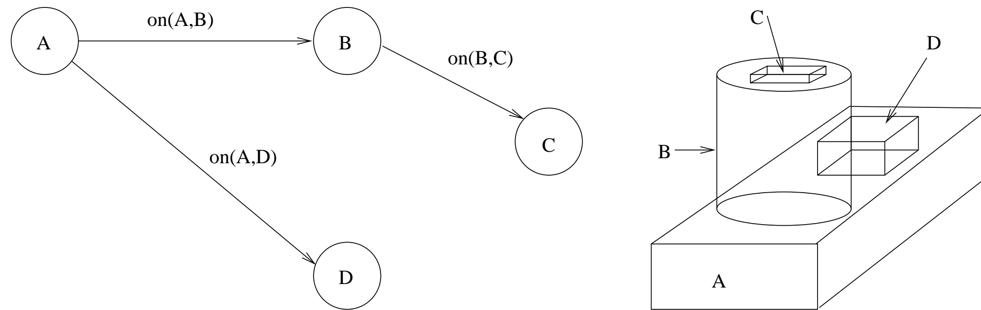


Figure 5.3: An Example of a Relationship Graph with a Solution to the Graph

5.3.2 Relationship graph representation

The Relationship Graph introduced by Fa *et al.* [25] is a directed† graph consisting of vertices V and directed edges E.

The set V is the set of entities and associated with each vertex is the domain of the relevant entity.

The set E is the set of directed edges. Each edge $[u, v]$ represents a constraint between vertex u and vertex v. The CTP for the constraint is the function associated with the geometric constraint represented by the edge. The imposed set of the constraint is $\{u, v\}$ with ordering $u < v$ preserved by the direction of the edge. Thus non-symmetric constraints can be reconstructed from the graph.

A solution of the Relationship Graph is a configuration such that the CTP of each edge is satisfied. Figure 5.3 shows an example of a relationship graph and a solution to the relationship graph.

5.3.3 Undirected graph representation

Undirected† graphs are used in both infinite and finite domain constraint solvers. Infinite domain solvers that use undirected graph structures, such as D-Cubed [86] and Erep [37], currently only handle symmetric constraints† , such as distance or angle constraints. Any change to the representation to handle un-symmetric constraints would necessitate introducing an ordering and would effectively change the undirected graph into a directed graph. It is therefore appropriate to treat the undirected graphs of D-Cubed and Erep as separate constructions from directed graphs, such as Relationship Graphs and investigate them as such.

Finite domain solvers do not appear to distinguish between symmetric and un-symmetric constraints. Since the matrices used to describe constraints can handle unsymmetric constraints, it is therefore assumed that the 'undirected' graphs used

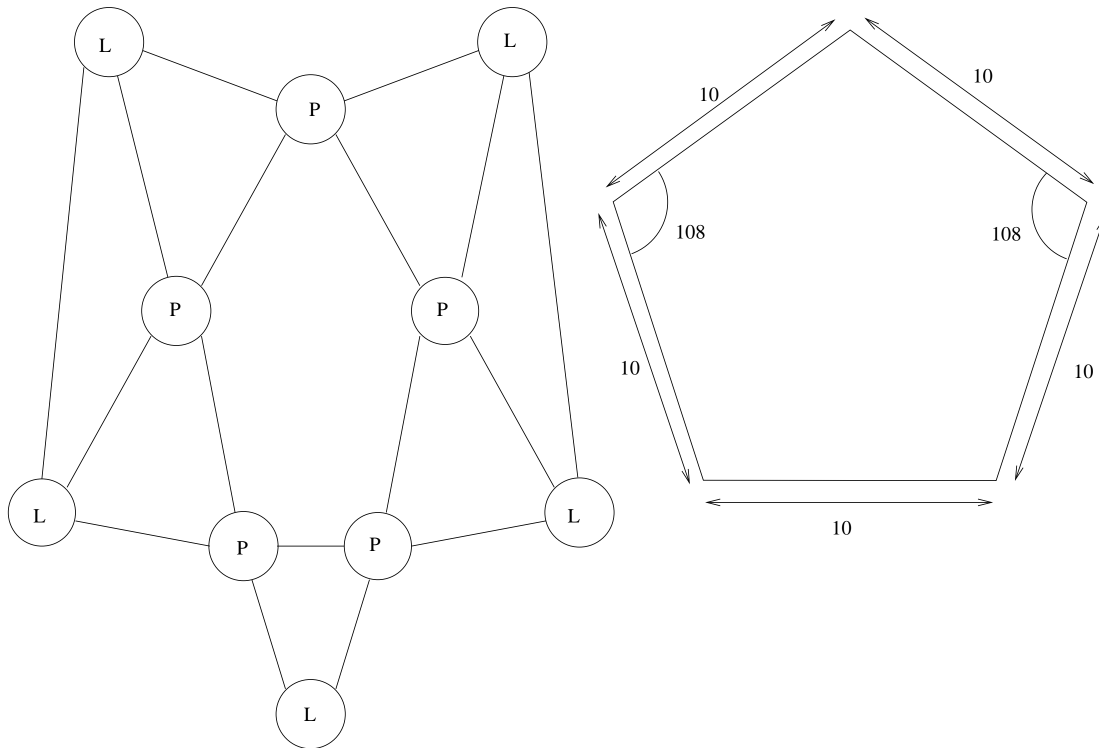


Figure 5.4: An example of an undirected constraint graph with a solution to the graph.

in finite domain problems do, in fact, contain an ordering and are effectively directed graphs.

An undirected graph G consists of a set of vertices V and a set of undirected edges E. A vertex v represents an entity with its domain. An undirected edge (u, v) represents a constraint between vertices u and v with the CTP associated with the constraint. The imposed set of the constraint is $\{u, v\}$ with no ordering, i.e. u and v are interchangeable.

A solution to an undirected graph is a configuration such that the CTP of each edge is satisfied. Figure 5.4 shows a simplified example of an undirected constraint graph and a solution to the graph (from [86]). Note that the actual constraint graph will contain descriptions of the constraints that are represented by the edges in the graph and will also have the domains of the entities.

5.3.4 Hypergraph representation

Serrano [100] uses a hypergraph representation for constraint problems. A hypergraph † CRS is a pair (V, HE), where V is a set of vertices and HE is a set of

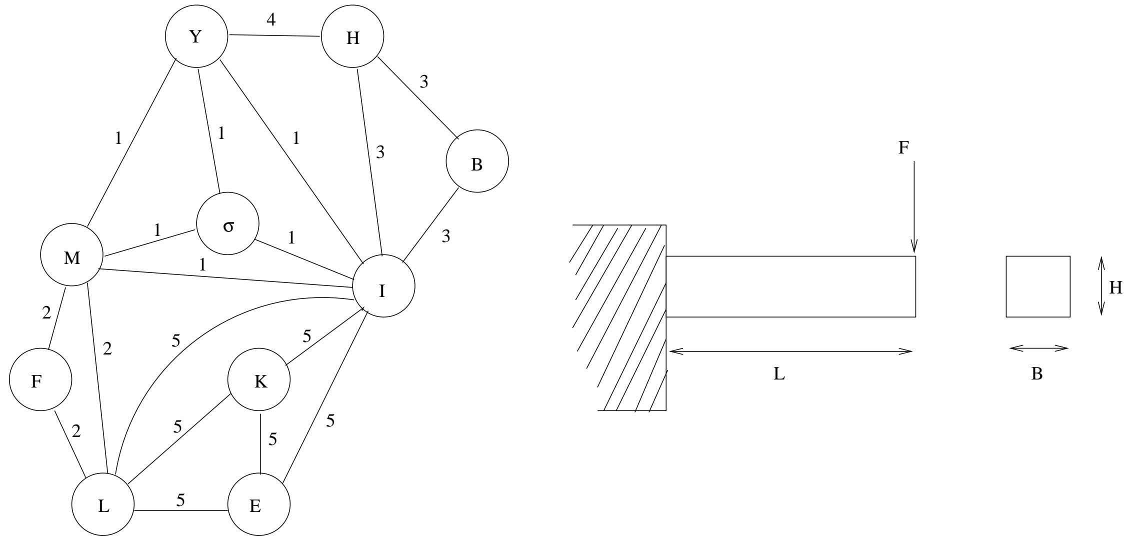


Figure 5.5: An example of a constraint hypergraph with a solution to the graph.

hyperedges† .

A vertex v represents an entity with its domain.

A hyperedge $he = \{v_1, v_2, \ldots, v_n\}$ represents a constraint with a CTP associated with the constraint and imposed set $\{v_1, v_2, \ldots, v_n\}$. Serrano does not mention an ordering on the edges, but it must exist as the constraints he describes are not symmetric. Hence an ordering is assumed to exist on $\{v_1, v_2, \ldots, v_n\}$.

A solution to a hypergraph is a configuration such that the CTP of each hyperedge is satisfied. Figure 5.5 shows a simplified example of a constraint hypergraph and a solution to the graph (from [100]). The hypergraph shown does not show the descriptions of the constraints or the domains of the entities in order to simplify the picture.

5.3.5 Bipartite representation

The *Constraint/Entity graph* (C/E graph) is a labelled,† undirected,† bipartite,† connected graph (C, V, E) with two types of vertices: constraints and entities. Note that the Constraint/Entity representation is adapted from the bipartite representations used by Serrano [100] and Latham and Middleditch [66]. An example Constraint/Entity graph is given in figure 5.6.

The set C is the set of constraint vertices. Each $c = (\mathit{label}, f_c) \in C$ consists of a unique label identifying the constraint and a CTP for checking possible solutions. Constraint vertices are represented as circles in the graph. For example, C in figure 5.6 is a constraint vertex, with CTP $f_C(x, y)$.

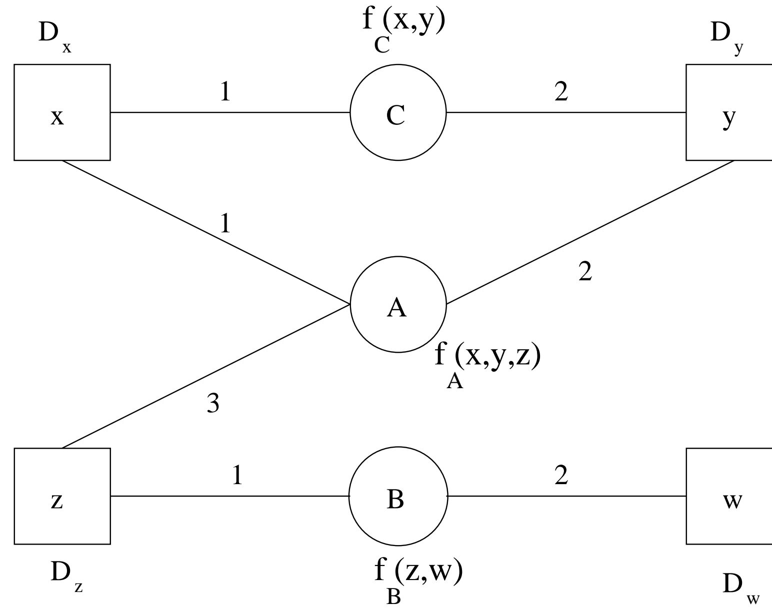


Figure 5.6: Example of a Constraint/Entity Graph

The set V is the set of entity vertices. Each $v = (label, D_v) \in V$ consists of a unique identifying label and the domain of the entity, D_v. Entity vertices are represented as squares in the graph. for example, x in figure 5.6 is an entity vertex, with domain D_x.

The set E is the set of labelled, undirected edges in the graph. In a C/E graph there exists an edge between a constraint node and an entity node iff the associated constraint is imposed on the associated entity, i.e. $(c, v, n) \in E \Leftrightarrow c = (label, f_c)$ and v is a parameter of f_c. Each edge is labelled with a number, n, between 1 and the number of parameters of f_c, denoting the position of the adjacent entity vertex in the list of parameters of f_c. The set of entity nodes adjacent to constraint node c is denoted ξ_c. For example, in figure 5.6, the edge $(A, z, 3)$ is labelled 3 as z is the third parameter in f_A.

A solution of a Constraint/Entity graph is a value $(v_1, \ldots, v_n) \in D_1 \times \cdots \times D_n$ such that $f_c(v_1, \ldots, v_n) = 1 \;\; \forall c \in C$.

5.3.6 Valid representation schemes

Table 5.1 illustrates the various schemes discussed in this section and catalogues the properties necessary for a valid CRS. All of the schemes presented in this section are valid CRSs.

Scheme	Reference	Entities	Constraints	Imposed Set
Algebraic	Section 5.3.1	$\mathcal{E}$	$\mathcal{C}$	Imposed set of C
C/E	Section 5.3.5	V	C	The set of edges incident to constraint vertex
Directed	[27]	V	E	Edge
Undirected	[86]	V	E	Edge
Hypergraph	[101]	V	Set of edges with different label	Set of edges with same label

Table 5.1: Valid Constraint Representation Schemes

5.4 Reductions

The various graph representation schemes have significant advantages over the algebraic CRS. For example the finite domain arc consistency and path consistency solution techniques take advantage of an undirected graph structure and ICBSM's Allowable Motion [27] takes advantage of the directed graph nature of the Relationship Graph to help guide solution. Erep [14] and DCM [86] take advantage of the structure of undirected graphs, Concept Modeler [100] takes advantage of the hypergraph representation and Connectivity Analysis [67] takes advantage of the structure of a bipartite graph representation.

Whilst the algebraic CRS is generic, it is not immediately obvious which of the other CRSs are. It is also useful to be able to compare the expressiveness of CRSs directly so that the relative expressiveness of constraint solvers can be identified.

This section presents a method of comparing CRSs such that not only can more expressive schemes be identified, but also so that there exists a method of translating one scheme to another. To this end the concept of *reducing* one CRS to another is introduced. The concept of reduction is analogous to reductions in complexity theory.

Definition 5.3 (Reductions) *A CRS α can be* **reduced** *to a representation β if*

1. *There exists a mapping ϕ from α to β, $\forall A \in \alpha, \exists B \in \beta$ such that $\phi(A) = B$;*
2. *Every solution of A is also a solution of B and every solution of B is also a solution of A. In other words A and B (the reduced problem) have the same solutions;*
3. *α and β are valid constraint representation schemes;*

4. *The reduction can be done in polynomial time.*

The last criterion is necessary to ensure that the problem does not become intractable due to the reduction. □

Reductions form a tool by which it is possible to compare and contrast constraint representation schemes in terms of their expressiveness.

The intuitive notion of reductions is that any problem in α can be described in β, and that every solution of the reduced problem is a solution of the original problem. Thus the reduced problem is describing the same problem as the original. Since every problem in α can be described in β, β is capable of describing at least as many problems as α and possibly more. This is usually denoted by saying that *β is at least as powerful as α*. It is natural therefore to wonder when two constraint representation schemes are equally powerful.

Definition 5.4 (Equivalent constraint representation schemes) *The two representations α and β are* **equivalent** *if and only if α can be reduced to β and β can be reduced to α. The notation $\alpha \equiv \beta$ is used to denote equivalence.* □

Since generic representation schemes are the goal of this chapter, constraint representation schemes that are equivalent to generic representation schemes will be particularly important.

Theorem 5.1 If $\alpha \equiv \beta$ and β is generic, then α is generic. □

Proof If $\alpha \equiv \beta$, then $\exists \phi$ such that $\forall B \in \beta, \exists A \in \alpha, \quad \phi(B) = A$. Hence B can be described and solved in α, and since B is arbitrary, α is generic. □

Two example reductions are given below. These reductions are important as they prove that there are constraint representation schemes other than the algebraic representation scheme that *are* generic and also that there are constraint representation schemes that are *not* generic. This allows the construction of a *hierarchy* of constraint representation schemes in terms of expressiveness. The two sample reductions prove that

1. The algebraic representation is equivalent to the Constraint/ Entity representation.

2. The Relationship Graph representation is *not* as expressive as the Constraint/ Entity representation.

These reductions allow construction of part of figure 5.2. The remaining reductions necessary to produce figure 5.2 are presented in appendix B.

The reductions presented in theorem 5.2 and theorem 5.3, combined with the reductions presented in appendix B lead to the conclusion that the Constraint/Entity graph presented in section 5.3.5 is *generic*. Since this thesis studies the solution of general engineering design constraint problems, a generic constraint representation scheme is necessary to represent the general problem. The Constraint/Entity graph is also equivalent to the Connectivity graph used by Middleditch and Latham [67] and so can be used for Connectivity Analysis. The identification of the constrainedness of subproblems is an important issue in this thesis and the direct use of Connectivity Analysis is a significant advantage for the Constraint/Entity graph. Consequently, the Constraint/Entity graph will be the constraint representation scheme of choice for this thesis.

Theorem 5.2 The algebraic representation is equivalent to the Constraint/ Entity graph representation. □

Proof The proof is in two parts. First a reduction is formed from the Algebraic Representation to the Constraint/Entity Representation.

A mapping is defined as follows:

For every $(x_i, D_i) \in \mathcal{E}$*, create* (x_i, D_i) *in* V.
For every $(c, f_c, \xi_c, <_c) \in \mathcal{C}$*, create* (c, f_c) *in* C.
Create E *such that* $((c, f_c), (x, D_x), n) \in E \Leftrightarrow x \in \xi_c$ *and* x *is in position* n *in* $<_c$.

By inspection, the resulting graph is a Constraint/Entity graph. As an example, consider the simple constraint problem below. The Constraint/Entity graph resulting from the reduction is as in figure 5.7.

$$\begin{aligned} \gamma \quad = \quad & (\{(x, D_x), (y, D_y), (z, D_z), (w, D_w)\}, \\ & \{(A, f_A, \xi_A, <_A), (B, f_B, \xi_B, <_B), (C, f_C, \xi_C, <_C)\}), \\ \text{such that} \quad & \xi_A = \{x, y, z\}, \xi_B = \{y, z, w\}, \xi_C = \{x, w\}, \\ \text{and} \quad & <_A = [x, y, z], <_B = [y, z, w], <_C = [x, w]. \end{aligned}$$

The entity vertices $\{x, y, z, w\}$, the constraint vertices $\{A, B, C\}$ and the edges

$$\{(A, x, 1), (A, y, 2), (A, z, 3), (B, y, 1), (B, z, 2), (B, w, 3), (C, x, 1), (C, w, 2)\}$$

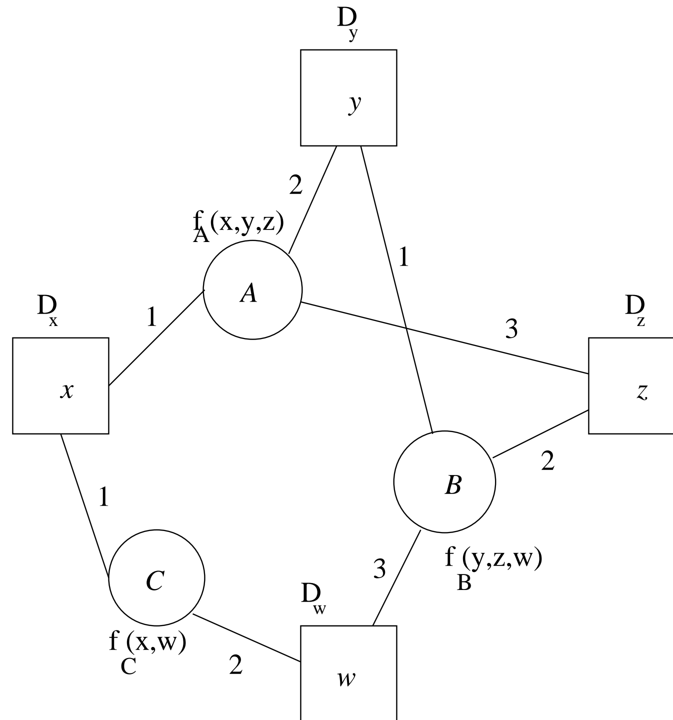


Figure 5.7: Constraint/Entity Representation for Constraint Problem γ

are created. The reduction criteria are demonstrated below:

1. Mapping is defined above.
2. A solution $(y_1, \ldots, y_n)$ of $(\mathcal{E}, \mathcal{C})$ is a solution of the Constraint/ Entity graph by the definition of a solution of the Constraint/ Entity graph and that of the algebraic representation, and vice versa.
3. Both are valid CRSs, by table 5.1.
4. For every constraint in the constraint problem, a constraint vertex is created, taking O(m), where m is the number of constraints. For every entity in the constraint problem, an entity vertex is created, taking O(n), where n is the number of entities. For every constraint in the constraint problem an edge for each entity the constraint is imposed on is created, taking O(mn). Hence the reduction is polynomial.

Secondly the reverse reduction, from a Constraint/Entity graph to the algebraic representation, is proved, using the following mapping:

Given Constraint/Entity graph (C, V, E), *construct* $\mathcal{E} = V$. *Construct* ξ_c *for each* $c \in C$ *by*

$$\xi_c = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\},$$

such that $((c, f_c), (x_{i_j}, D_{x_{i_j}}), n) \in E$. *Then construct* $<_c$ *for each* $c \in C$ *by*

$$x <_c y \Leftrightarrow (c, x, n) \in E \text{ and } (c, y, m) \in E \text{ and } n < m.$$

Then define

$$\mathcal{C} = \{(c, f_c, \xi_c, <_c) | (c, f_c) \in C\}.$$

Then the pair

$$(\mathcal{E}, \mathcal{C})$$

is the algebraic representation as defined in section 5.3.1.

The reduction criteria are now checked to make sure the reduction is valid.

1. Map is defined above.
2. By the definition of the solutions of a Constraint/Entity graph and of the algebraic representation, the solutions are the same.
3. Both are valid CRSs, from table 5.1.
4. For each entity vertex in the C/E graph, an entity in $\mathcal{E}$ is created, taking $O(n)$, where n is the number of entities. For each constraint in the C/E graph, a constraint in $\mathcal{C}$ is created. There may be $O(n)$ edges incident to each constraint and $O(m)$ constraints, so reconstructing the imposed sets is $O(mn)$. Hence the reduction is polynomial.

Therefore the algebraic representation for constraint problems is *equivalent* to the Constraint/Entity graph representation. This is as expected and means that every problem that can be described in terms of relations can be described as a Constraint/Entity graph. So Constraint/Entity graphs are generic. □

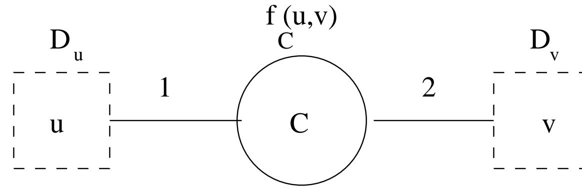


Figure 5.8: New Construct for Constraint Edges

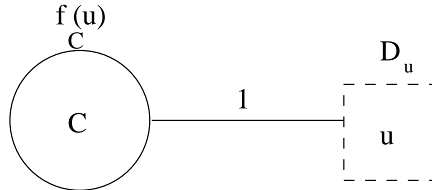


Figure 5.9: New Construct for Constraint Loops

Theorem 5.3 The Relationship Graph [27] is not equivalent to the Constraint/ Entity graph and is strictly less powerful than it. □

Proof The proof is in two parts. First a reduction is formed from the Relationship Graph to the Constraint/Entity representation, demonstrating that the Constraint/Entity Graph is at least as powerful as the Relationship Graph. The Relationship Graph is a directed graph, in which constraints are represented by directed edges and entities are represented by circular vertices.

The mapping used to reduce a directed constraint graph (V, E) to a C/E graph (V_B, C_B, E_B) is as follows:

Create $V_B = V$.

For each edge e in E, with CTP f_e*, create constraint vertex* (e, f_e) *in* C_B.

For each edge $e = [u, v]$ *in* E*, create edges* $(e, u, 1)$ *and* $(e, v, 2)$ *in* E_B*. Effectively, this means replacing all edges* $((u, D_u), (v, D_v))$ *with the construct in figure 5.8, where* $((u, D_u), (v, D_v))$ *represents constraint* C*, with* $\xi_C = \{u, v\}$ *and* $<_C = [u, v]$*,* C *has CTP* f*. As edges in the Relationship Graph are directed, there is an implicit ordering which is captured by having the edge from* C *to* u *labelled with a 1 and* (C, v) *labelled with a 2.*

For each edge $e = [u, u]$ *in* E*, create edge* $(e, u, 1)$ *in* E_B*. Effectively this means replacing all edges* $((u, D_u), (u, D_u))$ *with the construct in figure 5.9.*

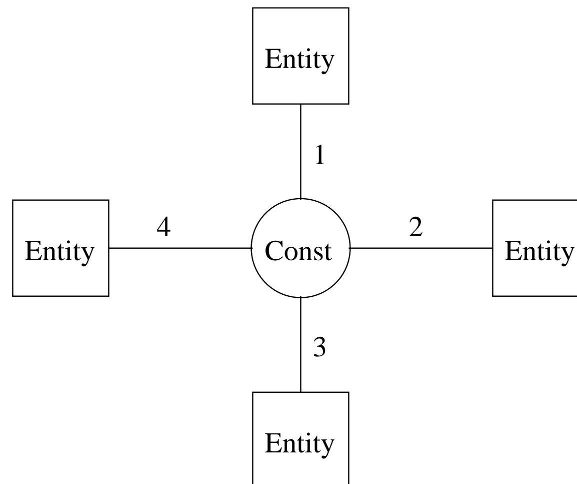


Figure 5.10: Representing Quaternary Constraints in a Constraint/Entity Graph

The resultant graph is clearly a C/E graph. Checking the reduction criterion:

1. The mapping is defined above.

2. A solution to the constraint graph will result in values being assigned to the various entity vertices. The assignment of the same values to the entity vertices in the reduced C/E graph will form a solution to the C/E graph problem, since the same CTPs are used in both schemes.

3. Both are valid CRSs, as in table 5.1.

4. The reduction can be done in linear time in the number of edges in the constraint graph. It is therefore polynomial time.

So the relationship graph can be reduced to a C/E graph.

Secondly, it is necessary to prove that Constraint/Entity graphs cannot be reduced to Relationship Graphs. This implies that Constraint/Entity graphs are strictly more powerful than Relationship Graphs. The weakness of RGs lies in the fact that only binary and unary constraints can be described.

It is however possible to represent ternary or n-ary constraints in a Constraint/ Entity graph by the number of edges $e \in E$ such that the constraint vertex is incident to e (see figure 5.10).

Since more complex, non-geometric constraints will probably require n-ary constraints, such a representation is clearly desirable.

Since n-ary constraints, $n \geq 3$ cannot be described using a Relationship Graph, it follows that the Constraint/Entity representation is more powerful or more general than Relationship Graphs. □

5.5 Conclusions

This chapter has discussed the problems of representing constraint problems on a computer. The problems of representing constraints and entities correspond to the problems of representing large or infinite sets of values. Sparse matrix techniques can be used in the case of finite entities and constraints, provided that there are few non-zero values. In general, implicit set notation is used for infinite domain entities and constraints. Equations and inequalities can be described using explicit symbolic mathematical descriptions, for example using Maple [18]. It is also possible to use domain specific knowledge to simplify description of constraints and entities. For example, the movements of rigid bodies in space can be described using degrees of freedom [27, 58], which allows description of an infinite domain to be simplified to a finite one.

Constraint problems are represented using *constraint representation schemes*. Constraint representation schemes are a means of abstracting out relevant information about a constraint problem, for example the imposed set of constraints, in such a way that they can be exploited by a constraint solver. Most constraint representation schemes take advantage of graph representations in order to use the large body of graph algorithms that already exist.

For example the finite domain arc consistency and path consistency solution techniques take advantage of an undirected graph structure and ICBSM's Allowable Motion [27] takes advantage of the directed graph nature of the Relationship Graph to help guide solution. Erep [14] and DCM [86] take advantage of the structure of undirected graphs, Concept Modeler [100] takes advantage of the hypergraph representation and Connectivity Analysis [67] takes advantage of the structure of a bipartite graph representation.

Some constraint representation schemes are *generic*. A generic constraint representation scheme is capable of describing all constraint problems. The algebraic representation scheme described in section 5.3.1 is generic, but there is no direct means of identifying which of the other representation schemes presented are. Consequently, the notion of *reductions* was introduced. Reductions can be used to convert between constraint representation schemes and also to compare two con-

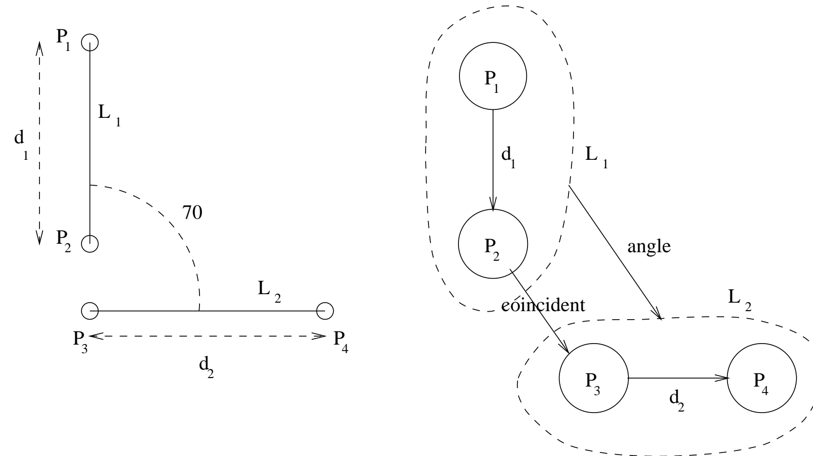


Figure 5.11: A Hierarchy of Constraint Representation Schemes

straint representation schemes. Two constraint representation schemes α and β are *equivalent* if α can be reduced to β and vice versa.

Theorem 5.1 proves that a constraint representation scheme that is equivalent to a generic representation scheme is generic. This theorem performs an important role in identifying the constraint representation scheme that will be used to represent the general engineering design constraint problems of interest to this thesis.

Using reductions, it is possible to form a hierarchy of constraint representation schemes in terms of expressiveness. The hierarchy described in figure 5.2 is reproduced here for convenience.

This hierarchy indicates that the Constraint/Entity graph presented in section 5.3.5 is *generic*. The Constraint/Entity graph is also equivalent to the Connectivity graph used by Middleditch and Latham [67] and so can be used for Connectivity Analysis. The identification of the constrainedness of subproblems is an important issue in this thesis and the direct use of Connectivity Analysis is a significant advantage for the Constraint/Entity graph. Consequently, the Constraint/Entity graph will be the constraint representation scheme of choice for this thesis.

Chapter 6

Constraint Satisfaction

Once a constraint problem is defined and represented on a computer, it only remains to find the required number of solutions to the problem, a process normally referred to as *constraint satisfaction*. Currently, there exist many different constraint satisfaction techniques, ranging from trying every possible configuration to sophisticated heuristic and reasoning methods.

The purpose of this thesis is to investigate the possibility of defining, representing and solving a general constraint problem efficiently on a computer. No single algorithm currently available can handle *all* constraint problems. Even versatile numerical algorithms, such as Newton-Raphson [90], cannot solve all constraint problems as not all constraint problems can be described using equations (see Kramer [59], p23). As noted in [90],

> There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods.

Consequently, because the general constraint problem may consist of a number of nonlinear equations and inequalities, it is unlikely that any single algorithm will be able to solve the general constraint problem. However, there exist many algorithms capable of solving specific types of constraint problem. It is logical therefore to wonder whether the specialised constraint solvers could be combined in some fashion and what kind of problem the combined solver could handle.

This gives rise to the concept of *hybrid* constraint solvers, which is discussed in more depth in chapter 7. However, in order to study hybrid solvers, it is necessary to investigate the constraint satisfaction process, to identify common properties of satisfaction algorithms that will allow hybrids to be constructed.

Consequently, this chapter presents a mathematical framework of the constraint satisfaction process. This framework has several benefits beyond the identification of common properties:

1. The framework helps to clarify current constraint satisfaction algorithms.
2. The framework allows formal definition of properties such as consistency, soundness and completeness, and provides a powerful method of proving their existence in a given algorithm.
3. The framework is sufficiently rich to allow the integration of previously unconnected topics such as constraint priorities, backtracking, variable-driven algorithms and incremental techniques.

Section 6.1 presents a definition of constraint satisfaction solvers and discusses *solution spaces* in this context. Solution spaces are the set of configurations to be searched for solutions at any given time. An extensive study of constraint satisfaction algorithms has led to the concept of *solution steps* and these are introduced in section 6.2. Solution steps may have the properties of *consistency*, *soundness* or *completeness* associated with them and these properties are defined also. A satisfaction scheme, however, is not typically a single solution step, but is instead a *sequence* of solution steps. Consequently, the notion of a *solution process* is defined. These also may be consistent, sound or complete and the link between individual steps having a property and a process having a property is a powerful tool for categorising a constraint satisfaction scheme.

Section 6.3.1 demonstrates the power of the framework as it allows the description of techniques used to improve the power of constraint satisfaction, such as constraint priorities, backtracking, variable-driven and incremental techniques.

Section 6.4 presents conclusions from this chapter.

6.1 Constraint solution

As defined in section 4.4, a constraint solver is an algorithm or technique that takes as input a constraint problem and produces as output a set of solutions that satisfy that constraint problem. This set can be empty, consist of one, all, some or the best solution, depending on the algorithm used.

Constraint solvers typically work by searching through a large, possibly infinite, space of possible configurations trying to find specific configurations that satisfy the

constraints. The set of configurations being explored is usually called the *solution space* of the problem.

6.1.1 Solution spaces

The purpose of a constraint solver is to identify a number of solutions from within a much larger set of configurations, most of which are not solutions. As the constraint solver progresses, it gradually narrows down the set of configurations by eliminating configurations.

The set of configurations at any given point in the solution process is the *solution space*. This is a generalisation of nodes in *search trees* in finite domain problems [103] and *C-spaces* [72, 121] in spatial planning problems.

Search trees are (semi-)graphical representations of a search algorithm as it examines the possible configurations of a finite domain problem. The branches of the tree correspond to differing choices of values for variables at that point. Nodes correspond to partial solutions to the constraint problem and are equivalent to a set of configurations. Since infinite domain problems will typically not use explicit search techniques, search trees are not appropriate.

A configuration space, or C-space, is the multi-dimensional space of possible configurations of an object [72]. C-spaces are of particular interest in spatial planning problems. If an object in a domain is to avoid a set of obstacles, then the problem of finding a position for the object reduces to the problem of identifying a point not within any of the projections of the object with respect to the obstacles into C-space.

Solution spaces will be used to describe the current state of a solution process.

Definition 6.1 (Solution space) *A* **solution space** *is a set of configurations. Solution spaces will be denoted by the symbol* $\boldsymbol{\mathcal{D}}$. □

Once a constraint problem is defined, there is an initial solution space of all possible configurations.

Example 6.1 (Solution space) Consider constraint problem G of example 4.8. The set of all possible configurations of $LS1, LS2, LS3, LS1_a, LS1_b, LS2_a, LS2_b, LS3_a, LS3_b$ is

$$\begin{aligned}\boldsymbol{\mathcal{D}}(0) = \{&LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \mathbb{R}^2, LS1_b = \mathbb{R}^2,\\ &LS2_a = \mathbb{R}^2, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, LS3_b = \mathbb{R}^2\}.\end{aligned}$$

This is the initial solution space of G. $\boldsymbol{\mathcal{D}}(0)$ represents the three line segments being allowed to take any position in the plane. Note that, since this is the initial solution space, the notation $\boldsymbol{\mathcal{D}}(0)$ is used, to indicate the solution space at time 0. The reason for this notation will become clear in section 6.2.3. □

As a constraint problem is solved, and configurations are eliminated, the solution space shrinks. Eventually, the constraint solver terminates and outputs a solution space. The contents of this solution space depend on the constraint solver. It may consist of all, one or no solutions. It may contain configurations that are not solutions. It may find no solutions though some exist or it may find 'solutions' where none exist.

Example 6.2 (Geometric terminal solution space) Using a constraint solver such as D-Cubed on problem G' would result in a terminal solution space of the form

$$\begin{aligned}\boldsymbol{\mathcal{D}}(\kappa) \quad = \quad & \{\{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,4,0)\}, LS3 = \{(4,0,0,0)\}, \\ & \{LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\}, \\ & LS2_b = \{(4,0)\}, LS3_a = \{(4,0)\}, LS3_b = \{(0,0)\}\}, \\ & \{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,-4,0)\}, LS3 = \{(-4,0,0,0)\}, \\ & LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\}, \\ & LS2_b = \{(-4,0)\}, LS3_a = \{(-4,0)\}, LS3_b = \{(0,0)\}\}\}.\end{aligned}$$

□

The notation $\boldsymbol{\mathcal{D}}(\kappa)$ will be used to denote a terminal solution space in the remainder of this thesis.

6.2 A framework for the solution process

As discussed in section 6.1, constraint solvers convert an initial solution space to a terminal solution space, hopefully consisting of solutions to the constraint problem. Although all constraint solvers work in different ways, it is possible to examine the inner workings of a variety of constraint solvers and extrapolate common features to give a framework for describing constraint satisfaction.

This section presents the results of such a study. Having examined the algorithms for all of the constraint solvers in chapter 2, it seemed that all of the constraint

solvers studied worked in an incremental fashion, gradually refining their knowledge of the constraint problem until it was possible to identify solutions. This principle translates well into the language of solution spaces. The framework that was developed to describe the constraint satisfaction process allows concrete statements to be made not only about individual constraint solvers, but also about the combination of constraint solvers to handle the general constraint problem.

Note that this is very similar to the description of incremental solvers such as INCES [62] and IGCS [112]. Incremental solvers try to reuse as much previous knowledge of the solution of a constraint problem as possible when a new constraint is added. Incremental solvers are discussed as a refinement of the satisfaction framework in section 6.3.4.

This section introduces the *solution step*, the basic structure in any solution process. *Solution processes* consist of a series of solution steps combined to gradually refine the solution space of a problem. The definition of solution steps allows a natural description of the properties of consistency, soundness and completeness, important properties of constraint solvers.

As a solution process is a series of solution steps, it is natural to apply the properties of consistency, soundness and completeness to solution processes. The link between properties of solution steps and solution processes forms an important tool in making statements about constraint solvers.

6.2.1 Solution steps

A *solution step* is the building block of constraint solvers. A solution step takes a solution space and uses a set of constraints to refine the solution space by eliminating some configurations.

Definition 6.2 (Solution step) *A* **solution step** *is a mapping* $\xrightarrow{\Psi_k}$,

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$$

where Ψ_k *is a set of constraints* $C_1, \ldots, C_m$ *and* $\boldsymbol{\mathcal{D}}(i)$ *is the solution space of the set of entities* Φ *at step* i. *For each step,*

$$\boldsymbol{\mathcal{D}}(k) \subseteq \boldsymbol{\mathcal{D}}(k-1). \tag{6.1}$$

□

Example 6.3 (Solution step) Consider the first solution step taken by Kramer's degrees of freedom analysis constraint solver [59] to solve problem G' of example 4.10. The initial solution space is:

$$\mathcal{D}(0) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \mathbb{R}^2, LS1_b = \mathbb{R}^2, \\ LS2_a = \mathbb{R}^2, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, LS3_b = \mathbb{R}^2\}.$$

as all three line segments float freely in space. Let the first solution step be

$$\mathcal{D}(0) \xrightarrow{LS1_a=0} \mathcal{D}(1),$$

where

$$\mathcal{D}(1) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, LS1_b = \mathbb{R}^2, \\ LS2_a = \mathbb{R}^2, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, LS3_b = \mathbb{R}^2\}.$$

This solution step can be read thus:

> From an initial solution space of $\mathcal{D}(0)$, the constraint that one point is fixed at the origin is processed and this results in a solution space of $\mathcal{D}(1)$. $\mathcal{D}(0)$ can be interpreted that all three line segments and all six points are free to move in the plane, whereas $\mathcal{D}(1)$ can be interpreted so that one of the points is fixed at the origin.

This is a solution step as $\{(0,0)\} \subseteq \mathbb{R}^2$ and so $\mathcal{D}(1) \subseteq \mathcal{D}(0)$. □

6.2.2 Properties of solution steps

Constraint solvers are frequently described in terms of properties that they have. These properties usually describe the format of the terminal solution space produced by the constraint solver and it is very useful to be able to say that a solver will produce a terminal solution space with a particular property no matter what problem is solved. Another desirable property is that a constraint solver terminate at all. The most common properties desired of constraint solvers are

1. Consistency. The terminal solution space contains a solution to the problem given if any solution exists.

2. Soundness. The terminal solution space consists *only* of solutions to the problem given.

3. Completeness. The terminal solution space contains *all* solutions to the problem given.

For example, numerical solution is not consistent, as it may fail to converge to a solution. Many finite domain techniques, such as forward checking, backtracking or backmarking [114], search through the entire solution space and are consistent, sound and complete. However, it is not always obvious whether a particular constraint solver has any or all of the above properties.

It is very desirable to be able to characterise constraint solvers with these properties both so that concrete statements can be made about the terminal solution space found by the constraint solver and also to help categorise combined solvers.

Consequently, it is important that the framework for constraint satisfaction be able to capture the three properties described above. This section presents the formal definitions of consistency, soundness and completeness of solution steps.

A solution step is *consistent* if it always retains at least one solution to the problem, if any solution exists. Solvers such as forward checking or backtracking are consistent as they search exhaustively for a solution, but hillclimbing [114] is not. Hillclimbing is a constraint solution technique that tries to find solutions using optimisation techniques. Hillclimbing examines the constraint problem and then (typically) uses the derivatives of the nonlinear equations to identify the most promising direction for a solution. However, hillclimbing can get stuck on local minima and will then not proceed to a solution as it does not think one exists. Consequently, hillclimbing is not consistent.

Definition 6.3 (Consistent solution step) *A solution step* $\xrightarrow{\Psi_k}$ *is* **locally consistent** *if* $\Psi_k = \{C_{\Psi_{k_1}}, C_{\Psi_{k_2}}, \ldots, C_{\Psi_{km}}\}$, *and*

$$C_{\Psi_{k_1}} \cap C_{\Psi_{k_2}} \cap \cdots \cap C_{\Psi_{km}} \neq \emptyset \Rightarrow C_{\Psi_{k_1}} \cap C_{\Psi_{k_2}} \cap \cdots \cap C_{\Psi_{km}} \cap \boldsymbol{\mathcal{D}}(k) \neq \emptyset.$$

That is there is always at least one solution that satisfies the local set of constraints.

A solution step $\xrightarrow{\Psi_k}$ *is* **(globally) consistent** *if* $\Psi = \{C_1, C_2, \ldots, C_r\}$, *and*

$$C_1 \cap C_2 \cap \cdots \cap C_r \neq \emptyset \Rightarrow C_1 \cap C_2 \cap \cdots \cap C_r \cap \boldsymbol{\mathcal{D}}(k) \neq \emptyset.$$

That is there is always at least one solution that satisfies the global set of constraints.
□

In the rest of this thesis, the shorthand notation

$$\boldsymbol{\mathcal{C}}(\Psi') = \bigcap_{C \in \Psi'} C = C_{\Psi'_1} \cap C_{\Psi'_2} \cap \cdots \cap C_{\Psi'_k}$$

represents the intersection of all the constraints in a constraint subset, Ψ'. This intersection represents all the values that satisfy all the constraints in the subset.

A solution space $\boldsymbol{\mathcal{D}}(k)$ contains many possible solutions of the constraint problem. Depending on the problem, many of these possible solutions may be valid or very few may be or none at all. Given that $\boldsymbol{\mathcal{D}}(k)$ may be infinite, it would not be desirable to search through $\boldsymbol{\mathcal{D}}(k)$ to find all of the valid solutions. It is therefore desirable that the final solution space $\boldsymbol{\mathcal{D}}(\kappa)$ contain *only* valid solutions. This gives rise to the notion of *soundness*.

A solution step $\xrightarrow{\Psi_k}$ is *locally sound* if $\boldsymbol{\mathcal{D}}(k)$ contains only valid solutions of Ψ_k in $\boldsymbol{\mathcal{D}}(k-1)$. A solution step is *sound* if $\boldsymbol{\mathcal{D}}(k)$ contains only valid solutions of the constraint problem.

Definition 6.4 (Sound solution step) *A solution step* $\xrightarrow{\Psi_k}$ *is* **locally sound** *if* $\Psi_k = \{C_{\Psi_{k_1}}, C_{\Psi_{k_2}}, \ldots, C_{\Psi_{k_m}}\}$*, and*

$$\boldsymbol{\mathcal{D}}(k) \quad \subseteq \quad \boldsymbol{\mathcal{C}}(\Psi_k) \cap \boldsymbol{\mathcal{D}}(k-1).$$

That is only local solutions in $\boldsymbol{\mathcal{D}}(k-1)$ *are in* $\boldsymbol{\mathcal{D}}(k)$.

A solution step $\xrightarrow{\Psi_k}$ *is* **(globally) sound** *if* $\Psi = \{C_1, C_2, \ldots, C_r\}$*, and*

$$\boldsymbol{\mathcal{D}}(k) \quad \subseteq \quad \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(k-1).$$

That is only global solutions in $\boldsymbol{\mathcal{D}}(k-1)$ *are in* $\boldsymbol{\mathcal{D}}(k)$. □

The property of *completeness* of a solution step is significant as it means that no solutions to the problem have been lost as a result of the solution step. In some circumstances we may only want one solution to a problem, in which case completeness is not an issue, and we need only enforce consistency.

Definition 6.5 (Complete solution step) *A solution step* $\xrightarrow{\Psi_k}$ *is* **locally complete** *if* $\Psi_k = \{C_{\Psi_{k_1}}, C_{\Psi_{k_2}}, \ldots, C_{\Psi_{k_m}}\}$*, and*

$$\boldsymbol{\mathcal{C}}(\Psi_k) \cap \boldsymbol{\mathcal{D}}(k-1) \quad \subseteq \quad \boldsymbol{\mathcal{D}}(k).$$

That is all local solutions in $\boldsymbol{\mathcal{D}}(k-1)$ *are in* $\boldsymbol{\mathcal{D}}(k)$.

A solution step $\xrightarrow{\Psi_k}$ *is* **(globally) complete** *if* $\Psi = \{C_1, C_2, \dots, C_r\}$, *and*

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(k-1) \subseteq \boldsymbol{\mathcal{D}}(k).$$

That is all global solutions in $\boldsymbol{\mathcal{D}}(k-1)$ *are in* $\boldsymbol{\mathcal{D}}(k)$. □

Example 6.4 (Properties of solution steps) The solution step $\xrightarrow{LS1_a=0}$ in example 6.3 is locally consistent as the constraint $LS1_a = 0$ is satisfied in $\boldsymbol{\mathcal{D}}(1)$. It is also locally sound, as there are no configurations in $\boldsymbol{\mathcal{D}}(1)$ that do not satisfy $LS1_a = 0$. It is locally complete as well, as any configuration in $\boldsymbol{\mathcal{D}}(0)$ not in $\boldsymbol{\mathcal{D}}(1)$ will *not* satisfy the constraint $LS1_a = 0$.

Unfortunately, it is much harder to gauge global properties of the solution step for this example. Without knowing what the solutions of the problem are, it is not usually possible to say in this way whether the solution step is consistent, sound or complete. In fact, the solutions of G' are known. They are

$$\begin{aligned}
&\{\{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,4,0)\}, LS3 = \{(4,0,0,0)\},\\
&LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},\\
&LS2_b = \{(4,0)\}, LS3_a = \{(4,0)\}, LS3_b = \{(0,0)\}\},\\
&\{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,-4,0)\}, LS3 = \{(-4,0,0,0)\},\\
&LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},\\
&LS2_b = \{(-4,0)\}, LS3_a = \{(-4,0)\}, LS3_b = \{(0,0)\}\}\}.
\end{aligned}$$

Thus the solution step is, in fact, (globally) consistent, sound and complete. However, it is not usually easy to find *all* of the solutions to a problem and consequently, it is not usually possible to say whether a solution step has a global property. □

6.2.3 Solution processes

A *solution process* is constructed from a sequence of solution steps. A solution process forms the basis for most constraint satisfaction algorithms.

Definition 6.6 (Solution process) *Given constraint problem* $P = (\Phi, \Psi)$, *with* Φ *a set of entities,* $\{(x_1, D_1) \dots, (x_n, D_n)\}$ *with domains* $D_1, \dots, D_n$, Ψ *a set of*

constraints on Φ *and* $\Psi' \subseteq \Psi$, *a* solution process $\xrightarrow{\Psi'}{}^{*}$ *is a sequence of mappings*

$$\begin{aligned} \boldsymbol{\mathcal{D}}(0) = (D_1 \times \cdots \times D_n) & \xrightarrow{\Psi_1} \boldsymbol{\mathcal{D}}(1) \\ & \xrightarrow{\Psi_2} \boldsymbol{\mathcal{D}}(2) \\ & \ldots \\ & \xrightarrow{\Psi_\kappa} \boldsymbol{\mathcal{D}}(\kappa) \end{aligned}$$

such that

$$\kappa \text{ is finite,}$$
$$\Psi_1, \Psi_2, \ldots, \Psi_\kappa \subseteq \Psi',$$
$$\Psi_i \cap \Psi_j = \emptyset, \forall i, j, i \neq j, i = 1..\kappa, j = 1..\kappa,$$
$$\bigcup_{i=1}^{m} \Psi_i = \Psi'.$$

□

Example 6.5 (Solution process) Constraint solvers such as degrees of freedom analysis, D-Cubed, Erep, ICBSM, FC, MAC are solution processes.

For example, solving constraint problem G' of example 4.10 using degrees of

freedom analysis may give a solution process of the form below.

$$
\begin{aligned}
& \boldsymbol{\mathcal{D}}(0) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4\ LS1_a = \mathbb{R}^2, \\
& LS1_b = \mathbb{R}^2, LS2_a = \mathbb{R}^2, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, LS3_b = \mathbb{R}^2\} \\
\xrightarrow{LS1_a=0}\ & \boldsymbol{\mathcal{D}}(1) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \mathbb{R}^2, LS2_a = \mathbb{R}^2, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, LS3_b = \mathbb{R}^2\} \\
\xrightarrow{LS1_b=LS2_a}\ & \boldsymbol{\mathcal{D}}(2) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \mathbb{R}^2, LS3_a = \mathbb{R}^2, \\
& LS3_b = \mathbb{R}^2, x, y \in \mathbb{R}\} \\
\xrightarrow{LS2_b=LS3_a}\ & \boldsymbol{\mathcal{D}}(3) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \{(z,w)\}, \\
& LS3_a = \{(z,w)\}, LS3_b = \mathbb{R}^2, x, y, z, w \in \mathbb{R}\} \\
\xrightarrow{LS3_b=LS1_a}\ & \boldsymbol{\mathcal{D}}(4) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \{(z,w)\}, \\
& LS3_a = \{(z,w)\}, LS3_b = \{(0,0)\}, x, y, z, w \in \mathbb{R}\} \\
\xrightarrow{d(LS1_a,LS1_b)=3}\ & \boldsymbol{\mathcal{D}}(5) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \{(z,w)\}, \\
& LS3_a = \{(z,w)\}, LS3_b = \{(0,0)\}, x, y, z, w \in \mathbb{R}, x^2 + y^2 = 9\} \\
\xrightarrow{d(LS2_a,LS2_b)=5}\ & \boldsymbol{\mathcal{D}}(6) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \{(z,w)\}, \\
& LS3_a = \{(z,w)\}, LS3_b = \{(0,0)\}, x, y, z, w \in \mathbb{R}, x^2 + y^2 = 9, \\
& (x-z)^2 + (y-w)^2 = 25\} \\
\xrightarrow{d(LS3_a,LS3_b)=4}\ & \boldsymbol{\mathcal{D}}(7) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(x,y)\}, LS2_a = \{(x,y)\}, LS2_b = \{(z,w)\}, \\
& LS3_a = \{(z,w)\}, LS3_b = \{(0,0)\}, x, y, z, w \in \mathbb{R}, x^2 + y^2 = 9, \\
& (x-z)^2 + (y-w)^2 = 25, z^2 + w^2 = 16\} \\
\xrightarrow{LS2=(0,3)}\ & \boldsymbol{\mathcal{D}}(8) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, \\
& LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\}, LS2_b = \{(\pm 4,0)\}, \\
& LS3_a = \{(\pm 4,0)\}, LS3_b = \{(0,0)\}\},
\end{aligned}
$$

$$\xrightarrow{endpoint(LS1,LS1_a)} \boldsymbol{\mathcal{D}}(9) = \{LS1 = \{(0,0,\mathbb{R},\mathbb{R})\}, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4,$$
$$LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},$$
$$LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\}, LS3_b = \{(0,0)\}\},$$
$$\xrightarrow{endpoint(LS1,LS1_b)} \boldsymbol{\mathcal{D}}(10) = \{LS1 = \{(0,0,0,3)\}, LS2 = \mathbb{R}^4, LS3 = \mathbb{R}^4,$$
$$LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},$$
$$LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\}, LS3_b = \{(0,0)\}\},$$
$$\xrightarrow{endpoint(LS2,LS2_a)} \boldsymbol{\mathcal{D}}(11) = \{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,\mathbb{R},\mathbb{R})\},$$
$$LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},$$
$$LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\}, LS3_b = \{(0,0)\}\},$$
$$\xrightarrow{endpoint(LS2,LS2_b)} \boldsymbol{\mathcal{D}}(12) = \{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,\pm 4,0)\},$$
$$LS3 = \mathbb{R}^4, LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\}, LS2_a = \{(0,3)\},$$
$$LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\}, LS3_b = \{(0,0)\}\},$$
$$\xrightarrow{endpoint(LS3,LS3_a)} \boldsymbol{\mathcal{D}}(13) = \{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,\pm 4,0)\},$$
$$LS3 = \{(\pm 4,0,\mathbb{R},\mathbb{R})\}, LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\},$$
$$LS2_a = \{(0,3)\}, LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\},$$
$$LS3_b = \{(0,0)\}\},$$
$$\xrightarrow{endpoint(LS3,LS3_b)} \boldsymbol{\mathcal{D}}(14) = \{LS1 = \{(0,0,0,3)\}, LS2 = \{(0,3,\pm 4,0)\},$$
$$LS3 = \{(\pm 4,0,0,0)\}, LS1_a = \{(0,0)\}, LS1_b = \{(0,3)\},$$
$$LS2_a = \{(0,3)\}, LS2_b = \{(\pm 4,0)\}, LS3_a = \{(\pm 4,0)\},$$
$$LS3_b = \{(0,0)\}\}.$$

Note that

$$\boldsymbol{\mathcal{D}}(14) \subseteq \boldsymbol{\mathcal{D}}(13) \subseteq \cdots \subseteq \boldsymbol{\mathcal{D}}(1) \subseteq \boldsymbol{\mathcal{D}}(0),$$

as required. □

6.2.3.1 Solution processes always head towards a solution

In most cases, it is desirable for a solution process to head towards a solution space that is manageable. In terms of dimension, this will involve having a solution space whose dimension is as small as possible. The actual size of the solution space found will depend on the algorithm and problem solved. However, it is possible to prove that the dimension of a solution space tends to decrease due to a solution step.

Lemma 6.1 proves this.

Corollary 6.1 $A \subseteq B \Rightarrow dim(A) \leq dim(B)$. □

Proof Assume $A \subseteq B$, then $B = A \cup C$, for some C

$$\begin{aligned} A \subseteq B &\Rightarrow B = A \cup C \\ &\Rightarrow dim(B) = dim(A \cup C) \\ &= \max(dim(A), dim(C)) \\ &\geq dim(A). \end{aligned}$$

□

Lemma 6.1 *If there is a solution step*

$$\boldsymbol{\mathcal{D}}(i-1) \xrightarrow{\Psi_i} \boldsymbol{\mathcal{D}}(i)$$

then

$$dim(\boldsymbol{\mathcal{D}}(i)) \leq dim(\boldsymbol{\mathcal{D}}(i-1)).$$

□

Proof Immediate since

$$\boldsymbol{\mathcal{D}}(i) \subseteq \boldsymbol{\mathcal{D}}(i-1) \Rightarrow dim(\boldsymbol{\mathcal{D}}(i)) \leq dim(\boldsymbol{\mathcal{D}}(i-1)).$$

□

6.2.4 Solution process properties

The definitions for solution properties for solution *steps* given in section 6.2.2 do not necessarily allow corresponding properties to be inferred about solution *processes*. In fact, a solution process can be interpreted as a single solution step as lemma 6.2 demonstrates.

Lemma 6.2 *A solution process* $\boldsymbol{\mathcal{D}}(0) \xrightarrow{\Psi}{}^{*} \boldsymbol{\mathcal{D}}(\kappa)$ *is a single solution step.* □

Proof A solution step is a function f such that

$$\boldsymbol{\mathcal{D}}(k) = f(\Psi_k, \boldsymbol{\mathcal{D}}(k-1)).$$

A solution process is a sequence of functions f_k such that

$$\boldsymbol{\mathcal{D}}(k) = f_k(\Psi_k, \boldsymbol{\mathcal{D}}(k-1)), \quad k = 1..\kappa.$$

Thus, for solution process $\longrightarrow^*$

$$\begin{aligned}
\boldsymbol{\mathcal{D}}(\kappa) &= f_\kappa(\Psi_\kappa, \boldsymbol{\mathcal{D}}(\kappa-1)), \\
\boldsymbol{\mathcal{D}}(\kappa-1) &= f_{\kappa-1}(\Psi_{\kappa-1}, \boldsymbol{\mathcal{D}}(\kappa-2)), \\
&\dots \\
\boldsymbol{\mathcal{D}}(1) &= f_1(\Psi_1, \boldsymbol{\mathcal{D}}(0)).
\end{aligned}$$

For each f_k, create f'_k such that

$$f'_k(\Psi, \boldsymbol{\mathcal{D}}(k-1)) = f_k(\Psi_k, \boldsymbol{\mathcal{D}}(k-1)).$$

Then it is possible to redefine $\boldsymbol{\mathcal{D}}(\kappa)$ so that

$$\boldsymbol{\mathcal{D}}(\kappa) = f'_\kappa\left(\Psi, f'_{\kappa-1}\left(\Psi, \dots f'_1\left(\Psi, \boldsymbol{\mathcal{D}}(0)\right)\right)\right).$$

Since a composition of functions is a function, there exists a function g such that

$$\boldsymbol{\mathcal{D}}(\kappa) = g(\Psi, \boldsymbol{\mathcal{D}}(0)).$$

It remains only to show that

$$\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{D}}(0)$$

which follows immediately due to the nature of the solution process. $\Box$

Consequently, it is natural to wish to ascribe certain properties to solution processes. However, since processes can be interpreted as both a single solution step *and* as a sequence of solution steps, there exist two possible ways of defining the properties desired.

One method is to interpret a solution process as a solution step and this gives rise to *consistency, soundness* and *completeness of solution processes* and the corresponding local properties.

The other method is to interpret a solution process as a sequence of solution steps and examine the properties of *each* solution step in the process. This gives rise to *strong consistency, strong soundness* and *strong completeness of solution processes*

as well as corresponding local properties.

Definition 6.7 (Solution process consistency) *A solution process α is* **(globally) consistent** *iff for all constraint problems $P = (\Phi, \Psi)$,*

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is consistent, where $\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}$ and $\Psi = \{C_1, \ldots, C_r\}$. That is

$$\boldsymbol{\mathcal{C}}(\Psi) \neq \emptyset \Rightarrow \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(\kappa) \neq \emptyset.$$

A solution process α is **(globally) strongly consistent** *iff for all constraint problems P, each solution step is consistent.*

Process α is **locally consistent** *iff for all constraint problems P and subsets $\Psi' \subseteq \Psi$,*

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is locally consistent. That is, for $\Psi' = \{C_{\Psi'_1}, \ldots, C_{\Psi'_m}\}$ and

$$\boldsymbol{\mathcal{C}}(\Psi') \neq \emptyset \Rightarrow \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(\kappa) \neq \emptyset.$$

Process α is **strongly locally consistent** *iff for all constraint problems P, each solution step is locally consistent.* $\Box$

Definition 6.8 (Solution process soundness) *A solution process α is* **(globally) sound** *iff for all constraint problems $P = (\Phi, \Psi)$,*

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is sound, where $\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}$ and $\Psi = \{C_1, \ldots, C_r\}$. That is,

$$\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap (D_1 \times \cdots \times D_n).$$

A solution process α is **(globally) strongly sound** *iff for all constraint problems P, each solution step is sound.*

Solution process α is **locally sound** *iff for all constraint problems P and subsets*

$\Psi' \subseteq \Psi$,

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is sound. That is, $\Psi' = \{C_{\Psi'_1}, \ldots, C_{\Psi'_m}\}$ *and*

$$\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi') \cap (D_1 \times \cdots \times D_n)\,.$$

Solution process α *is* **strongly locally sound** *iff for all constraint problems* P*, each solution step is locally sound.* □

Definition 6.9 (Solution process completeness) *A solution process* α *is* **(globally) complete** *iff for all constraint problems* $P = (\Phi, \Psi)$,

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is complete, where $\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}$ *and* $\Psi = \{C_1, \ldots, C_r\}$*. That is,*

$$\boldsymbol{\mathcal{C}}(\Psi) \cap (D_1 \times \cdots \times D_n) \subseteq \boldsymbol{\mathcal{D}}(\kappa).$$

Solution process α *is* **(globally) strongly complete** *iff for all constraint problems* P*, each solution step is complete.*

Solution process α *is* **locally complete** *iff for all constraint problems* P *and subsets* $\Psi' \subseteq \Psi$,

$$(D_1 \times \cdots \times D_n) \stackrel{\Psi'}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$$

is locally complete. That is, $\Psi' = \{C_{\Psi'_1}, \ldots, C_{\Psi'_m}\}$ *and*

$$\boldsymbol{\mathcal{C}}(\Psi') \cap (D_1 \times \cdots \times D_n) \subseteq \boldsymbol{\mathcal{D}}(\kappa).$$

Solution process α *is* **strongly locally complete** *iff for all constraint problems* P*, each solution step is locally complete.* □

Example 6.6 (Degrees of freedom analysis) The most desirable solution process would be one that was consistent, sound and complete as the terminal solution space would consist only of solutions and would contain all solutions. However, it is extremely difficult to prove that a process has a property directly. Although it

seems likely that degrees of freedom analysis is sound, for example, there is no easy way of justifying that statement.

On the other hand, it is possible to examine a single, arbitrary, solution step. In degrees of freedom analysis, for example, the solution step

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$$

is always locally sound as only solutions to Ψ_k are retained in $\boldsymbol{\mathcal{D}}(k)$. Since an arbitrary solution step is locally sound, each solution step in a sequence is locally sound. Correspondingly, degrees of freedom analysis is strongly, locally sound. □

Unfortunately, without a method of linking strong local properties to global properties, knowing that a process is strongly, locally sound is not particularly useful.

6.2.5 Using local properties to draw conclusions about processes

As discussed in section 6.2.4, it is desirable to be able to say whether a solution process has a certain global property. Unfortunately, it is not usually possible to make concrete statements about global properties directly. It is, however, frequently possible to comment on local properties of individual solution steps.

Theorem 6.1 provides the link between strong local properties and global properties. Theorem 6.1 means, for example, that if a solution process is strongly, locally complete, then it is globally complete. Theorem 6.1 allows concrete statements to be made about solution processes.

Theorem 6.1 For solution process $\longrightarrow^*$,

a. Strongly Consistent $\Leftrightarrow$ Consistent.

b. Locally Consistent $\Rightarrow$ Strongly Locally Consistent.

Strongly Locally Consistent $\not\Rightarrow$ Locally Consistent.

c. Locally Consistent $\Leftrightarrow$ Consistent.

d. Strongly Sound $\Leftrightarrow$ Sound.

Sound $\not\Rightarrow$ Strongly Sound.

e. Locally Sound $\not\Rightarrow$ Strongly Locally Sound.

Strongly Locally Sound $\Rightarrow$ Locally Sound.

f. Locally Sound $\Leftrightarrow$ Sound.

g. Strongly Complete $\Leftrightarrow$ Complete.

h. Locally Complete $\not\Rightarrow$ Strongly Locally Complete.

Strongly Locally Complete $\Rightarrow$ Locally Complete.

i. Locally Complete $\Leftrightarrow$ Complete.

□

Proof Proof is deferred to appendix C. □

Example 6.7 (Degrees of freedom analysis) Since degrees of freedom analysis is strongly, locally sound, by theorem 6.1, it is globally sound. This means that any configuration in a terminal solution space found by Degrees of Freedom Analysis is a solution. □

6.2.6 Consequences of the Local-Global Theorem

Theorem 6.1 forms a significant contribution to the understanding of constraint solvers. The quality of solution of a constraint process is an important issue for designers. The three properties of consistency, soundness and completeness capture the concepts of *a* solution, *only* solutions and *all* solutions and as such describe vital properties of constraint solvers.

For example, if a designer is using a sound constraint solver, then they can guarantee that any results from the solver are solutions to the constraint problem. However, if the constraint solver is not consistent, then the designer cannot draw any conclusions from the fact that the solver failed to find a solution to the problem.

Previously, it was difficult to be able to make concrete statements about constraint solvers. Statements about the properties of finite domain constraint solvers depended on being able to say that the solver exhaustively searched the solution space. Owen [86] used Galois theory to prove that the DCM algorithm was sound and complete. Theorem 6.1 allows for significantly simpler proofs of the properties of constraint solvers as demonstrated in example 6.7.

However, the power of theorem 6.1 does not lie solely in the ability to state whether simple constraint solvers are consistent, sound or complete. *Hybrid* constraint solvers can also be studied using theorem 6.1 and this allows several important and interesting conclusions to be drawn about existing hybrid solvers and also about hybrids in general. For example, INCES [62], IGCS [112] and MechEdit [15] are *not* consistent. The use of theorem 6.1 to study hybrid constraint solvers is discussed in more detail in section 7.4.

6.3 Enrichment of the constraint satisfaction framework

Although the constraint satisfaction framework in section 6.2 is rich enough to describe many constraint solvers, there exist some techniques that cannot be described using the framework as it stands. In fact, the techniques of constraint priorities, backtracking, variable-driven and incremental satisfaction are powerful enhancements of the general constraint problem. The satisfaction framework's inability to handle them unaltered should not be taken as a weakness. Just as the definition of what a constraint problem is must be enhanced to describe these cases, the basic satisfaction framework can be enhanced to describe all four special cases and this is an indication of the power of the framework.

This section describes the enhancements that can be made to the constraint satisfaction framework in order to incorporate constraint priorities, variable-driven, backtracking and incremental satisfaction.

6.3.1 Constraint priorities

Borning *et al.* [11] introduced the concept of constraint priorities in order to allow over constrained problems to be solved. Intuitively, a *strength* is associated with each constraint. This strength is an indication of how important it is that the constraint be satisfied. Thus, less important constraints are sacrificed and not satisfied so that the more important constraints are satisfied. This is a means of dealing with problems that are over-constrained.

Definition 6.10 (Constraint priority problem) *Given an ordering $\alpha_0 > \alpha_1 > \ldots > \alpha_z$ of strengths, a* **constraint priority problem** *is a pair (Φ, Ψ), where Φ is a set of entities and Ψ a set of constraints. The set Ψ is enhanced so that each member of Ψ is a pair (C, β), where C is a constraint and $\beta \in \{\alpha_0, \ldots, \alpha_z\}$ is the strength associated with constraint C.*

□

The concept of a *solution* to a constraint priority problem is somewhat different to the concept given in definition 4.5.

Definition 6.11 (Solution to a constraint priority problem) *Given constraint priority problem*

$$P = (\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}, \Psi = \{(C_1, \beta_1), \ldots, (C_m, \beta_m)\}),$$

define sets of constraints $\mathcal{H}_i$ as follows

$$\begin{aligned} \mathcal{H}_i &= \{(C_j, \beta_j) | \beta_j = i, (C_j, \beta_j) \in \Psi\}, \\ \mathcal{H} &= \bigcup_i \mathcal{H}_i. \end{aligned}$$

Effectively this gathers the constraints into sets of equal strength. Given a comparator, better, configuration u is a solution *to P iff*

$$\begin{aligned} \mathcal{S}_0 &= \{v | \forall (C_j, \beta_0) \in \mathcal{H}_0, v \in C_j\} = \boldsymbol{\mathcal{C}}(\mathcal{H}_0), \\ \mathcal{S} &= \{v | v \in \mathcal{S}_0 \wedge \forall w \in \mathcal{S}_0, \neg better(w, v, \mathcal{H})\}, \\ u &\in \mathcal{S}. \end{aligned}$$

□

The choice of comparator *better* significantly affects the solutions to the problem. Borning *et al.* [11] give examples of several such comparators and demonstrate the differences in the solution spaces for various problems. Consequently, this will not be gone into in detail here. However, it is worth noting that constraints with strength α_0 (called *required* in [11]) will always be satisfied if possible.

Proposition 6.1 If possible, all constraints with priority α_0 will be satisfied. □

Proof Any constraint with priority α_0 will be in the set $\mathcal{S}_0$. By the construction of set $\mathcal{S}$, any solution to the constraint priority problem must be in $\boldsymbol{\mathcal{C}}(\mathcal{H}_0)$ and consequently must satisfy all constraints with priority α_0. □

The properties associated with solution steps and processes are slightly different, as it is possible for a new constraint to be processed incrementally and for this to alter the solution space. For example, consider the constraint priority problem $P = (\{(a, \mathbb{R}), (b, \mathbb{R})\}, \{(a = 2, \mathit{required}), (b = 2, \mathit{weak}), (b = 3, \mathit{strong})\})$. The initial solution space is $\boldsymbol{\mathcal{D}}(0) = \mathbb{R}^2$. Suppose that there exists a solution process as follows

$$\begin{aligned} \boldsymbol{\mathcal{D}}(0) &\xrightarrow{a=2} \boldsymbol{\mathcal{D}}(1) = \{a = \{2\}, b = \mathbb{R}\} \\ &\xrightarrow{b=2} \boldsymbol{\mathcal{D}}(2) = \{a = \{2\}, b = \{2\}\} \\ &\xrightarrow{b=3} \boldsymbol{\mathcal{D}}(3). \end{aligned}$$

Since the strength of the constraint $b = 3$ is greater than the strength of the constraint $b = 2$, the former should be satisfied to the detriment of the latter. However, this violates the definition of a solution process as no matter what choice is made to satisfy $b = 3$, $\boldsymbol{\mathcal{D}}(3) \not\subseteq \boldsymbol{\mathcal{D}}(2)$.

The definitions of solution steps and processes are much the same for constraint priority problems as defined in section 6.2, except that property 6.1 is relaxed as follows. Thus, each solution space in a process is only a subset of the initial solution space and not the previous solution space, as finding solutions to a new constraint may violate previous constraints without upsetting the constraint process.

Definition 6.12 (Solution step for constraint priority problems) *For constraint priority problem* $P = (\Phi, \Psi)$ *a* **solution step** *is a mapping* $\xrightarrow{\Psi_k}$,

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$$

where $\Psi_k \subseteq \Psi$ *is a set of constraints and* $\boldsymbol{\mathcal{D}}(i)$ *is the solution space of the set of entities* Φ *at step* i*. For each step,*

$$\boldsymbol{\mathcal{D}}(k) \subseteq \boldsymbol{\mathcal{D}}(0).$$

□

For solution step properties, the comparator *better* is used to indicate correct solutions. The case of consistent solution steps is presented here; sound and complete solution steps and solution processes are similar.

Definition 6.13 (Consistent solution step for set of constraints) *Given a solution process for constraint priority problem* $P = (\Phi, \Psi)$*, with* $\Psi' \subseteq \Psi$*,*

$$\begin{array}{rcl} (D_1 \times \cdots \times D_n) & \overset{\Psi'}{\longrightarrow}^{*} & \boldsymbol{\mathcal{D}}(j-1) \\ & \overset{\Upsilon}{\to} & \boldsymbol{\mathcal{D}}(j) \end{array}$$

where $\Upsilon = \{(C_1, \beta_1), \ldots, (C_k, \beta_k)\}$*, with comparator better, the solution step,*

$$\boldsymbol{\mathcal{D}}(j-1) \overset{\Upsilon}{\to} \boldsymbol{\mathcal{D}}(j)$$

is **(globally) consistent** *if*

$$\mathcal{S}_0 \neq \emptyset \Rightarrow \mathcal{S} \cap \boldsymbol{\mathcal{D}}(j) \neq \emptyset.$$

□

6.3.2 Variable-driven satisfaction

The constraint satisfaction framework presented in section 6.2 can be described as *constraint-driven*; as the solution progresses, a set of constraints is chosen to be processed to produce the next solution space. Thus, the choice of constraints *drives* the solution process. However, finite domain algorithms such as FC typically work slightly differently. Finite domain techniques can be described as *variable-driven* as it is the choice of variable to be processed next that drives the solution process. It is relatively easy to describe the variable-driven constraint satisfaction process. Only the variable-driven solution step is described here; extension to the other structures of the constraint satisfaction process is trivial.

Definition 6.14 (Solution step (variable-driven)) *For constraint problem $P = (\Phi, \Psi)$, a* **(variable-driven) solution step** *is a mapping* $\xrightarrow{\Phi_k}$

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Phi_k} \boldsymbol{\mathcal{D}}(k)$$

where $\Phi_k \subseteq \Phi$ is a set of variables that are instantiated with subsets of their appropriate domains. □

In fact, variable-driven satisfaction is a subset of constraint-driven satisfaction as the variable chosen for the variable-driven solution step is instantiated with a value. For example, if a variable x in Φ_k is assigned the set $\{v\}$, then this is a constraint of the form $x = v$ and can easily be described using a constraint-driven solution step. However, the constraints processed in variable-driven satisfaction are not normally known when the constraint problem is created and are added dynamically as the constraint problem is solved. This means that variable-driven constraint satisfaction involves some degree of incremental satisfaction (see section 6.3.4).

6.3.3 Backtracking

Backtracking is a popular method of searching through a solution space. Finite domain constraint solvers use backtracking extensively. Backtracking itself is very simple to describe, but its inclusion in the basic satisfaction framework complicates descriptions of proofs and for this reason it is included here. Backtracking involves undoing a solution step and restoring the previous solution space. In some techniques, such as backjumping, it is necessary to jump back several solution steps. Consequently the definition of backtracking allows undoing several steps.

Definition 6.15 (Backtracking) *For constraint problem $P = (\Phi, \Psi)$, with $\Phi = \{(x_1, D_1), \ldots, (x_n, D_n)\}$, assume there exists a sequence of solution steps*

$$\begin{aligned}(D_1 \times \cdots \times D_n) &\xrightarrow{\Psi_1} \boldsymbol{\mathcal{D}}(1) \\ &\xrightarrow{\Psi_2} \boldsymbol{\mathcal{D}}(2) \\ &\vdots \\ &\xrightarrow{\Psi_r} \boldsymbol{\mathcal{D}}(r)\end{aligned}$$

with $\Psi_i \subseteq \Psi, \quad i = 1..r$.

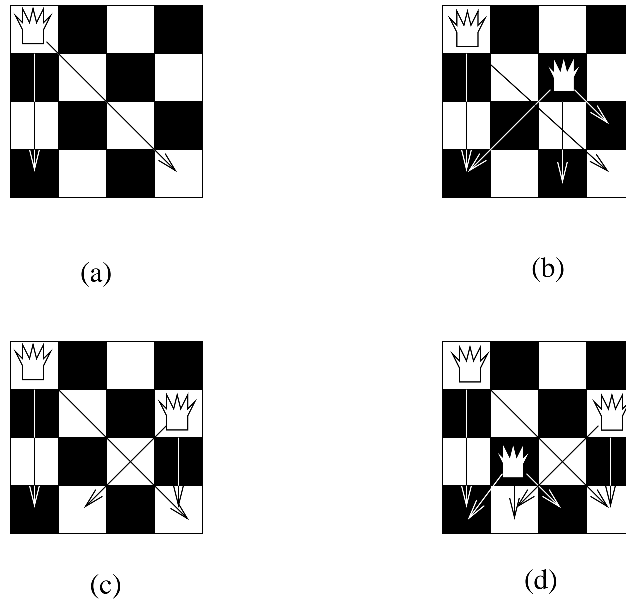


Figure 6.1: Solving the 4 queens problem

Then the solution process **backtracks** *by s steps by introducing the step*

$$\boldsymbol{\mathcal{D}}(r) \xleftarrow{\Psi_r, \Psi_{r-1}, \ldots, \Psi_{(r-s+1)}} \boldsymbol{\mathcal{D}}(r-s).$$

The process can then proceed with a different choice of Ψ_{r-s+1}. □

Example 6.8 (Finite domain backtracking) Consider the 4-queens problem, defined in example 4.5. This involves placing 4 queens on a 4 by 4 chessboard. The initial solution space, $\boldsymbol{\mathcal{D}}(0)$, is that none of the 4 queens has been placed on the chessboard. A solution step could be placing the first queen in the first column (see figure 6.1 (a)),

$$\boldsymbol{\mathcal{D}}(0) \xrightarrow{Q1:=1} \boldsymbol{\mathcal{D}}(1) = \{Q1 = \{1\}, Q2 = D_4, Q3 = D_4, Q4 = D_4\}.$$

The next solution step could then involve placing the second queen so that it does not attack the first queen, for example in the third column, (figure 6.1 (b)),

$$\boldsymbol{\mathcal{D}}(1) \xrightarrow{Q2:=3} \boldsymbol{\mathcal{D}}(2) = \{Q1 = \{1\}, Q2 = \{3\}, Q3 = D_4, Q4 = D_4\}.$$

However, it is now not possible to place the third queen without attacking any other queen. The solution process has reached a dead end. Finite domain search techniques backtrack at this point to a previous state. Since there is another choice

for $Q2$, there are another two possible steps, (see figure 6.1 (c)),

$$\begin{array}{lll} \boldsymbol{\mathcal{D}}(2) & \xleftarrow{Q2:=3} & \boldsymbol{\mathcal{D}}(1) \\ & \xrightarrow{Q2:=4} & \boldsymbol{\mathcal{D}}(3) = \{Q1 = \{1\}, Q2 = \{4\}, Q3 = D_4, Q4 = D_4\}. \end{array}$$

There is one possible choice for positioning $Q3$, (see figure 6.1 (d)),

$$\boldsymbol{\mathcal{D}}(3) \xrightarrow{Q3:=3} \boldsymbol{\mathcal{D}}(4) = \{Q1 = \{1\}, Q2 = \{4\}, Q3 = \{3\}, Q4 = D_4\}.$$

However, there are no positions to place $Q4$ without violating the constraints. Since there are no further choices for $Q3$ or $Q2$ it is necessary to backtrack further,

$$\boldsymbol{\mathcal{D}}(4) \xleftarrow{Q3:=3, Q2:=4} \boldsymbol{\mathcal{D}}(1),$$

and proceed with another choice of $Q1$. Thus the solution process so far is

$$\begin{array}{lll} \boldsymbol{\mathcal{D}}(0) & \xrightarrow{Q1:=1} & \boldsymbol{\mathcal{D}}(1) = \{Q1 = \{1\}, Q2 = D_4, Q3 = D_4, Q4 = D_4\} \\ \boldsymbol{\mathcal{D}}(1) & \xrightarrow{Q2:=3} & \boldsymbol{\mathcal{D}}(2) = \{Q1 = \{1\}, Q2 = \{3\}, Q3 = D_4, Q4 = D_4\} \\ \boldsymbol{\mathcal{D}}(2) & \xleftarrow{Q2:=3} & \boldsymbol{\mathcal{D}}(1) \\ & \xrightarrow{Q2:=4} & \boldsymbol{\mathcal{D}}(3) = \{Q1 = \{1\}, Q2 = \{4\}, Q3 = D_4, Q4 = D_4\} \\ & \xrightarrow{Q3:=3} & \boldsymbol{\mathcal{D}}(4) = \{Q1 = \{1\}, Q2 = \{4\}, Q3 = \{3\}, Q4 = D_4\} \\ & \xleftarrow{Q3:=3, Q2:=4} & \boldsymbol{\mathcal{D}}(1) \\ & \ldots & \end{array}$$

□

Currently, the introduction of backtracking to a solution process means that theorem 6.1 cannot be applied to the process as $\boldsymbol{\mathcal{D}}(k) \not\subseteq \boldsymbol{\mathcal{D}}(k-1)$ after applying a backtracking solution step. However, since all that a backtracking solution step does is reintroduce an old solution step, it is not hard to see how theorem 6.1 can be generalised to cover backtracking.

6.3.4 Incremental satisfaction

Many constraint solvers use an incremental paradigm to improve efficiency, such as [27, 34, 62, 94, 112]. The incremental paradigm means that as new constraints are added they are solved immediately, rather than waiting for the whole constraint

problem to be defined and then solved. The solution process framework presented above easily captures the concept of incremental addition of constraints as this is an integral part of the framework. As each constraint is added, the next solution step is to process the newly added constraint to find the next solution space. Thus, constraint C is added incrementally to constraint problem $P = (\Phi, \Psi)$,

$$\begin{aligned} \boldsymbol{\mathcal{D}}(0) &\stackrel{\Psi}{\rightarrow} \boldsymbol{\mathcal{D}}(k) \\ &\stackrel{C}{\rightarrow} \boldsymbol{\mathcal{D}}(k+1), \end{aligned}$$

and constraint problem $P' = (\Phi, \Psi \cup C)$ is formed.

Adding a new entity, E, is slightly more complex as this will affect the rest of the constraint problem and the solution spaces. The constraint problem is altered so that the new entity is added to Φ to give $\Phi' = \Phi \cup E$. Since constraints are already defined as enhanced constraints with respect to Φ (section 4.7), the new set of constraints, Ψ' is given by

$$\Psi' = \{C \times D_E | C \in \Psi\}.$$

These changes become part of the solution process, so that when a new entity is added to the constraint problem, a new solution space is calculated to take into account the new entity.

$$\boldsymbol{\mathcal{D}}(k-1) \stackrel{E}{\rightarrow} \boldsymbol{\mathcal{D}}(k) = \boldsymbol{\mathcal{D}}(k-1) \times D_E.$$

Example 6.9 (Incremental geometric constraint satisfaction) Consider solving problem G of example 4.8. In an incremental solver, such as ICBSM [27], the initial constraint problem would be an empty set and constraints and entities would be added to it. Thus, the initial problem $P_1 = (\emptyset, \emptyset)$ and the initial solution space, $\boldsymbol{\mathcal{D}}(0) = \emptyset$. If line segment $LS1$ is added to the problem, then $P_2 = (\{LS1\}, \emptyset)$ and solution space $\boldsymbol{\mathcal{D}}(1) = D_{LS1} = \{LS1 = \mathbb{R}^4\}$.

A possible solution process then would be of the form:

$$
\begin{aligned}
\boldsymbol{\mathcal{D}}(0) \quad & \xrightarrow{LS1} \quad \boldsymbol{\mathcal{D}}(1) = \{LS1 = \mathbb{R}^4\} \\
& \xrightarrow{LS2} \quad \boldsymbol{\mathcal{D}}(2) = \{LS1 = \mathbb{R}^4, LS2 = \mathbb{R}^4\} \\
& \xrightarrow{LS1_a=0} \quad \boldsymbol{\mathcal{D}}(3) = \{LS1 = \{(0, 0, \mathbb{R}, \mathbb{R})\}, LS2 = \mathbb{R}^4\} \\
& \xrightarrow{LS1_b=LS2_a} \quad \boldsymbol{\mathcal{D}}(4) = \{LS1 = \{(0, 0, x, y)\}, LS2 = \{(x, y, \mathbb{R}, \mathbb{R})\}, \\
& \qquad\qquad x, y \in \mathbb{R}\} \\
& \xrightarrow{LS3} \quad \boldsymbol{\mathcal{D}}(5) = \{LS1 = \{(0, 0, x, y)\}, LS2 = \{(x, y, \mathbb{R}, \mathbb{R})\}, \\
& \qquad\qquad LS3 = \mathbb{R}^4, x, y \in \mathbb{R}\} \\
& \xrightarrow{LS2_b=LS3_a} \quad \boldsymbol{\mathcal{D}}(6) = \{LS1 = \{(0, 0, x, y)\}, LS2 = \{(x, y, z, w)\}, \\
& \qquad\qquad LS3 = \{(z, w, \mathbb{R}, \mathbb{R})\}, x, y, z, w \in \mathbb{R}\} \\
& \quad \ldots
\end{aligned}
$$

□

6.4 Conclusions

This chapter has presented a formal abstraction of the constraint satisfaction process. This abstraction is sufficiently rich to describe all of the constraint solvers described in chapter 2. The concept of *solution spaces* was introduced. Solution spaces consist of the set of possible configurations of a constraint problem under consideration at a particular time. Solution spaces are equivalent to the search spaces used in finite domain constraint satisfaction [103, 114] and the configuration spaces used in spatial planning [71, 121].

Extensive study of the constraint solution algorithms of chapter 2 led to commonalities of their approaches. All of the constraint algorithms studied gradually refine an initial solution space, consisting of all the possible configurations in the constraint problem, using *solution steps*, until a terminal solution space is reached. The terminal solution space then consists of a number of configurations that may or may not be solutions. The sequence of solution steps transforming an initial solution space to a terminal solution space is a *solution process*.

The quality of the terminal solution space is critical for a constraint solver. It is not hard to identify constraint processes that will work efficiently but do not find solutions to the constraint problem. In such cases, the terminal solution space consists of configurations that are not solutions to the constraint problem. The most

important properties of solution spaces are to know whether they contain *a* solution; whether they contain *only* solutions; and whether they contain *all* of the solutions.

A *consistent* solution step ensures that the solution space contains *a* solution if one exists. A *sound* solution step ensures that the solution space contains *only* solutions. A *complete* solution step ensures that the solution step contains *all* solutions to the constraint problem.

However, a constraint process consists of a number of solution steps. It was by no means certain that because one solution step has a property then the solution process has that property. It was not even certain that if *all* solution steps in the process have a property then the solution process has that property. Consequently the properties of solution processes were examined.

Theorem 6.1 can be used to identify properties of constraint processes given properties of solution steps. The key results of theorem 6.1 are that

- Even if every solution step retains *a* local solution to the current set of constraints, the terminal solution space may not have a solution to the whole constraint problem, even though many may exist.

- If every solution step retains *only* solutions to the current set of constraints, then the terminal solution space will consist *only* of solutions to the whole constraint problem.

- If every solution step retains *all* of the solutions to the current set of constraints, then the terminal solution space will contain *all* of the solutions to the whole constraint problem.

Theorem 6.1 is a significant and powerful tool in describing the quality of results of a constraint process. However, note that theorem 6.1 cannot be used to describe constraint priority and backtracking solvers directly, as these solvers do not have to satisfy all of the constraints they impose.

The descriptive power of the framework is apparent as it can be used to describe a number of powerful enhancements of the basic constraint definition and satisfaction models. Using the framework presented in this chapter it is possible to describe backtracking, constraint priority problems, incremental constraint solution and variable-driven constraint solution. The framework also makes transparent the similarities and common elements between constraint solvers.

With the abstraction built up over the past four chapters, it is now possible to study in detail the use of domain specific knowledge in constraint solvers and also

the use of hybrid constraint solvers. In particular, theorem 6.1 developed in this chapter allows concrete statements to be made about the nature of hybrid constraint solvers that were not previously possible.

Chapter 7

Hybrid Collaboration in Constraint Solvers

As mentioned in chapter 1, one of the purposes of this thesis is to investigate the use of a hybrid of domain specific constraint solvers to solve complex engineering design constraint problems efficiently on a computer.

A *hybrid constraint solver* is a constraint solver that employs more than one method to solve a constraint problem. The strategy of a hybrid is usually to decompose a problem into a number of smaller subproblems and then apply the various solution methods to solve the subproblems. Using solutions to the subproblems, solutions to the original constraint problem can be found.

Hybrid constraint solvers are a particular example of the divide-and-conquer strategy frequently employed in Computer Science. There are many advantages to using hybrids, including speeding up solution, increasing the number of constraint problems that can be solved and making more flexible constraint solvers.

Non-hybrid constraint solvers include ICBSM [27], Gröbner basis and Newton-Raphson solvers. Hybrid constraint solvers include DCM [86], Connectivity Analysis [67], Erep [14], MechEdit [15], INCES [62] and IGCS [112].

DCM, Connectivity Analysis and Erep use very small, specialised solvers to handle the subproblems created. Most of the work done in these algorithms goes into the decomposition strategy that identifies the subproblems. On the other hand, MechEdit, INCES and IGCS use fairly simple decomposition techniques to identify large subproblems that are then handled by complex constraint solvers.

For example, DCM reduces a geometric constraint problem to a number of triangular subproblems consisting of lines and points fixed by three constraints. Very simple and fast routines are used to find the solutions to each triangular subproblem.

Most of the effort in DCM goes into identifying the subproblems in the first place.

The decomposition strategy used by INCES is to determine cyclic and acyclic subproblems of the constraint problem. Acyclic subproblems are solved using a local propagation constraint solver, whereas cyclic subproblems are solved using a specialised simultaneous equation solver, typically Newton-Raphson.

Although hybrids are extensively used in the literature, they are rarely identified as such. This chapter uses the framework built up over the previous four chapters to investigate hybrids and study the key aspects of hybrid constraint solvers.

Using the description of the constraint solution framework in chapter 3, hybrid constraint solvers work along the following lines:

1. Decompose a constraint problem into a number of subproblems.
2. Order the subproblems.
3. Solve the subproblems in order and recombine solutions to the subproblems into solutions to the original problem.

This chapter concentrates on the process of solving constraint problems using hybrid constraint solvers and then recombining solutions. Eric Monfroy has studied means of combining constraint solvers and he has created the BALI framework [84] for describing solver collaborations.

BALI has been adapted for use in this chapter as a means of formalising the way in which a hybrid constraint solver solves subproblems and recombines solutions to the subproblems. BALI is a particular means of describing this relation. It is primarily useful because it makes explicit the types of collaboration available and the way they interact.

It is assumed for the purposes of this chapter that decomposition and ordering strategies already exist. This chapter is primarily concerned with methods of solving subproblems and recombining solutions. Decomposition strategies and ordering strategies are covered in more detail in chapter 3.

Section 7.1 discusses using domain specific knowledge in constraint solution and uses the particular example of geometric reasoning to show why domain specific knowledge helps to create efficient constraint solvers.

Section 7.2 discusses hybrid constraint solvers. Hybrid constraint solvers are the combination of more than one constraint solver acting together. Hybrid solvers can potentially be used to solve problems that cannot typically be handled efficiently by other solvers. There is very little on hybrid solvers in the research literature.

Monfroy has developed a framework called BALI [84] which is discussed and adapted to produce a description of hybrid constraint solvers that decompose constraint problems.

Section 7.3 presents an example demonstrating the power of a hybrid constraint solver constructed from two domain specific solvers developed at Leeds. This simple study of an internal combustion engine demonstrates the feasibility of combining domain specific constraint solvers and gives some empirical evidence that the hybrid solver is, as hoped, very efficient.

Section 7.4 discusses the paradigms for collaboration introduced by Monfroy. These are serial, parallel and concurrent collaboration. The serial paradigm is described using the constraint satisfaction abstraction and the pros and cons of the paradigm are discussed in detail. Appendix E debates the parallel and concurrent collaboration approaches.

Section 7.5 introduces the solver collaboration language used by Monfroy in BALI. The collaboration language is extended so that decomposition strategies can be used by the hybrid and a new operation, the *conditional branch* is introduced so that different solver expressions can be used depending on the result of a test.

Section 7.6 presents an example demonstrating the asymptotic behaviour of the hybrid system developed in section 7.3. This involves using the serial collaboration paradigm to combine many instances of the two domain specific solvers together. When these failings are addressed and handled, empirical evidence obtained from the case studies indicate that the hybrid solver is very fast when compared to other solvers.

Section 7.7 draws conclusions from this chapter.

7.1 Using domain specific knowledge in constraint solvers

Using knowledge that is implicit in a problem is a common way of efficiently solving that problem. For example, suppose one wishes to find the word ‘toadstool’ in a dictionary. If one opens the dictionary at a page with the word ‘mushroom’ in it, then one can use one’s implicit knowledge of the alphabet and dictionaries to reason that ‘toadstool’ is probably after ‘mushroom’ in the dictionary. This is not usually actually stated but the implicit knowledge is used to direct the search.

Many constraint solvers use domain specific knowledge to help aid solution. For

example, FC uses domain specific knowledge of the nature of finite domains and that a failed partial instantiation implies a failed full instantiation. ICBSM [27] and Degrees of Freedom Analysis [58] use the implicit knowledge that rigid bodies can only translate and rotate in space. In this thesis, solvers that take advantage of implicit knowledge are called *domain specific*. Solvers that do not are called *(domain) general*.

A domain general solver may use, for example, numerical solution. General solvers are typically more *expressive* than domain specific solvers. Numerical solution of linear and nonlinear equations, though subject to convergence and numerical conditioning restrictions, is nevertheless capable of finding a solution to a much wider variety of problems than, say, a solver that uses implicit geometric knowledge on geometric constraint problems.

However, there are four chief disadvantages to numerical solution:

1. Convergence. Numerical solvers are not robust, in the sense that they can fail to converge to a solution when one exists. They also suffer from numerical conditioning problems [90] and the use of floating point arithmetic leads to numerical accuracy problems. These problems can be addressed at a considerable cost in computational effort.

2. Multiple roots. Numerical solvers typically only find one solution at a time to a problem. Users frequently wish to look through all of the solutions in order to identify the most desirable (Erep [13] is an excellent example of a solver that allows multiple roots and solutions). Numerical solvers *can* be adjusted to find more than one root but it would be very difficult to create an algorithm that could guarantee to find all roots due to convergence difficulties (see [90], pp 240-242), unless the constraint problem consists of polynomials.

3. Efficiency. Generally speaking, numerical solution is $O(n^2)$ complexity, where n is the number of constraints, whilst domain specific solvers are faster, frequently $O(n)$ complexity. For example, NAG's C05NBC function can be considered typical for our purposes and is $\Omega(n^2)$ [46]. The use of exact arithmetic to counter the inaccuracies resulting from floating point arithmetic will result in an increase in the complexity of the numerical solver.

4. Unintuitive. Numerical solvers may converge to a root but it may not be the expected solution to the problem.

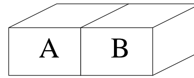


Figure 7.1: Two Blocks with an Against Constraint

The third criterion is of great interest if *efficient* general purpose solvers are to be constructed. In particular, this thesis concentrates on the solution of engineering constraint problems that are to be solved quickly and the results presented in real time. Because of this, domain specific solvers are very important. Domain general solvers are still important as a *fall-back* position.

Example 7.1 (Geometric constraint solvers) Solvers that take advantage of the nature of rigid bodies [1] and Euclidean space, such as [13, 27, 86, 112] are domain specific. Typically problems are described in terms of geometric entities and constraints.

For example, consider two blocks A and B with an *against* constraint so that the blocks remain in contact (figure 7.1). Blocks A and B are rigid bodies and as such have the following implicit knowledge associated with them:

1. The size of A or B cannot be altered,
2. A and B can translate in X, Y, Z directions,
3. A and B can rotate about the X, Y, Z axes,
4. A and B have no other allowable configurations.

It is the use of this implicit knowledge that marks out a domain specific solver. Degrees of freedom analysis [59], ICBSM [27] and IGCS [112] make use of this knowledge to convert a complex, nonlinear, continuous domain problem into a compact, discrete problem [59]. Erep [13] and D-Cubed [86], also take advantage of this information, though in a different way. □

7.1.1 Using domain specific knowledge is fast

Numerical solution to the problem in figure 7.1 is possible, as the constraint is simple to describe as an equation: the distance between A and B is 0 and they have the same orientation. However, the implicit geometric knowledge is lost in making this

[1] The restriction to rigid bodies is not always necessary (see Kramer's GCE for example [58]), but for the purposes of this example only rigid bodies are allowed, for simplicity.

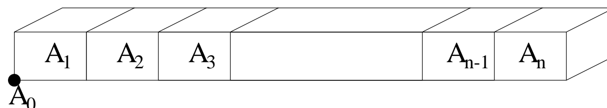


Figure 7.2: A Chain of Blocks with Against Constraints

step. Domain specific solvers can use the implicit geometric knowledge and are typically much faster and more robust, but are restricted to a smaller variety of problems than general solvers.

Example 7.2 (Solution of blocks) Consider a long chain of blocks joined by *against* constraints (figure 7.2).

Commonly used numerical techniques would solve this problem in $\Omega(n^2)$ time [46], where n is the number of constraints in the constraint problem. Kramer's action analysis [59] uses local propagation to find the positions of the blocks. Action analysis is a domain specific solver as it uses the geometric knowledge of the possible positions and orientations of the blocks in order to position them. Action analysis can find positions for the blocks in linear time. On top of this, adding a new constraint in degrees of freedom analysis can be an O(1) operation, provided certain conditions hold. The *incremental* addition of block A_{n+1} to the chain, for example, would involve only satisfying $against(A_n, A_{n+1})$ in O(1) time. □

7.1.2 Using domain specific knowledge is not enough

Domain specific constraint solvers are very important as they are fast. However, they are not usually expressive in the sense that they are restricted to a relatively small class of problems with specific structures. For example action analysis cannot solve the simple example below.

Example 7.3 (Coincident rods) Consider two rods grounded at points G_1 and G_2 respectively (figure 7.3). If coincident constraints are placed on the two opposite end-points, A and B, then there are two possible configurations of the rods that satisfy that constraint. However, knowing that either rod can rotate about their grounded end-point is not enough to allow calculation of those two positions. For example, line 1 can rotate through 2π radians and not have A coincident with B. The problem must be solved *simultaneously* in order to find the two positions. Line 1 is rotated a little bit and line 2 a little bit in order to solve the constraint.

In fact, this example shows a problem with most constraint solvers. Fast algorithms can be developed for simple cases, such as action analysis, local propagation

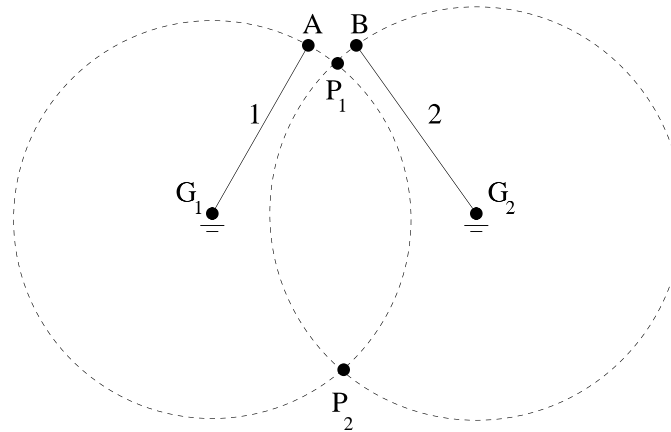


Figure 7.3: Two Rods

or triangular form. More complex problems must usually be solved simultaneously and this is much less efficient and, in general, harder. □

Kramer had to enhance action analysis with *locus analysis* in order to solve problems such as in the above example. Both action analysis and locus analysis are *domain specific*, but neither is effective on its own. Action analysis cannot solve problems simultaneously, whilst locus analysis must deal with the loci of objects which may be complicated and time-consuming.

7.2 Hybrid constraint solvers

A constraint solver that consists of two or more constituent solvers acting together is called a *hybrid* solver. This section studies the underlying structure of hybrid solvers. Using the framework of chapter 6, it is possible to describe hybrid solvers in a simple and elegant manner, and to draw conclusions about properties of hybrid solvers.

This section discusses hybrid solvers in terms of the framework and compares this approach with that of Monfroy. The example below introduces the concept of hybrid constraint solvers.

Example 7.4 (Geometric constraint problem) At the University of Leeds two constraint solvers have been developed independently. INCES [62] was developed by Lamounier *et al* and is capable of finding a solution to a system of equations, though it is most efficient at finding solutions to a triangular system of equations. IGCS [112] was developed by Tsai *et al* and is capable of finding all solutions to a geometric constraint problem, though it is best suited to problems without cycles.

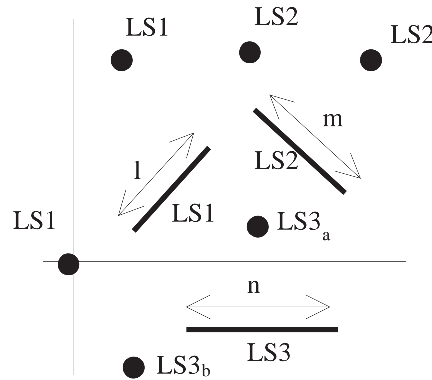


Figure 7.4: Problem G''

A hybrid of these two solvers would hopefully be able to solve any system of equations and any geometric problem that can be solved by INCES and IGCS individually. However, the hybrid should also be capable of solving a constraint problem that contains both variables and geometric entities, and both equations and geometric constraints.

Consider, for example, a modified version of problem G of example 4.8, as depicted in figure 7.4,

$$\begin{aligned} G'' \quad = \quad & (\{(LS1, \mathbb{R}^4), (LS2, \mathbb{R}^4), (LS3, \mathbb{R}^4), (LS1_a, \mathbb{R}^2), (LS1_b, \mathbb{R}^2), (LS2_a, \mathbb{R}^2), \\ & (LS2_b, \mathbb{R}^2), (LS3_a, \mathbb{R}^2), (LS3_b, \mathbb{R}^2), (l, \mathbb{R}), (m, \mathbb{R}), (n, \mathbb{R})\}, \\ & \{LS1_a = (0,0), LS1_b = LS2_a, LS2_b = LS3_a, LS3_b = LS1_a, \\ & d(LS1_a, LS1_b) = l, d(LS2_a, LS2_b) = m, d(LS3_a, LS3_b) = n, \\ & endpoint(LS1, LS1_a), endpoint(LS1, LS1_b), \\ & endpoint(LS2, LS2_a), endpoint(LS2, LS2_b), \\ & endpoint(LS3, LS3_a), endpoint(LS3, LS3_b)\}), \end{aligned}$$

and the simple system of equations,

$$E'' = (\{(l, \mathbb{R}), (m, \mathbb{R}), (n, \mathbb{R})\}, \{l^2 + n^2 = m^2, l + m = 8, 4m - 3n = 8\}).$$

In words, G'' consists of three line segments and six points in 2D space. The first constraint fixes a point at the origin. The next three constraints make three pairs of points coincident. The next three constraints set the lengths of the three line segments and the final six constraints associate points with the ends of the line segments. E'' uses a system of equations to fix the variables that describe the

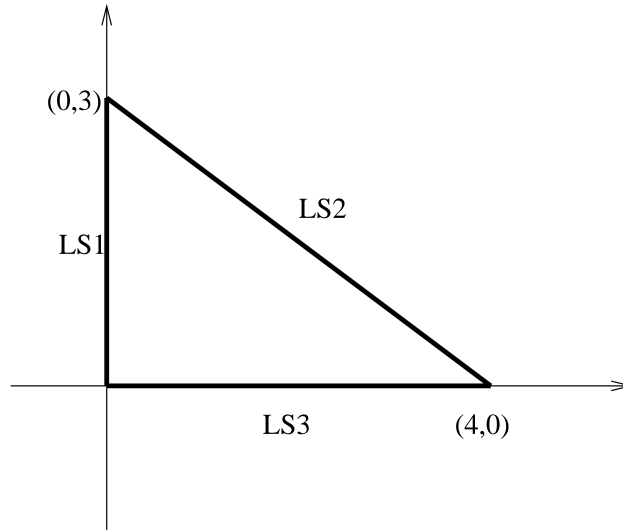


Figure 7.5: A solution to constraint problem G of example 4.8

lengths of the line segments in G''.

IGCS can find an underconstrained solution space for G'', but this solution space is not particularly useful as it is simply the set of all triangles with one vertex at the origin (see figure 7.5 for one such solution). INCES can find all of the solutions to E'', but on their own, these are not particularly useful either.

However, consider the combined problem,

$$
\begin{aligned}
H'' &= E'' \cup G'' \\
&= (\{(LS1, \mathbb{R}^4), (LS2, \mathbb{R}^4), (LS3, \mathbb{R}^4), (LS1_a, \mathbb{R}^2), (LS1_b, \mathbb{R}^2), (LS2_a, \mathbb{R}^2), \\
&\quad (LS2_b, \mathbb{R}^2), (LS3_a, \mathbb{R}^2), (LS3_b, \mathbb{R}^2), (l, \mathbb{R}), (m, \mathbb{R}), (n, \mathbb{R})\}, \\
&\quad \{LS1_a = (0,0), LS1_b = LS2_a, LS2_b = LS3_a, LS3_b = LS1_a, \\
&\quad d(LS1_a, LS1_b) = l, d(LS2_a, LS2_b) = m, d(LS3_a, LS3_b) = n, \\
&\quad endpoint(LS1, LS1_a), endpoint(LS1, LS1_b), \\
&\quad endpoint(LS2, LS2_a), endpoint(LS2, LS2_b), \\
&\quad endpoint(LS3, LS3_a), endpoint(LS3, LS3_b), \\
&\quad l^2 + n^2 = m^2, l + m = 8, 4m - 3n = 8\}).
\end{aligned}
$$

IGCS cannot find solutions to H'' as it cannot solve the set of equations which are from E''. INCES *can* solve H'', but only by converting the geometric constraints in G'' into equations, losing all domain-specific knowledge in the process. However, if INCES is used to find all solutions of E'', then the values of l, m, n found could be

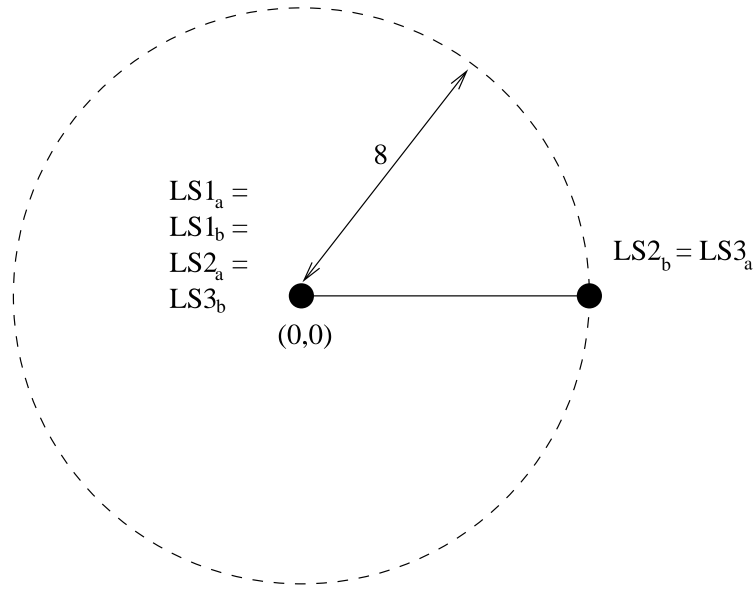


Figure 7.6: Solutions of constraint problem G'' with $l = 0, m = 8, n = 8$

used by IGCS to solve G''.

In this case, INCES would find the following solutions to E'':

$$\{l = \{3\}, m = \{5\}, n = \{4\}\} \text{ or } \{l = \{0\}, m = \{8\}, n = \{8\}\}.$$

Using either of these solutions IGCS can solve G'' when l, m and n are instantiated to the values in the solutions. Solving for the first solution gives the same solutions as problem G. Using the second solution results in a degenerate case for G'', with solutions as shown in figure 7.6. For this problem, the hybrid has found all solutions to the combined problem H'' efficiently, where the constituent solvers were either not expressive enough or would be much slower. □

In this example, the combined problem H'' was split into two subproblems G'' and E'' by manual inspection. In general, the decomposition of a constraint problem is a non-trivial task. However, decomposition of a problem to subproblems is one of the most important issues that a hybrid will face. This chapter does not discuss the decomposition issue, but assumes that decomposition of the constraint problem has already taken place. Chapter 3 deals with decomposition techniques in more detail.

In a similar vein, in general recombination of the solutions to subproblems is also a difficult task. In the above example, there were only two solutions to E''. Both solutions could be studied by using them as input to problem G'' and then solving G''. In general, there may be an infinite number of solutions to a problem or it may

not be possible to use solutions as input to another problem. The recombination problem depends on the hybrid collaboration used and this is discussed in more detail in section 7.4.

7.2.1 BALI

Monfroy has studied the problem of hybrid solvers for continuous domain constraint problems. This has resulted in an environment called BALI for describing solver collaborations [84]. Monfroy defines solver collaboration as either combination or cooperation. Solver combination focuses on a solver for the union of constraint problems, that is a solver combination of two solvers operates on mixtures of the constraint problems the solvers individually operate on. Solver cooperation concentrates on communications problems between solvers on a single domain but dealing with different sets of constraints.

Monfroy defines constraint systems as a *language* for describing constraint problems. A constraint system consists of a quadruple (Σ, D, V, ℓ), where

Σ	is the set of symbols in the language,
D	is the domain of the union of all the domains of the symbols in Σ,
V	is the set of variables in the language,
ℓ	is the set of constraints in the language, consisting of all possible quantifier free† , first order formulae† built over Σ and V.

A solver in Monfroy's notation is a function

$$S : \ell^n \longrightarrow \ell \text{ such that}$$
$$\forall c \in \ell^n, D \models S(c) \Leftrightarrow D \models c$$
$$\text{and } \forall c \in \ell^n, \exists n \in \mathbb{N}, S^{n+1}(c) = S^n(c).$$

The notation $D \models S(c)$ is used to represent $S(c)$ being a valid solution in D. Thus, a solver is a function which produces a valid solution, terminates and has a fixed point at $S^n(c)$.

The definition of a solver is interpreted thus:

> A solver S is a function that maps a conjunction of constraints to a single constraint such that, for all conjunctions of constraints, if $S(c)$ is a

> valid solution to the conjunction of constraints, then so is c. Also, there exists a number $n \in \mathbb{N}$ such that, if the solver S is applied $n+1$ times to c, then the result is the same as if S were applied n times. Consequently, no refinement is achieved by applying S more than n times and so S terminates when applied n times and $S^n(c)$ is a fixed point of S.

A constraint C in Monfroy's definition is reduced to a "simpler" constraint by the application of a constraint solver. However, Monfroy's constraints are the conjunction of a number of formulae. In the terms of this thesis, a single constraint of Monfroy is equivalent to the set of constraints Ψ in a constraint problem.

Under definition 6.6, a constraint solver does not transform a set of constraints to a simpler set but instead transforms a solution space to a simpler solution space. Since a conjunction of constraints is equivalent to a solution space, Monfroy's definition of a component solver is equivalent to the following

Definition 7.1 (Component solvers) *A component solver is a function S*

$$S : \boldsymbol{\mathcal{D}}(k-1) \longrightarrow^{*} \boldsymbol{\mathcal{D}}(k),$$

such that

$$\forall C \in \ell^n, \boldsymbol{\mathcal{D}}(k) \subseteq \boldsymbol{\mathcal{D}}(0) \Leftrightarrow \boldsymbol{\mathcal{D}}(k-1) \subseteq \boldsymbol{\mathcal{D}}(0).$$

□

The definition of a constraint process in chapter 6 does not enforce the termination or fixed point conditions of Monfroy's constraint solvers explicitly. However, since the set Ψ of constraints in the process is finite and since each Ψ_i is pairwise disjoint, the number of steps in a solution process is finite and the solver will always terminate if each step takes a finite amount of time.

Further application of solution processes to the terminal solution space $\boldsymbol{\mathcal{D}}(\kappa)$ will not refine $\boldsymbol{\mathcal{D}}(\kappa)$ any further. Consequently, $\boldsymbol{\mathcal{D}}(\kappa)$ represents the fixed point of the solution process. The definitions of constraint solvers are therefore equivalent.

Note however, that Monfroy does not discuss the structure of the solution space. The advantage of the solution framework developed in chapters 4, 5 and 6 is that the nature of the solution space can be investigated and conclusions drawn from it.

Monfroy also forces the output of a constraint solver to be "smaller", in the following sense (taken from [84]).

book

BookCode	Author	Title	Price
01	Dante	Inferno	20
27	Joyce	Ulysses	30
21	Tolstoy	War and Peace	27
54	Greene	The Third Man	15

sale

Salesman	BookCode	Quantity
Jones	21	80
Smith	54	50
Robinson	54	50
Smith	21	100

Table 7.1: Two database tables (from [5])

book $\bowtie$ sale

BookCode	Author	Title	Price	Salesman	Qty
21	Tolstoy	War and Peace	27	Jones	80
54	Greene	The Third Man	15	Smith	50
54	Greene	The Third Man	15	Robinson	50
21	Tolstoy	War and Peace	27	Smith	100

Table 7.2: The result of joining the *book* and *sales* tables (from [5])

Definition 7.2 (Component solver ordering) *Let S be a component solver on the constraint system $CS = (\Sigma, D, V, \ell)$. Then the relation $\leq_S$ is defined on CS as follows: $C_1 \leq_S C_2$ if $\exists n \in \mathbb{N}$ s.t. $C_1 = S^n(C_2)$.* $\Box$

Given the definition of component solvers above in the terms of this thesis, it is obvious that the relation $\subseteq$ used on solution spaces is a $\leq_S$ ordering.

Monfroy also identifies the need to enrich constraint problems so that solvers designed for simpler problems can be applied to problems designed for hybrid solvers without losing any solutions. This concept is significant in terms of the framework outlined in this thesis and so is adapted in the next section.

7.2.2 Enhanced solution spaces

In relational algebra, it is sometimes necessary to form queries on objects that are spread over two tables. In order to satisfy this query, the two tables are 'joined' together and the query acts on the merged table. For example, given the two tables in table 7.1, the join of the two tables is table 7.2.

Similarly, given two constituent constraint solvers α and β, solving constraint problems $P_1 = (\Phi_1, \Psi_1)$ and $P_2 = (\Phi_2, \Psi_2)$ respectively, the hybrid of α and β operates on $P_3 = P_1 \cup P_2 = (\Phi_1 \cup \Phi_2, \Psi_1 \cup \Psi_2)$. It is useful to be able to talk about the solution space of P_3.

If $\Phi_1 \cap \Phi_2$ is empty then P_1 and P_2 are unrelated and can be solved separately. In this case, there is no difficulty in using a hybrid constraint solver to deal with P_3. However, if $\Phi_1 \cap \Phi_2$ is non-empty , then P_1 and P_2 share common entities. In this case the solution space of P_3 is not simple.

For example, suppose that $P_1 = (\Phi_1 = \{(x, \{0,1\}), (y, \{1,2\})\}, \Psi_1), P_2 = (\Phi_2 = \{(y, \{1,2\}), (\{2,3\})\}, \Psi_2)$. Then the solution space of $P_1, \boldsymbol{\mathcal{D}}(P_1)$ is

$$\{0,1\} \times \{1,2\}$$

and the solution space of $P_2, \boldsymbol{\mathcal{D}}(P_2)$ is

$$\{1,2\} \times \{2,3\}.$$

Here, $\Phi_1 \cap \Phi_2 = \{(y, \{1,2\})\}$. The solution space of P_3 is therefore not $\boldsymbol{\mathcal{D}}(P_1) \times \boldsymbol{\mathcal{D}}(P_2)$ as might be expected. Instead it is $\{0,1\} \times \{1,2\} \times \{2,3\}$.

Given Φ_1 and Φ_2 the solution space of P_3 is called the *enhanced solution space of* Φ_1 *with respect to* Φ_2, denoted in this thesis as $\boldsymbol{\mathcal{D}}_{\Phi_1}|^{\Phi_1 \cup \Phi_2}(0)$. The enhanced solution space is defined in appendix D. Enhanced solution spaces and embedded solution spaces, which are also defined in appendix D, are useful as they allow discussion of the structure of solution spaces used in hybrid constraint solvers. Embedded solution spaces are denoted as $\boldsymbol{\mathcal{D}}_{\Phi_1 \cup \Phi_2}|_{\Phi_1}(0)$. For simplicity, however, the precise definition is left to the appendix.

7.3 A simple example hybrid constraint solver

A simple hybrid solver based on two existing solvers at the University of Leeds was constructed in order to study the interaction and communication necessary for the solvers to work in concert and also as an exemplar case study to show some of the benefits of a hybrid of domain specific solvers.

The two solvers combined were INCES [62] and IGCS [112]. INCES is a domain specific solver for linear, triangular equations between entities in the real domain. IGCS is a domain specific solver for simple geometric objects in two dimensions.

In his thesis [64], Lamounier described the implementation of an internal com-

bustion engine in INCES. The internal combustion engine contained some geometric constraints describing the geometric structure of the engine and some algebraic constraints describing the operation of the engine. However, the geometric part of the problem had to be converted into algebraic equations in order to be solved by INCES. Unfortunately, this meant that INCES did not take advantage of the domain specific knowledge incorporated into the geometric problem. This made both the interactive manipulation of the entities difficult and solution of the combined problem dependent on the speed of INCES at solving geometric constraints converted into quadratic equations. IGCS could have handled the geometric problem quickly and intuitively.

A simple solution to this problem presented itself. This was to construct a hybrid of IGCS and INCES which allowed direct manipulation of the geometric problem through IGCS and communicated values of entities in both problems between the two solvers.

In the following example, the notation (x, y, z) is used to describe the degrees of freedom available for a geometric entity. The x component refers to the number of rotational degrees of freedom, y to the number of scalar (dimensional) degrees of freedom and z to the number of translational degrees of freedom. This notation is used so that the geometric solver IGCS can take advantage of this information. Thus a point $crank_1$ is free to move about the plane $\mathbb{R}^2$ and has a domain of $\mathbb{R}^2$. However, the point has only two translation degrees of freedom, and no rotational or scalar degrees of freedom. Consequently, the domain of the point can be described as (0,0,2) representing 0 rotational degrees of freedom, 0 scalar degrees of freedom and 2 translational degrees of freedom. For simplicity, all of the geometric objects in the following case study are points. For this case study, let $P_{geom} = (\Phi_{geom}, \Psi_{geom})$ be the geometric constraint problem in figure 7.7. The set Φ_{geom} describes the entities in the problem, where:

$$\begin{aligned}\Phi_{geom} \quad = \quad & \{(crank_1, (0,0,2)), (crank_2, (0,0,2)), (connecting_rod_1, (0,0,2)), \\ & (connecting_rod_2, (0,0,2)), (piston_1, (0,0,2)), (piston_2, (0,0,2)), \\ & (frame_1, (0,0,2)), (frame_2, (0,0,2)), (a, \mathbb{R}), (r, \mathbb{R})\}.\end{aligned}$$

The set Ψ_{geom} describes the constraints in the problem, where:

$$\begin{aligned}\Psi_{geom} = \{ & crank_1 = (0,0), \\ & crank_2 = connecting_rod_1, \\ & connecting_rod_2 = piston_1, \\ & piston_2 = frame_1, \\ & distance(crank_1, crank_2) = a, \\ & distance(frame_1, frame_2) = 10 \\ & distance(connecting_rod_1, connecting_rod_2) = 10, \\ & distance(piston_1, piston_2) = r\}.\end{aligned}$$

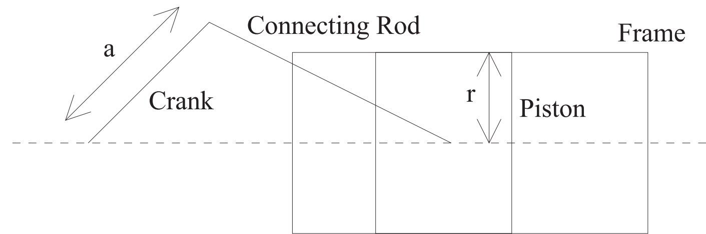


Figure 7.7: The Internal Combustion Engine

The piston and frame are assumed to be of constant shape and length for this case study and so can be represented by line segments.

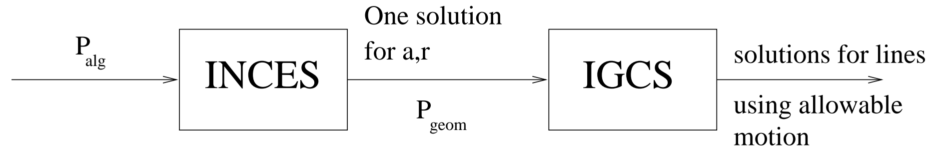


Figure 7.8: A Serial Hybrid of INCES and IGCS

Let $P_{alg} = (\Phi_{alg}, \Psi_{alg})$ be the associated algebraic constraint problem, where

$$\begin{aligned}
\Phi_{alg} &= \{(power, \mathbb{R}), (\alpha, \mathbb{R}), (displacement, \mathbb{R}), (c_ratio, \mathbb{R}), \\
&\quad (top_gap, \mathbb{R}), (a, \mathbb{R}), (\pi, \mathbb{R}), (r, \mathbb{R}), (n, \mathbb{R})\}, \\
\Psi_{alg} &= \{power = \alpha \times displacement \times c_ratio, \\
&\quad c_ratio = \frac{2a + top_gap}{top_gap}, \\
&\quad displacement = 2a\pi r^2 n, \\
&\quad power = 10, \\
&\quad top_gap = 10, \\
&\quad displacement = 10, \\
&\quad n = 10, \\
&\quad \alpha = 0.056, \\
&\quad \pi = 3.14159265\}.
\end{aligned}$$

In Φ_{alg}, α is a constant used in the computation of the engine power; *c_ratio* is the approximate ratio between the maximal and minimal pressures during the compression part of the cycle; *displacement* is the volume of mixed air and fuel consumed per engine cycle; a is the length of the rotary part of the crankshaft; *top_gap* is the length of the minimal distance between the top of the cylinder and the piston top during the cycle; n is the number of engine cylinders; and r is the radius of the engine's cylinders.

Communication between the two solvers depends on the common entities, $\Phi_{geom} \cap \Phi_{alg} = \{(a, \mathbb{R}), (r, \mathbb{R})\}$. Since P_{alg} has three equations in three unknowns, it is well-constrained and there are a finite number of solutions for a and r (in fact only one). Consequently, a hybrid was constructed that solved P_{alg} using INCES first and then used the values of a and r found to solve P_{geom} using IGCS (see figure 7.8).

Note that good use is made of the domain specific knowledge of the two constraint problems so that domain specific solvers are used to best effect. Note also that the fact that P_{alg} was well-constrained was used to decide the order in which P_{alg} and

Hybrid	INCES	NAG	Gröbner basis
0.0027s	0.013s	0.11s	2.3s

Table 7.3: Results for Solving ICE engine 50000 times on an SGI Indy

P_{geom} were solved.

In general determining whether a particular subproblem is well-constrained or not is difficult. In particular, determining the constrainedness of a subproblem will typically involve decomposing the subproblem to such an extent that it is already solved, defeating the purpose of determining the constrainedness. Latham and Middleditch's Connectivity Analysis [67] determines the constrainedness of subproblems by decomposing to residual sets. However, the residual sets are sufficiently small that they are simple to solve. Consequently, most of the work has gone into decomposing the constraint problem to determine the constrainedness. Using constrainedness as a guide to determining in which order subproblems should be solved is therefore counter-productive. This issue is dealt with in more detail in chapter 3.

The hybrid was constructed and solution timed on a Silicon Graphics Indy machine. The hybrid was compared with INCES, a numerical solver from NAG [46] and a Gröbner basis package within Maple [18]. See table 7.3 for the results of running the experiments. Although the hybrid is in fact an order of magnitude faster than the other constraint solvers and a thousand times as fast as the Gröbner basis, no firm conclusion should be drawn from this table, as timing is highly dependent on algorithm, compiler and system use at the time.

However, the numerical algorithm used, NAG's C05NBC, is probably one of the fastest available and it is relevant that an optimised numerical algorithm (a domain general constraint solver) takes a hundred times as long as the simple hybrid.

These results correspond to what one would expect. Gröbner basis algorithms are very slow, with an exponential complexity. The NAG algorithm is $\Omega(n^2)$. INCES uses the NAG algorithm on cyclic subproblems but a linear algorithm for acyclic subproblems. Consequently, the worst case complexity for INCES is $O(n^2)$, but the average case complexity is less than this. Similarly, the hybrid has worst case complexity $O(n^2)$ but has average case complexity less than INCES. This corresponds to table 7.3.

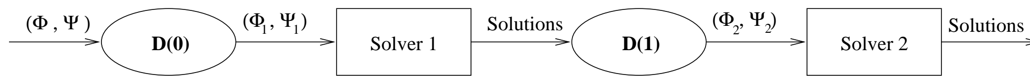


Figure 7.9: Sequential Collaboration

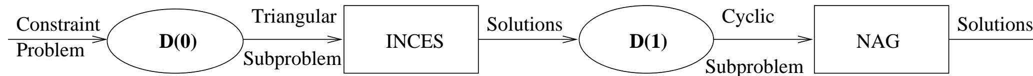


Figure 7.10: INCES as a sequential hybrid

7.4 Paradigms of collaboration

Monfroy [84] cites three paradigms of collaboration that can be used to construct hybrid solvers: sequential, parallel and concurrent. In fact, these paradigms follow naturally from the satisfaction framework presented in this thesis. In this section, the sequential paradigm is discussed in detail in terms of the constraint satisfaction framework. The advantages and disadvantages of the paradigm are discussed, particularly in terms of practical issues.

The parallel and concurrent collaborative paradigms are presented in appendix E.

7.4.1 Sequential hybrids

The most obvious collaboration paradigm is sequential. In this case, solvers operate on the solution space produced by the preceding solver. Since solution processes can be described as solution steps (lemma 6.1), sequential collaboration is the solution process formed by using constituent solvers as solution steps (see figure 7.9).

Theorem 6.1 applies directly to sequential hybrids. Consequently, even if all constituent solvers in a sequential hybrid are locally consistent, then the hybrid is not necessarily consistent. If all constituent solvers are complete (sound) then the hybrid is complete (sound).

This is particularly significant for most current sequential hybrids. For example, INCES [62] solves linear, triangular algebraic constraint problems. However, when INCES comes across equations that must be solved simultaneously, it resorts to numerical solution of the simultaneous equations. Thus, INCES is a sequential hybrid of the form in figure 7.10.

However, numerical solution is not globally consistent. Convergence problems mean that numerical solution is not robust and sometimes the solver will not find a solution even if one exists. Since the numerical subsolver used by INCES is not globally consistent, by theorem 6.1 the hybrid is not globally consistent. It is also

significant that numerical solution may converge to a solution but that this may not be the right solution for the rest of the problem.

Example 7.5 (Sequential hybrid inconsistency) For example, consider the sequential hybrid of figure 7.10, INCES. Let P be the constraint problem $P = (\Phi, \Psi)$, where Φ and Ψ are as defined below.

$$\begin{aligned}\Phi &= \{(a, \mathbb{R}), (b, \mathbb{R}), (c, \mathbb{R}), (d, \mathbb{R}), (e, \mathbb{R})\},\\ \Psi &= \{a^2 + b^2 = c, a^4 + b^4 = 32, c = d - 10, d = 18, b + e = 2, e = 4\}.\end{aligned}$$

INCES will solve P by first creating an Equation Graph describing the problem. Study of the Equation Graph results in INCES solving the triangular problem $P_1 = (\{(c, \mathbb{R}), (d, \mathbb{R})\}, \{c = d - 10, d = 18\})$ first to give solution $d = \{18\}, c = \{8\}$. INCES cannot solve the problem $P_2 = (\{(a, \mathbb{R}), (b, \mathbb{R})\}, \{a^2 + b^2 = 8, a^4 + b^4 = 32\})$ using local propagation and so resorts to numerical solution. There are actually four solutions to P_1,

$$\begin{gathered}\{a = \{2\}, b = \{2\}\}, \{a = \{-2\}, b = \{2\}\},\\ \{a = \{2\}, b = \{-2\}\}, \{a = \{-2\}, b = \{-2\}\}.\end{gathered}$$

However, numerical solution will only find one. The solution found depends on the precise numerical technique used and the initial starting point. Let us assume that the solution found is $\{a = \{2\}, b = \{2\}\}$. The remainder of P to be solved is now triangular and is equivalent to $P_3 = (\{(e, \mathbb{R})\}, \{2 + e = 2, e = 4\})$, which is inconsistent.

However, if the numerical solver had found solution $\{a = \{2\}, b = \{-2\}\}$ or solution $\{a = \{-2\}, b = \{-2\}\}$, then a consistent solution to P could have been found. This problem arises because the numerical solver is not globally consistent, nor is it locally complete. □

Similarly, IGCS [112] is not consistent. IGCS consists of a number of different algorithms for dealing with certain situations. For example, IGCS uses allowable motion to find solutions to simple problems by local propagation and locus analysis to solve simultaneous problems. Thus IGCS is a sequential hybrid of the form in figure 7.11.

However, the locus analysis used by IGCS is not locally complete. Consider example 7.3 (repeated here as figure 7.12 for convenience).

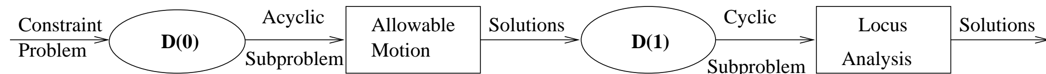


Figure 7.11: IGCS as a sequential hybrid

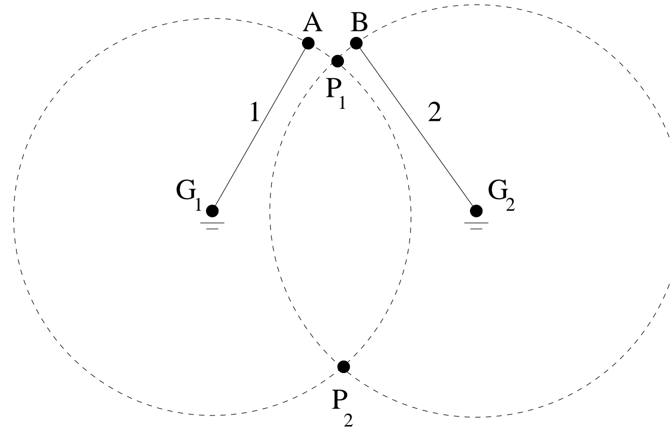


Figure 7.12: Two Rods

There are two solutions to this problem. However, the locus analysis in IGCS will only consider the solution which involves moving the rods the least, to P_1. Locus analysis is *locally* consistent and IGCS is not globally consistent. For example, if the only solution to the whole constraint problem involves the rods being coincident at P_2 then IGCS will not find it.

Previously INCES and IGCS were not known to be inconsistent.

7.4.1.1 Limitations of serial hybrids

Serial solution is difficult primarily because the output from one solver may not be valid input to the next solver. For example, if a finite domain solver produces solutions such that a line can take only lengths $\{1, 3, 5, 7\}$, then there are very few constraint solvers that allow this kind of line. It is, of course, possible to run the second solver four times, for each of the solutions, but this approach is clearly infeasible if there are a large number of solutions. Such an approach would also eliminate many of the advantages of using a serial hybrid.

In fact there is only a limited dependency between the two solvers. Any entity in the first problem not in the second problem does not need to be communicated to the second solver. Correspondingly, the second solver only needs to use as input solutions for the intersection of entities between the two problems. Even so, this may be many possible configurations.

Serial collaboration will work best when only one solution is found to a subproblem $P_1 = (\Phi_1, \Psi_1)$ and passed on to the next subproblem $P_2 = (\Phi_2, \Psi_2)$. In this case, serial collaboration is very simple. All entities in $\Phi_1 \cap \Phi_2$ are assigned a value from the solution to P_1. Thus the size of P_2 has been reduced by the number of entities in $\Phi_1 \cap \Phi_2$ which are now fixed. Hopefully some of the constraints in Ψ_2 will have been made trivial or easier by this reduction. Unfortunately, as is the case with INCES and IGCS, this serial collaboration is not globally consistent.

If a subproblem has only one solution, as is usually the case in linear programming, then serial collaboration is an obvious choice, as it will be globally consistent. The more solutions a subproblem can be expected to have, the less likely it is a solution will be chosen that is globally consistent. Therefore serial collaboration becomes less attractive.

In fact, many constraint solvers such as DCM [86], Erep [14] and MechEdit [15] use rules or heuristics to determine the likely intent of the user in constructing the constraint problem.

For example, Erep insists that when a user sets a distance between a point and a line, the point will have to be on the same side of the oriented line before and after constraint solution. Thus, a single solution is selected from two possible solutions. If such rules are applied throughout the constraint subproblem, a single solution is selected from amongst an exponential number. The key point here is that this single solution is the one *most likely* to satisfy the user's intent and so is a good candidate to be a globally consistent solution to the combined constraint problem.

If a constraint subproblem can be expected to have an infinite number of solutions then the use of serial collaboration becomes more problematic. Choosing just one solution from an infinite number is unlikely to provide a globally consistent solution. Consequently all of the solutions should be passed to the next solver in the collaboration. Whether this is possible or not depends on the nature of the solutions and the allowable domains of entities in the second subproblem. If the solutions provided by the first solver are continuous and simple then this transfer may be quite straightforward. For example, if $\Phi_1 \cap \Phi_2 = \{(x, \mathbb{R}), (y, \mathbb{R})\}$ and the solutions to P_1 are $\{(x = \{1\}, y = \mathbb{R})\}$, then P_2 can be adjusted accordingly.

However, if the solutions provided by the first solver are complex then it may not be possible to transfer the solutions. For example, if $\Phi_1 \cap \Phi_2 = \{(x, \mathbb{R}), (y, \mathbb{R})\}$ and the solutions to P_1 are $\{(x = \{a\}, y = \{b\}) | a^3 + b^5 = 16\}$, then it is not simple to alter P_2 to take into account this information, other than adding it as another constraint, at which point a parallel collaboration becomes more appropriate.

In conclusion, serial collaboration should be considered when:

1. Global consistency is not important.
2. Rules or heuristics can be used to bias solutions towards globally consistent solutions.
3. Only one or a small finite number of solutions are expected from each subproblem.
4. An infinite number of solutions are expected but these will be continuous and simple.

If serial collaboration is not appropriate then a parallel collaboration or concurrent collaboration may be helpful. In some cases, however, no collaboration may be appropriate and the problem must be solved as a whole using a general solver such as Newton-Raphson or Gröbner bases.

7.5 Solver collaboration language

Monfroy uses a solver collaboration language to build complex solvers using the collaborative primitives introduced in the previous section and appendix E. However, Monfroy's collaboration language is predicated only by the solvers available and not by the constraint problem being solved. In order to take advantage of domain specific knowledge implicit in a constraint problem, the problem must be solved using a constraint solver capable of handling that domain specific knowledge. Since domain specific solvers can also sometimes fail to find solutions, it is also important to provide backup solvers should a first attempt fail. Domain specific solvers can fail to find solutions when they are asked to solve problems that are not within their speciality. For example, local propagation algorithms cannot solve cyclic problems and so ICBSM [27] cannot find solutions to the 4-bar linkage problem (figure 7.5).

Unfortunately, the only conditional statement in Monfroy's collaboration language is the conditional guard which calls another constraint solver when a previous expression of constraint solvers terminates with a particular solver. Monfroy gives no example of the conditional guard in use and its purpose is difficult to fathom.

Decomposition strategies as discussed in chapter 3 assign a set of subsolvers to a subproblem. The hybrid constraint solver then applies the subsolvers to the subproblems using the sequential, parallel or concurrent collaboration.

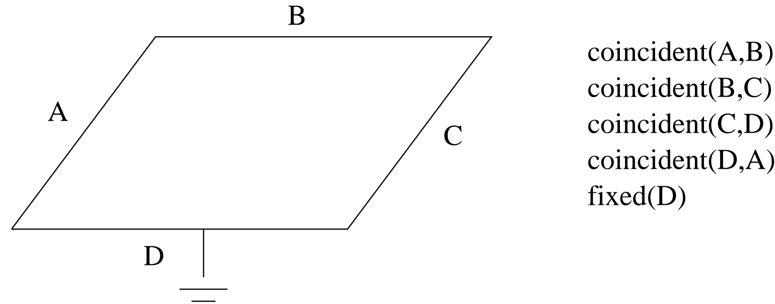


Figure 7.13: The 4 bar linkage problem

$$
\begin{array}{rcl}
Id & \in & \text{Identifiers}, \\
S & \subseteq & \text{Solvers}, \\
\psi & \in & \text{Conditional selection}, \\
P & \in & \text{Constraint problems}, \\
\mathcal{D} & \in & \text{Solution spaces}, \\
Col & ::= & Id = E, \\
E & ::= & \diamond | Id | B | E_1 ; E_2 | EP | \text{ repeat } (E) | \psi(EC) | \text{ if } T \text{ then } E_1 \text{ else } E_2, \\
T & ::= & E = \mathcal{D}, \\
B & ::= & (P,S) | (P,S) \parallel B, \\
EP & ::= & E | E \parallel EP, \\
EC & ::= & E | E ? EC.
\end{array}
$$

Table 7.4: Solver collaboration language (adapted from BALI [84])

Monfroy's solver collaboration language is an important means of designing and building hybrid constraint solvers that use the hybrid collaboration paradigms. However, Monfroy's language only uses solvers - it does not allow for subproblems to be assigned to specific sub-solvers.

Monfroy's collaboration language has been extended so that subsolvers operate on specific subproblems rather than the whole problem. The extended solver collaboration language used in the remainder of this thesis is explicated in table 7.4. An instance of this language is a solver collaboration. The addition of subsolvers to act on subproblems is witnessed by the inclusion of constraint problems P in table 7.4 and the definition of basic solvers B which is enhanced to include constraint problem-solver pairs.

The language given in table 7.4 also includes a *conditional branch* statement that can be used to drive backup constraint solvers. The branch operator is described in more detail in appendix F. For ease of comprehension, Monfroy's conditional guard has been dropped from the language in table 7.4.

A more detailed description of the collaboration language is given in appendix F.

The extended language in table 7.4 will be used to describe example constraint solvers in the remainder of this thesis.

7.6 An example of many solvers in serial

This section describes an experiment carried out to study the asymptotic behaviour of various constraint satisfaction algorithms on a simple case study. The algorithms studied were INCES [62], a numerical algorithm [46] and a sequential hybrid. The purpose of this experiment was to give empirical as well as theoretical evidence that the hybrid algorithm was sound, complete and more efficient than the other two algorithms as well as to investigate sequential collaboration. It was anticipated that the hybrid would be approximately linear in complexity, whilst the other two algorithms would be quadratic. This would help to underline the advantages of using hybrid algorithms.

This section presents the case study used, discusses the algorithms used to solve the case study and gives results. Appendix G presents a more detailed examination of the case study.

7.6.1 Case study

The case study chosen was an extension of Lamounier's internal combustion engine case study (see [64] and section 7.3). That problem studied the integration of some algebraic equations with the geometric constraints describing the construction of the piston. The two problems were linked so that the size of the piston and the length of the crankshaft were variables both in the functional problem and also in the geometric problem. However, this is a fixed size of problem. In order to study the asymptotic behaviour of the algorithms, n piston problems were joined together, as in figure 7.6.1.

In this case study, the n pistons are linked by coincident constraints at each end of the piston. Thus the problems are all connected and the complexity of the problem does increase as the size of n increases. The functional problems are not linked and are effectively lots of small, fixed problems solved independently.

7.6.2 The solvers used

Three solvers were examined. The NAG C05NBC function [46] was used as a numerical solver. It was passed the whole set of constraints and used numerical techniques

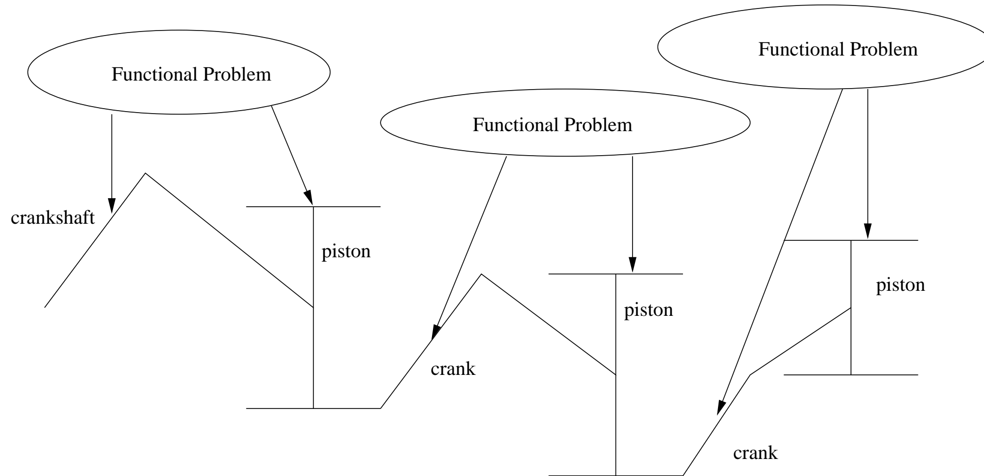


Figure 7.14: Case Study of n Piston Problems Linked Together

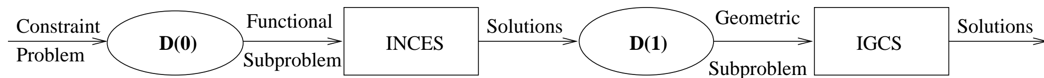


Figure 7.15: Serial Hybrid used to Solve n Piston Problems Linked Together

to converge towards a solution. The speed of convergence depended heavily on the initial guess, but the best case complexity of the NAG function is $\Omega(n^2)$, where n is the number of constraints.

Lamounier's INCES solver [62] is also capable of solving the problem as a whole. However, INCES deals only with equations and not geometric constraints. Geometric constraints can be handled if they are reduced to the constituent equations. INCES was expected to be quadratic, as it dealt with the problem as a whole and resorted to numerical solvers if loops appeared.

These constraint solvers were compared with a hybrid formed from combining the functional solver INCES and the geometric solver IGCS, much as in case study 1 (section 7.3). Each functional problem was solved using INCES and the results were passed to IGCS by varying the size of the lines in IGCS (see figure 7.15). It was hoped that the hybrid would be able to take best advantage of the domain-specific knowledge incorporated in the INCES and IGCS solvers and would be linear.

The decomposition strategy, De, used in this case is to decompose problem $P = (\Phi, \Psi)$ into a set $\{(S_i, P_i)\}$, where solver S_i is IGCS if subproblem P_i is geometric and S_i is INCES if subproblem P_i is algebraic. Decomposition is performed by first identifying constraints as geometric or algebraic. These form two sets of constraints Ψ'_1 and Ψ'_2. Constructing Ψ'_1 and Ψ'_2 takes time $O(n)$, where n is the number of constraints.

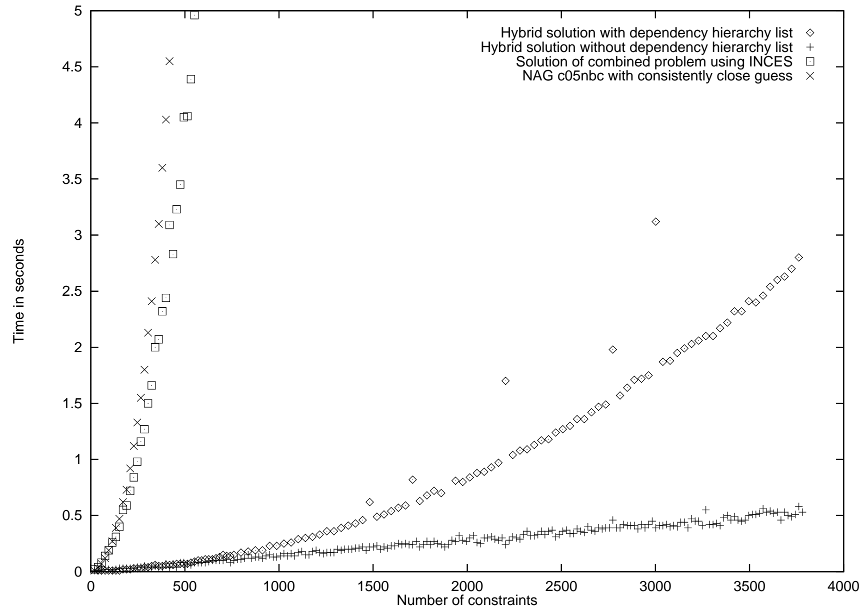


Figure 7.16: A comparison of the C05NBC function and INCES algorithm with the hybrid solver

Set Ψ'_i is then decomposed further into sets Ψ''_j of connected components, where $\Psi_1 \in \Psi'_i$ and $\Psi_2 \in \Psi'_i$ are connected if there is a path between Ψ_1 and Ψ_2 in the constraint/entity graph of constraint problem $(\Phi, \Psi \setminus \Psi'_j), j \neq i$. Finding the connected components can be done in a simple graph traversal algorithm that takes time $\mathrm{O}(m)$, where m is the number of edges in the constraint/entity graph. Since the imposed sets of constraints are usually quite small, m will typically be a multiple of the number of constraints in P. Thus decomposition of P takes time $\mathrm{O}(n)$.

With this decomposition strategy, the hybrid can be described in the solver collaboration language of section 7.5 as

$$((S_1, P_1); (S_2, P_2); (S_3, P_3); \ldots ; (S_n, P_n))\,.$$

7.6.3 Results

The case study was run for problem sizes between 1 and 200. This gave problems with between 19 and 3800 variables. All case studies were run on a Silicon Graphics Indy with an R4600 100MHz IP22 processor and 32 Mbytes of memory. The results of the case study are presented in figure 7.16. In the graph, the x-axis is the number of variables in the problem and the y-axis is the amount of time taken to solve the problem in seconds.

From the graph, it is apparent that the hybrid constraint solver is very fast

indeed. It is linear, whereas the other solvers compared were quadratic at best. It is three or four orders of magnitude faster than the NAG function. Even for problems of 100 or so variables, the hybrid is much faster.

7.7 Conclusions

Hybrid constraint solvers are an important means of solving constraint problems. Many current constraint solvers use hybrid techniques to solve constraint problems. Constraint solvers such as DCM [86], Erep [14] and Connectivity Analysis [67] decompose a constraint problem into a large number of very small subproblems that are solved using small, specialised domain specific solvers. Solvers such as MechEdit [15], INCES [62] and IGCS [112] decompose a problem to a small number of large subproblems that are solved using more complex solvers.

In general, constraint solvers work by decomposing a constraint problem into a number of subproblems; ordering the subproblems; solving the subproblems and recombining the solutions of the subproblems into solutions to the original problem.

This chapter has investigated the last stage of this process - solving and recombining. In particular, the chapter has analysed the use of *domain specific knowledge* and *hybrid constraint solvers* in solving complex constraint problems. Domain specific knowledge is added knowledge that is implicit in a problem. Many constraint solvers take advantage of domain specific knowledge to guide the satisfaction process towards solutions. For example, Degrees of Freedom Analysis [27, 58] takes advantage of the nature of rigid bodies in Euclidean space and the fact that such bodies can only rotate and translate in a very limited number of ways. Ruler-and-compass construction [14, 86] takes advantage of the fact that angle and distance constraints can be resolved using only simple construction steps. Finite domain solution techniques take advantage of the finite nature of the solution space to exhaustively explore it.

Domain specific knowledge allows the construction of fast, efficient solvers, such as those described above. However, domain specific knowledge is very limited in the problems that it can solve. Precisely because domain specific solvers take advantage of the structure of a problem domain, problems outside that domain cannot be solved easily.

Hybrid constraint solvers use two or more constraint solvers in *collaboration* to solve problems that the individual solvers could not solve on their own. The most prevalent form of hybrid solver is one that uses domain specific knowledge to solve as much of a constraint problem as possible and then uses a general solver, such as a

numerical solver, to solve the remainder of the problem. Solvers such as COSAC [85], INCES [62] and MechEdit [15] adopt this approach. The advantage of this approach is that the domain specific solver helps to make the hybrid solver faster than the general solver would have been on its own.

Monfroy's BALI environment [84] is an abstract framework for describing collaborations of multiple constraint solvers. The BALI framework provides a powerful means of describing complex hybrid constraint solvers. BALI tries to solve the entire problem using a single constraint solver. When it can proceed no further, it switches to another solver according to the semantics of the hybrid. Thus, BALI does not take into account the *structure* of the constraint problem and does not use any domain specific knowledge implicit in the constraint problem.

Consequently, this chapter has used the definition, representation and satisfaction framework developed over the previous three chapters to describe a more satisfactory collaborative framework for constraint solvers. The advantages of this framework are that it has all the descriptive power of BALI, but allows the constraint problem to be decomposed using a decomposition strategy and then solved using a hybrid.

In order to discuss the hybrid framework developed, the notions of *enhanced* and *embedded solution spaces* were introduced. Enhanced solution spaces allow the discussion of constraint processes in relation to a wider problem. Embedded solution spaces allow the discussion of constraint processes in relation to smaller subproblems. Both definitions are evident in BALI and have been adapted to the framework developed in this thesis.

The *sequential, parallel* and *concurrent* collaborative paradigms identified by Monfroy were then described using the framework of this thesis. Using the three collaborative paradigms it is possible to construct complex and elegant hybrid constraint solvers. Several case studies of sequential collaboration were presented in the chapter and appendix E contains an example of parallel collaboration.

It is then possible to use theorems 6.1 and E.1 to investigate the properties of hybrid constraint solvers. Theorem 6.1 can be used to study sequential collaboration and is used to identify problems with INCES [62] and IGCS [112]. INCES turns out to be not consistent, as the numerical subsolver used to handle cyclic sets of constraints only finds one solution that may not be the solution necessary for a solution to the whole constraint problem. Similarly, IGCS is not consistent, as the locus analysis of IGCS only takes into account one possible solution, when there may be many.

Serial collaboration is simple to implement and understand. It is also powerful as it can use knowledge from subproblems solved early to simplify subproblems solved later. However, serial collaboration is dependent on passing very few solutions from one subproblem to the next and this means that it is much more likely that any serial hybrid will suffer from global inconsistency.

Parallel and concurrent collaborations are discussed in appendix E.

Monfroy uses a solver collaboration language to describe the collaboration of solvers using the various paradigms of collaboration. However, Monfroy's language does not allow solvers to work on subproblems, nor does it contain an operator for choosing new solver expressions depending on the result of an expression. Consequently, BALI has been extended to allow solver expressions to act on subproblems and a conditional branch statement has been added to increase the power of the language.

The main contributions of this chapter are: a deeper understanding of the nature of hybrid collaborations; identification of the importance of domain specific knowledge; and the extension of BALI to conditional solution and subproblems.

The mathematical framework built up in chapters 4, 5 and 6 provide insight into the solver collaborations suggested by Monfroy. In particular, theorems 6.1 and E.1 identify the importance of global consistency in serial and parallel collaboration. The inability to guarantee global consistency for serial collaboration is a significant point as it means that most current constraint solvers that use hybrid techniques, such as INCES, IGCS, Erep and DCM are not globally consistent. Similarly, parallel hybrids are typically not globally consistent. The choice of when to use serial, parallel, concurrent or no collaboration is dependent on the type and number of solutions expected.

Domain specific knowledge is very important in constraint solution as it is intuitive, fast and robust within the class of problems it can handle. Most constraint solvers use domain specific knowledge as much as possible. In particular, solvers that decompose to small subproblems, such as Connectivity Analysis, DCM and Erep, advocate the use of domain specific solvers to handle the subproblems.

The addition of subproblems to Monfroy's BALI allows the description of the hybrid collaborations presented in this chapter to be formalised and made robust. This means that any hybrids constructed from the techniques in this chapter should be well-defined. The conditional branch statement allows for complex hybrids to be built that involve backup solvers when things go wrong.

Chapter 3 asked a number of questions about the process of solving constraint

problems by decomposition. Some of these questions can now be answered.

In fact, it is possible to lose solutions by decomposing and recombining. Theorems 6.1 and E.1 both conclude that solving subproblems individually will usually lead to a hybrid constraint solver that is not consistent and may lose solutions or may fail to find solutions when many exist. However, theorems 6.1 and E.1 also suggest that constraint solvers that retain all solutions (i.e. are locally complete) *can* be safely linked together using hybrid collaborations. However, few current constraint solvers are locally complete. Gröbner basis solvers are the most appropriate as they are both sound and complete. However, they are too slow for interactive use.

The question as to whether it is more efficient to decompose and recombine or to solve as a whole is more difficult to answer. To solve a constraint problem as a whole requires a general solver, such as Gröbner bases or Newton-Raphson. Since numerical solvers such as NAG's C05NBC function are $\Omega(n^2)$ complexity, where n is the number of constraints, decomposing and recombining must be at most $O(n^2)$ in order to compete.

In fact, nearly all current constraint solvers, such as DCM, Erep, IGCS, INCES and MechEdit are $O(n^2)$ complexity and average-case complexity $\Theta(n)$. Incremental constraint solvers such as IGCS, INCES and SkyBlue have worst case complexity $O(n)$ but average case complexity $O(1)$ to add a new constraint. Empirical evidence therefore suggests that decomposing and recombining is more efficient than solving as a whole.

The use of domain specific knowledge has been identified in this chapter as the key element in making the solution of subproblems fast. The combination of domain specific solvers using a well-defined hybrid collaboration leads to hybrid solvers that are faster than solving as a whole.

The fourth questions that was asked was whether it was possible to have a fast decomposition strategy and fast solution of subproblems or must one always dominate. Chapter 8 addresses this question by building a new constraint solver that uses coarse-grain decomposition to identify subproblems and then solves the subproblems using complex constraint solvers.

Chapter 8

New Directions

With the exception of INCES, the solvers presented in the examples in this thesis have mostly concentrated on complex decomposition strategies. The subsolvers they have used have been small and simple. For example, the constraint subsolvers used in DCM are the special case solvers for dealing with each set of three quadratic equations in three unknowns. The constraint subsolvers used in Connectivity Analysis find solutions to each residual set. Although Middleditch and Latham do not explore the issue in much detail, it seems likely that residual sets will be fairly small and amenable to simple solvers.

One of the questions asked in chapter 3 was whether it was better to have complex decomposition and simple solution or simple decomposition and complex solution. Most existing constraint solvers adopt the former approach and put most of their effort into the decomposition strategy. However, this thesis is primarily concerned with improving the performance of existing solvers, such as IGCS and ICBSM.

Of particular interest, is the possibility of improving the performance of IGCS so that it can handle loops and well-constrained problems better. Study of existing constraint solvers, such as Erep, suggest that the IGCS algorithms, such as local propagation and locus analysis are not as well-suited for solving loops and well-constrained problems as Erep is. Chapter 7 suggests that the domain specific knowledge used by IGCS and Erep limit their use to their own particular specialities. However, chapter 7 also suggests that a hybrid of IGCS and Erep could potentially handle constraint problems that are a mix of problems that IGCS and Erep can individually solve.

In general, it is difficult for a decomposition strategy to identify domain specific subproblems as the strategy will need to employ domain specific knowledge to find the subproblems. Using such domain specific knowledge will usually necessitate

the same complex decomposition that the domain specific solvers will themselves undertake and this will duplicate the effort involved. However, if knowledge about the combined problem can be exploited, it may be possible to perform a simple decomposition to identify the domain specific subproblems.

At this point, it is worth recalling that INCES adopts such an approach. The decomposition strategy used in INCES is very simple: a constraint problem P is split into subproblems whose constraint graph is a tree and subproblems that form strongly connected components. Tree subproblems are solved using local propagation and strongly connected subproblems are solved using Newton-Raphson. Note that most of the effort in finding solutions to P is taken up by the constraint solvers, particularly the Newton-Raphson solver.

This chapter explores the possibility of constructing a new hybrid constraint solver composed of two existing constraint solvers, Erep [14] and IGCS [112]. The hybrid will use coarse-grain decomposition to identify the domain specific subproblems suitable for Erep and IGCS. The natures of Erep and IGCS have been discussed extensively elsewhere in this thesis (see sections 2.3.3.2 and 3.1.3 for example) and so are not discussed in detail here. This section notes the reasons and processes behind the design of such a solver in order to characterise the important steps in designing a new hybrid solver. Note that the proposed solver has not been implemented.

The proposed solver will be a hybrid of Erep and IGCS. The solver must define:

1. a decomposition strategy,
2. an ordering strategy and
3. a solution strategy.

8.1 Decomposition strategy

The decomposition strategy for the Erep/IGCS solver is the key to its success. The strategy must be simple enough that it is fast to implement and yet sufficiently useful that it can identify subproblems suitable to be solved by either Erep or IGCS. In order to identify subproblems that are best solved by either Erep or IGCS, it is first necessary to identify the strengths of each solver, in particular the domain specific knowledge used by each solver.

Erep uses a ruler-and-compass approach to solve constraint problems consisting of distance and angle constraints between points and lines. Typically ruler-and-

Erep	IGCS
non-zero point-point distance	point-point coincidence (0 point-point distance)
point-line distance	point-line coincidence (0 point-line distance)
line-line angle	line-line angle
	circle-circle tangent
	circle-line tangent

Table 8.1: Constraints that can be handled by Erep and IGCS

compass constructs manifest themselves as *clusters* (see [14] and section 2.3.3.2) in a constraint graph.

IGCS uses rigid body allowable motion to solve constraint problems consisting of coincidence, tangent and angle constraints between points, circles and lines. Typically, such problems manifest themselves as trees or loops in constraint graphs.

In particular, it is possible to describe the constraints that each solver can handle explicitly. It is possible to decompose circle-circle tangent and circle-line tangent constraints to distance constraints that Erep can handle. However, doing so loses domain specific knowledge that IGCS exploits. Consequently, tangent constraints will remain as high-level constructs. The constraints each solver can handle are described in table 8.1.

Note that these constraints can be divided into three groups, α, β and γ. Group α consists of non-zero point-point distance and non-zero point-line distance constraints. Group β consists of point-point coincidence, circle-circle tangent and circle-line tangent constraints. Group γ consists of point-line coincidence and line-line angle constraints. Constraints in group α can only be solved using Erep. Constraints in group β can only be solved by IGCS. Constraints in group γ can be solved using either Erep or IGCS. In this way the strengths of Erep and IGCS are identified and correspondingly the subproblems that can best be handled by Erep or IGCS.

Each constraint in a constraint problem can then be associated with a group and therefore with the constraint solver(s) that can handle it. For example, figure 8.2 shows the Constraint/Entity graph for the constraint problem Q in figure 8.1 (from [37]). Each constraint has a group identifier associated with it to show which constraint solver can handle it. Note that all of the constraints in figure 8.2 can be solved by Erep.

The constraint problem R in figure 8.3 has the Constraint/Entity graph in figure 8.4. This problem consists of a number of constraints that can be solved in Erep and a number that can only be solved in IGCS.

The decomposition strategy should place constraints of type α in subproblems

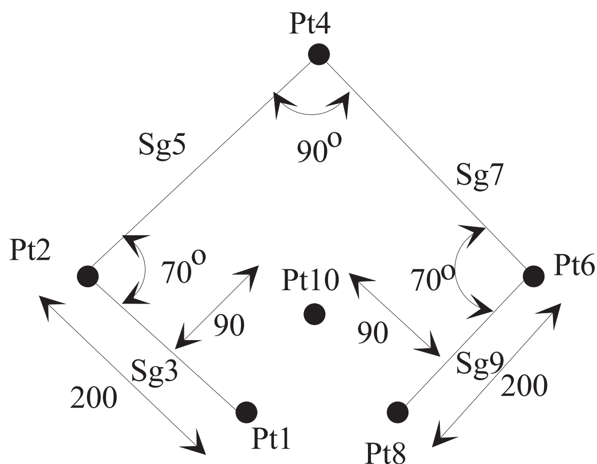


Figure 8.1: Constraint problem Q using distance and angle constraints

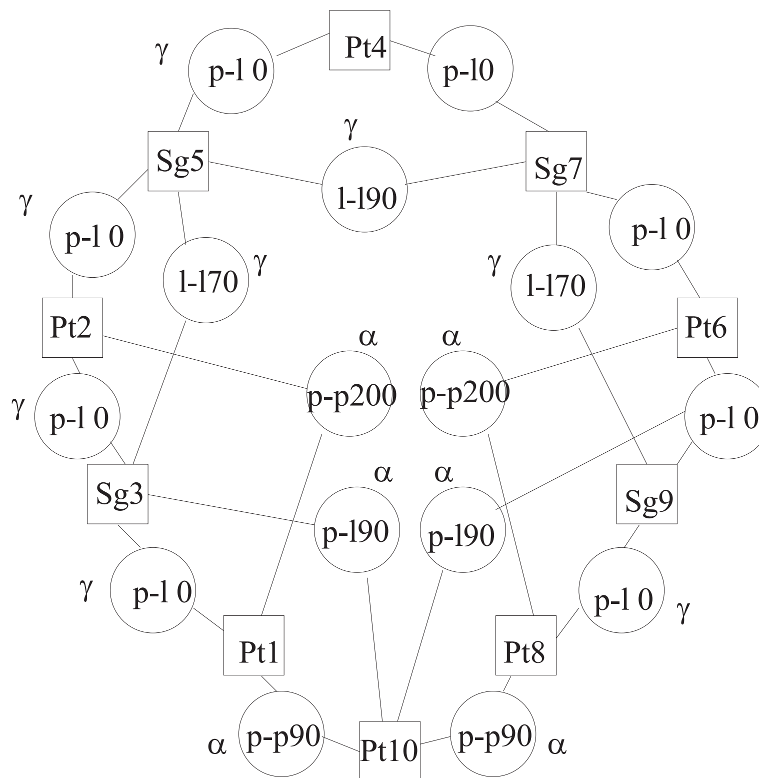


Figure 8.2: Constraint/Entity graph for constraint problem Q

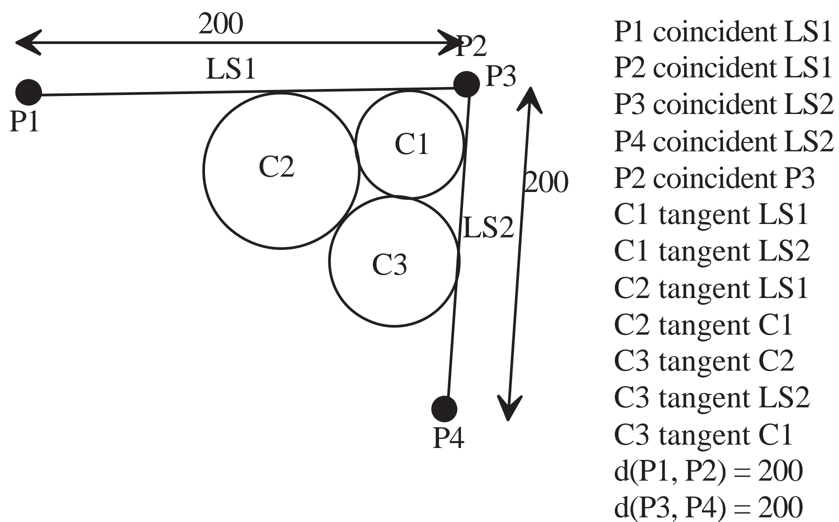


Figure 8.3: Constraint problem R with three tangent circles

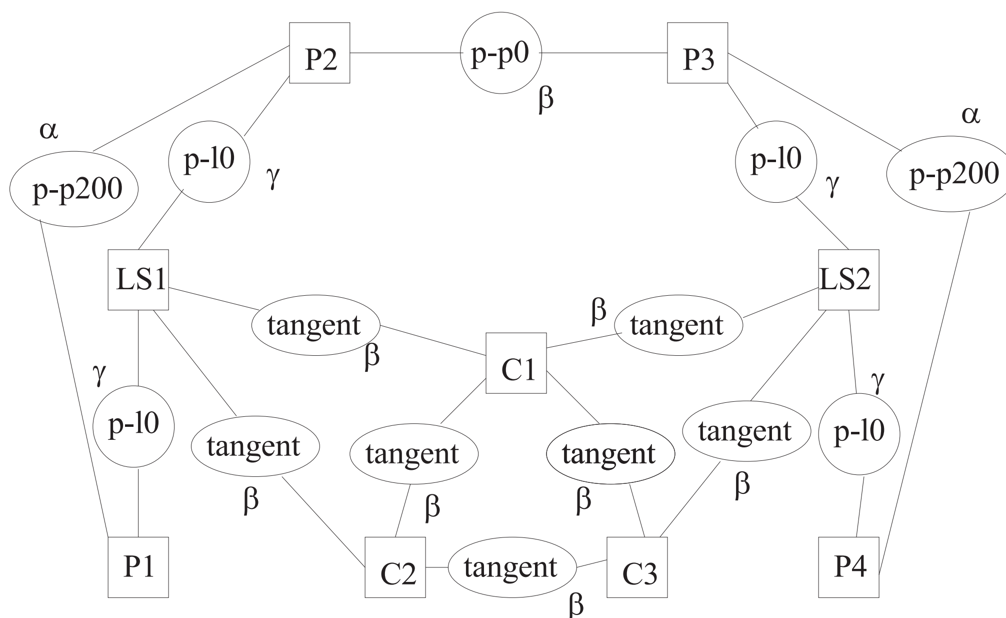


Figure 8.4: Constraint/Entity graph for constraint problem R

that will be solved by Erep and constraints of type β in subproblems that will be solved in IGCS. It remains to decide what to do with constraints of type γ and how to form the subproblems.

Erep solves constraint problems by identifying "clusters" of entities. Consequently, all type α constraints should be part of a cluster, so that Erep can find solutions for them. In theory, clusters could be determined using the same cluster finding techniques adopted by Erep itself, using type γ constraints when necessary. For example, figure 8.2 can be divided into two clusters with type α and γ constraints and one with only type γ constraints. Figure 8.4 has two clusters with type α and γ constraints. However, using the cluster finding techniques of Erep to find the subproblems to pass to Erep is a duplication of effort as the clusters will be found twice: once in the decomposition strategy and once in the solution strategy.

It is difficult to identify any simple decomposition strategy that will identify clusters but not need to perform the cluster finding algorithm using in Erep. This is a general point: the only way to *identify* domain specific subproblems may be to decompose the problems to such a degree that the decomposition strategy has in effect solved the subproblems and the solvers have no work to do.

Consequently it is proposed that the Erep/IGCS hybrid uses a slightly less discriminatory decomposition strategy:

Algorithm 8.1 Decomposition strategy for Erep/IGCS hybrid

1. Remove all type β constraints.
2. Place the remaining connected components into subproblems $\{P_i\}$.
3. Restore all type β constraints.
4. Remove all type α constraints.
5. Place the remaining connected components into subproblems $\{P'_j\}$.
6. Restore all type α constraints.

□

At this point each P_i contains subproblems consisting only of type α and type γ constraints. Each P'_j contains subproblems consisting only of type β and type γ constraints. Inevitably, there will be some type γ constraints that will be in both a P_i and a P'_j. However, each type γ constraint will be in at most one P_i and at most one P'_j.

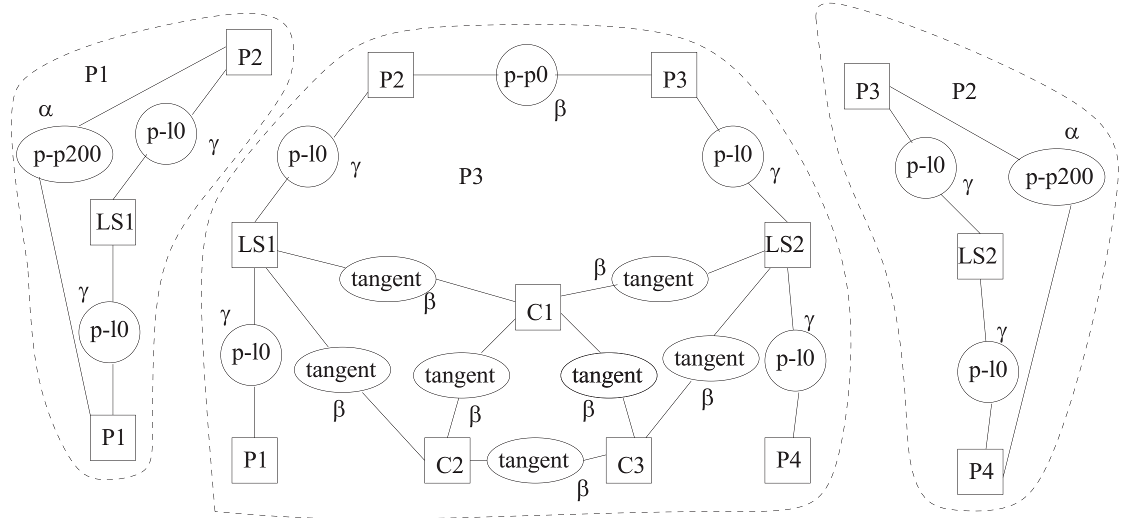


Figure 8.5: Decomposition of Constraint/Entity graph for problem R

The output of the decomposition strategy is therefore:

$$\{(P_i, \{Erep\})\} \cup \{(P'_j, \{IGCS\})\}.$$

For example, the constraint problem in figure 8.2 consists only of type α and type γ constraints. It will therefore be passed as a whole to Erep and solved using Erep only, as would be expected.

The constraint problem in figure 8.4 is split into three subproblems, P_1, P_2 and P_3 as shown in figure 8.5. P_1 and P_2 are solved using Erep. P_3 is solved using IGCS. The output of the decomposition strategy is

$$\{(P_1, \{Erep\}), (P_2, \{Erep\}), (P_3, \{IGCS\})\}.$$

Note that constraints such as P_2 lying on $LS1$ will be solved for twice. The ordering and solution strategies will deal with this problem. Note also that this decomposition strategy takes $\mathrm{O}(n)$ time, where n is the number of constraints.

This decomposition strategy is simple and powerful. In this example hybrid it is used to split a complicated problem into a number of subproblems, each of which has been associated with a constraint solver to solve it. A similar approach could be used to separate any constraint problem that has been labelled with constraints suited to a particular solver.

The most difficult part of this decomposition strategy will be determining the domain specific knowledge and specialities of the various constraint solvers in such

a way that constraints can be associated with particular solvers.

The fact that the decomposition strategy takes linear time in the number of constraints means that the overall time for the hybrid will not be dominated by the decomposition as is frequently the case. In fact, the time complexity of the hybrid is dominated by the time complexities of the individual solvers Erep and IGCS.

8.2 Ordering strategy

A common ordering strategy is to go from known values to unknown values. That is to say, solve easy problems first and use the solutions to the easy problems to find solutions to harder problems. This strategy has been usefully applied to such solvers as DESIGNPAK [100], DCM [86], INCES [62], Erep [14] and MechEdit [15].

Connectivity Analysis uses this approach in order to guide solution of residual sets. In effect, residual sets that are fixed are solved first, followed by residual sets that can be fixed using solutions from previous residual sets, and so on.

The 'known to unknown' strategy generalises to solving over-constrained subproblems first, followed by well-constrained subproblems and under-constrained subproblems. Over-constrained subproblems should be solved first as they are most likely to have no solutions. If an over-constrained subproblem has no solution, then the problem as a whole has no solution and so the over-constrained subproblem might as well be studied first. If an over-constrained subproblem *does* have solutions, it is likely that it will have only a very few of them. Over-constrained subproblems are more 'known' than well-constrained subproblems.

In a similar vein, well-constrained subproblems will likely have only a finite number of solutions and are more likely to have no solutions than under-constrained subproblems.

It *is* possible to identify the constrainedness of the subproblems generated by the decomposition strategy. However, the identification of constrainedness is nearly always as much work as the decomposition adopted by subsolvers. For example, Connectivity Analysis effectively identifies the residual sets of a constraint problem as a side-effect of investigating the constrainedness of the problem.

In order to avoid this effort the ordering strategy for the proposed hybrid of Erep and IGCS is very simple. Since Erep only finds solutions to well-constrained subproblems, it can be assumed that the subproblems passed to Erep are well-constrained (extraneous type γ constraints that are not part of a cluster confuse this argument, but this case is discussed in the next section). Similarly, subproblems

solved by IGCS can be considered to be under-constrained, as such problems play to the strengths of IGCS. Consequently, all Erep subproblems should be solved before all IGCS subproblems.

In the case of figure 8.5 this results in a partial order

$$P_1 < P_3,$$
$$P_2 < P_3.$$

Note that this ordering strategy is *very* simple and does not capitalise on the structure of the subproblems at all. This is because the ordering strategy, like the decomposition strategy, should be simple enough that it can be performed quickly and yet useful enough to help guide solution of the whole problem.

8.3 Solution and recombination

Recall that each subproblem P_i to be solved using Erep may contain extraneous γ constraints that are not part of a cluster. It is assumed here that Erep should solve P_i by trying to find as many clusters as possible, find solutions for each cluster and then stop. Extraneous type γ constraints and imposed entities will therefore not be solved for and will be left floating.

There are two situations whereby a type γ constraint may not be part of a cluster but may be part of a subproblem to be solved using Erep. The first is that the γ constraint lies in a subgraph of only γ constraints between an α constraint and a β constraint and is not in a cluster. In this case, the γ constraint will also be in a subproblem P'_j due to be solved using IGCS and so can be dealt with there.

Consider, for example, the constraint problem in figure 8.6. Constraints C_1, C_2 and C_3 are not in cluster Cl but are in subproblem P_1. When Erep tries to solve for C_1, C_2 and C_3, it fails and leaves them. However, as C_1, C_2 and C_3 are part of the all γ subgraph between C_0 and C_4, they are all also part of subproblem P'_2, which is due to be solved using IGCS. Consequently, Erep does not need to deal with them.

The other circumstance whereby a γ constraint may not be part of a cluster but may be a part of a subproblem due to be solved using Erep is when there is a subgraph consisting only of γ constraints connected to a cluster. For example, if circle $Circ_1$ and constraint C_4 are removed from figure 8.6, then constraints C_1, C_2 and C_3 will not be solved by Erep and are also not part of a subproblem due to be solved by IGCS.

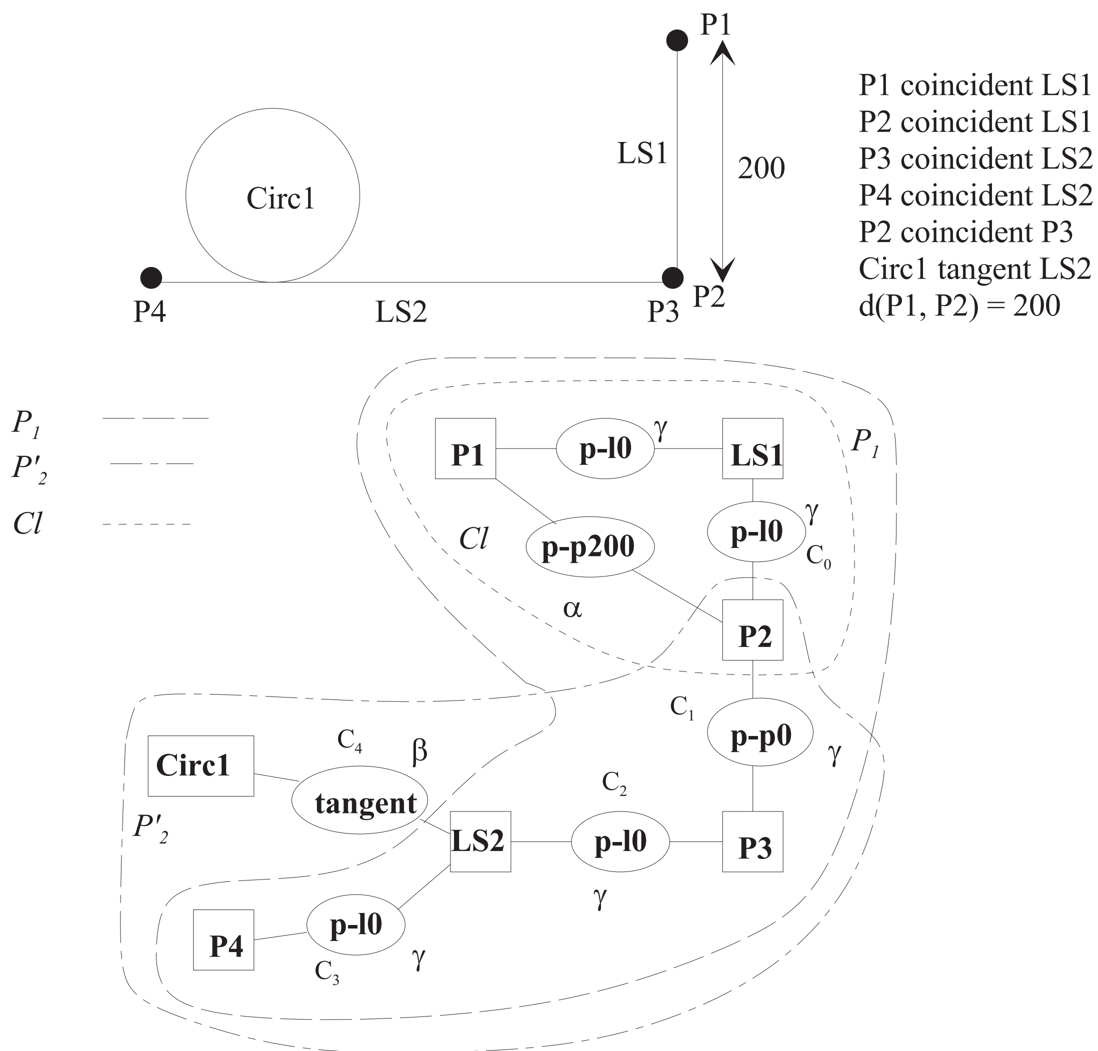


Figure 8.6: A constraint problem with α, β and γ constraints

The subgraph containing C_1, C_2 and C_3 can be solved by IGCS. Correspondingly, C_1, C_2 and C_3 should be made into a new subproblem that is due to be solved by IGCS and added to the stack of subproblems to be solved.

If a subproblem P_i exists that contains a subgraph containing α constraints that is not part of a cluster, then Erep will not be able to find solutions to the α constraints in the subgraph. IGCS cannot solve α constraints and so cannot solve the subgraph either. Consequently a hybrid of Erep and IGCS will not be able to solve such subproblems unless a backup, domain general solver such as Newton-Raphson or Gröbner bases is added. Such a situation is not addressed in this thesis but could form a powerful constraint solver.

The crudity of the ordering strategy means that no fine control of the hybrid can be deduced. Since the partial order only dictates that all Erep subproblems be solved before all IGCS subproblems, the following hybrid can be deduced:

$$((P_1, \{Erep\}) \parallel (P_2, \{Erep\}) \parallel (P_3, \{Erep\}) \parallel \ldots \parallel (P_n, \{Erep\}))\,;$$
$$((P'_1, \{IGCS\}) \parallel (P'_2, \{IGCS\}) \parallel (P'_3, \{IGCS\}) \parallel \ldots \parallel (P'_m, \{IGCS\}))$$

Note that each P_i is disjoint with each other P_i and similarly for each P'_j and so each Erep subproblem can be solved independently and each IGCS subproblem solved independently.

Recombination of solutions occurs when all Erep subproblems have been solved. At this point only *one* solution is produced by all of the applications of the Erep algorithm and so choosing the solution to pass to the IGCS subproblems is trivial. Note that it will certainly be possible to use the root identification algorithms of Erep to select other solutions to the Erep subproblems and that this will lead to other solutions of IGCS subproblems. Note also that the under-constrained nature and use of direct manipulation for the IGCS subproblems can be utilised with no effect on the Erep subproblems.

The ordering strategy forces subproblems to be solved by Erep completely first and then subproblems by IGCS. Consequently, there can be no cycles between Erep subproblems and IGCS subproblems.

Note that clusters in Erep describe rigid bodies with two translational degrees of freedom and one rotational degree of freedom. Since IGCS acts on rigid bodies with degrees of freedom, it seems likely that IGCS and Erep can be even more closely coupled.

8.4 Advantages of the Erep/IGCS hybrid

There are a number of advantages of the Erep/IGCS hybrid:

1. Reuses existing constraint solvers. The hybrid reuses existing solvers and so has less risk than developing a new system from scratch. It also allows for fast development.

2. Fast and efficient. The hybrid uses very simple decomposition and ordering strategies that each take time $\mathrm{O}(n)$, where n is the number of constraints. In contrast, Erep and IGCS take time $\mathrm{O}(n^2)$ each.

3. Simple to implement. Since the decomposition and ordering strategies are so simple, the hybrid should be simple to implement. The hybrid has not been implemented because the author has no access to the internal workings of Erep and has not had the time to construct the Erep algorithm from scratch.

4. Keeps strengths of individual solvers. The hybrid allows the use of both the root identification of Erep and the direct manipulation of IGCS. Both techniques have proved extremely popular and useful in exploring the solution space of a constraint problem.

5. Success when individual solvers fail. The hybrid will find solutions to constraint problems containing both type α and type β constraints. Individually, neither constraint solver handles both types and so the hybrid can handle more constraint problems. Although Erep *can* solve type β constraints by describing β constraints using degenerate distance constraints, this approach is discounted because it loses the domain specific knowledge that can be used to advantage in IGCS.

8.5 Limitations of the Erep/IGCS hybrid

There are limitations to the Erep/IGCS hybrid:

1. Inconsistent. Since only one solution is found for the Erep subproblems and passed on to the IGCS subproblems, by theorem 6.1, the hybrid is inconsistent.

2. Nonlinear time complexity. Although the decomposition and ordering strategies are fast, the solution of the subproblems dominates the hybrid. Since both

algorithms have $O(n^2)$ complexity, where n is the number of constraints, the complexity of the hybrid is $O(n^2)$.

Although the time complexity of the Erep/IGCS hybrid is equivalent to the time complexities of Erep and IGCS, it is important to remember that IGCS is not equivalent to the hybrid as IGCS cannot solve many of the constraints the hybrid can. Erep can solve all of the constraints the hybrid can but only if tangent constraints are reduced to distance constraints and the zero point-point distance constraint is removed by pre-processing. Both activities remove domain specific knowledge from the constraint problem that IGCS exploits. Thus one would expect the average case time complexity of the hybrid solver to be better than n^2 as it can exploit that domain specific knowledge.

8.6 Incremental implications of new solver

In section 3.2.5, the possibility of an incremental version of DCM was suggested. The solver developed in this chapter satisfies this vision of an incremental DCM. For the purposes of this discussion, Erep and DCM are used interchangeably. In fact, Erep and DCM are equivalent, as reported in [14].

An incremental constraint solver starts with a blank sheet of no entities and no constraints. A this point entities are added so that the constraint problem becomes under-constrained. Then constraints are added one by one so that the constraint problem becomes more and more constrained until it becomes well- and then over-constrained.

The Erep/IGCS hybrid cannot be made fully incremental for the same reasons that DCM cannot be made fully incremental. However, the hybrid as described in this chapter can take advantage of the incremental nature of IGCS and the compatibility of Erep and IGCS.

Starting with no entities and no constraints the incremental Erep/IGCS hybrid works as follows:

If a new entity is added, it forms a separate, disconnected constraint subproblem and is treated as such.

If a new constraint is added of type α, then the new constraint can only be solved by Erep. A new subproblem is therefore formed consisting of the new constraint together with any subproblems due to be solved by Erep connected to the new constraint. This new subproblem must be solved by Erep also. Consequently, the decomposition strategy is simple and uses most of the previous decomposition, and

the order of solution is much the same.

Solution of the new subproblem may not be possible in Erep as the subproblem may not form a rigid body. If this is so, then the subproblem is left in an undefined state until more constraints are added. A problem consisting entirely of type α constraints processed in this way will lead to the first type of incremental DCM described in section 3.2.5.

In a similar fashion, a new type β constraint is put into a new subproblem consisting of the new constraint plus any connected subproblems due to be solved using IGCS. Solution of the new subproblem can happen incrementally as IGCS processes the new constraint alone and tries to solve it.

If the new constraint is of type γ then it can be solved by either Erep or IGCS. Similar to the above two techniques, the new constraint is used to form two new subproblems to be solved using Erep and IGCS respectively. The Erep subproblem is solved first and will lead to some of the γ constraints in the subproblem being satisfied to give rise to rigid body clusters.

These rigid body clusters are well-constrained and are unlikely to be changed in the future. However, the rest of the subproblem will be solved by IGCS and will almost certainly change as the constraint problem evolves.

Therefore the rigid body cluster should be separated from the under-constrained part of the subproblem and put into a separate subproblem to be solved using Erep alone. In this way the rigid body is not resolved every time the under-constrained part is changed. This will save a great deal of effort on the part of the solver.

Thus, a constraint problem evolves from being under-constrained to being well-constrained and more of the problem becomes well-defined rigid bodies that have been solved using Erep and need not be resolved.

For example, consider the constraint problem in figure 8.7. For the purposes of this example, since all constraints are binary, the constraint graph of the problem is described using edges as constraints and vertices as entities. This helps to make the problem concise and clear in this case. The constraint graph for the problem is presented in figure 8.8.

This problem is well-constrained and can be solved entirely using Erep. Apart from the distance constraint between p_1 and p_2, it can also be solved entirely using IGCS. However, it can also be solved incrementally using both solvers. This incremental method is both intuitively obvious to a user and also an efficient means of reusing past solutions.

At the first iteration, the constraint graph consists of no entities and no con-

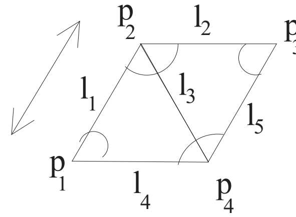


Figure 8.7: Two triangles with a common edge

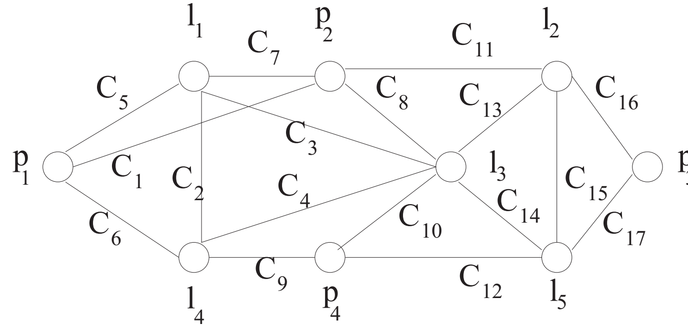


Figure 8.8: Constraint graph for figure 8.7

straints. At the second iteration, all of the entities are added. On the third iteration, the point-point distance constraint C_1 is added to give a new subproblem $P_1 = (\{p_1, p_2\}, \{C_1\})$ (see figure 8.9.

On the next eight iterations, the line-line angle constraints C_2, C_3 and C_4 and the point-line coincident constraints C_5, C_6, C_7, C_8 and C_9 are added so that the graph now looks like figure 8.10. Currently, there are two subproblems consisting of

$$P_1 = (\{p_1, p_2, p_4, l_1, l_2, l_3\}, \{C_1, \ldots, C_9\}),$$

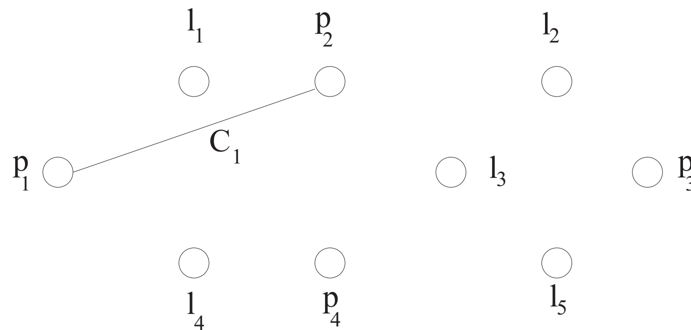


Figure 8.9: Iteration two of incremental solution of figure 8.7

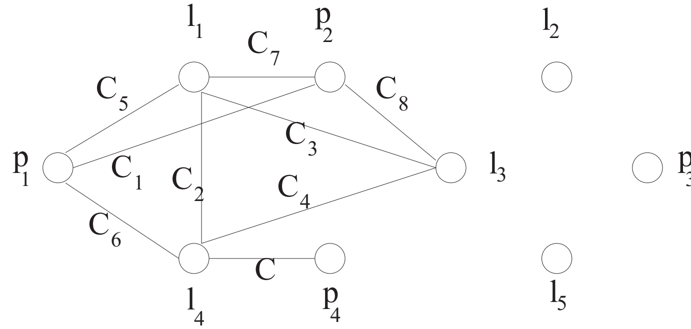


Figure 8.10: Iteration ten of incremental solution of figure 8.7

to be solved using Erep, and

$$P_2 = (\{p_1, p_2, p_4, l_1, l_2, l_3\}, \{C_2, \dots, C_9\}),$$

to be solved using IGCS.

At this point, Erep cannot solve P_1 at all and so P_2 is being solved using IGCS. This allows the user to directly manipulate the lines and points as desired and so explore the solution space. The status of C_1 is undefined.

On the next iteration, constraint C_10 is added and the decomposition strategy places C_{10} in both P_1 and P_2. P_1 can now be solved fully using Erep to give a rigid body, the left most triangle in figure 8.7. IGCS no longer solves any of the problem. Note that the constraint problem has suddenly gone from being under-constrained to being well-constrained.

Since the triangle is a rigid body and unlikely to be changed, it need never be resolved. Unfortunately, the sudden switch from the under-constrained geometry solved by IGCS to the well-constrained geometry solved by Erep means that all of the previous work done by IGCS has been discarded in place of the solution found by Erep. A better idea would be to use the information developed by IGCS to assist the solution found by Erep.

In successive iterations, constraints $C_{11}, C_{12}, C_{13}, C_{14}, C_{15}$ and C_{16} are added. At this point, there exist two subproblems,

$$P_1 = (\{p_1, \dots, p_4, l_1, \dots l_5\}, \{C_1, \dots, C_{16}\}),$$

and

$$P_2 = (\{triangle, p_3, l_2, l_5\}, \{C_{11}, \dots, C_{16}\}).$$

$C_{11}, \ldots, C_{16}$ are solved using IGCS so that the user can still manipulate the lines and points and the triangles as a rigid body. However, when C_{17} is added, the geometry is again well-constrained and is all solved using Erep.

However, note that the decomposition to identify the cluster representing the left most triangle has already been done and need not be recalculated. Only the constraints that are added need be solved for. This is equivalent to the first incremental DCM suggested in section 3.2.5.

This is a powerful incremental algorithm and is a natural extension of the hybrid developed in this chapter.

8.7 Conclusions

This chapter has presented the design of a new hybrid constraint solver. This solver is a hybrid of two previously existing constraint solvers, Erep and IGCS. Erep can find solutions for well-constrained geometric constraint problems, whereas IGCS can find solutions for under-constrained geometric constraint problems. The advantage of a hybrid between these two solvers is that it can find solutions to geometric constraint problems consisting of well- and under-constrained subproblems.

However, the identification of well- and under-constrainedness of subproblems is difficult. Middleditch and Latham [67] identify under- and over-constrained subproblems as part of identifying the residual sets used to solve the constraint problem. Since identifying the constrainedness of subproblems appears to be just as difficult as decomposing to residual sets, it was decided that the hybrid of Erep and IGCS would not try and identify well- and under-constrained subproblems.

Instead, the domain specific knowledge employed by each constraint solver was captured and studied. It was obvious from the list of constraints that each solver could handle that there were some constraints that Erep alone could solve, some that IGCS alone could solve and some that both could solve. The decomposition strategy assigns Erep constraints to subproblems to be solved using Erep and IGCS constraints to subproblems to be solved using IGCS.

It remained to determine what to do with constraints that could be solved by either constraint solver. Again, the domain specific knowledge used by both solvers could be used to solve this dilemma. Since Erep builds up subproblems into clusters and effectively clusters are the most basic elements of an Erep solution, Erep subproblems must have all of the constraints necessary to build a cluster. Since identifying a cluster would effectively mean solving it, all constraints that can be

solved by both Erep and IGCS are placed in both Erep and IGCS subproblems.

In this way, Erep will always have all of the constraints that it can possibly use to build a cluster and any that are left over will be solved using IGCS.

The ordering strategy that is used by the hybrid is that all Erep subproblems are solved first, followed by all IGCS subproblems. This is because all Erep subproblems are assumed to be well-constrained and therefore have fewer solutions than the under-constrained IGCS subproblems

The subproblems can then be solved using the appropriate solvers and solutions recombined to give solutions to the original constraint problem.

The advantages of the Erep/IGCS hybrid are that it reuses currently existing constraint solvers; has fast decomposition and ordering strategies and good average time complexity for solving constraint problems; retains the root identification of Erep and the direct manipulation of IGCS and will solve more problems than the individual solvers will. The disadvantages are that the hybrid may fail to find solutions to a constraint problem when some exist. It also has nonlinear time complexity.

Whilst the Erep/IGCS hybrid is new and powerful, it is untested. The process that this chapter followed to build the Erep/IGCS hybrid is significant as it has highlighted the principle areas that anyone constructing a coarse-grain decomposition hybrid must consider.

The incremental version of the hybrid constraint solver is also very powerful. The incremental version allows both the direct manipulation of IGCS and the root identification of Erep in an incremental context. The incremental hybrid allows constraint problems to be defined and evolved in an intuitive manner but also provides an efficient means of solving the constraint problem as it is created.

Chapter 3 asked whether it was better to have a complex decomposition strategy and simple solvers or a simple decomposition strategy and complex solvers. This chapter has demonstrated that it is possible to build efficient coarse-grain decomposition hybrid constraint solvers. However, the complexity of the Erep/IGCS hybrid is dominated by the complexity of Erep and IGCS themselves, which is $O(n^2)$, where n is the number of constraints. Consequently, it is unlikely that the Erep/IGCS hybrid lies on the trade-off point of figure 3.9.

Chapter 9

Future Work

The work presented in this thesis represents a step towards the goal of the Virtual Working Environment group at the University of Leeds. This goal is to develop an interactive environment for developing engineering designs within a Virtual Environment. This chapter introduces further research that must be carried out before this goal can eventually be realised. The research necessarily can be divided into three areas: the interactive constraint solver, the mathematical framework and the Virtual Environment used to develop the engineering designs. These areas are discussed in more detail below.

9.1 The interactive constraint solver

The interactive constraint solver suggested in this thesis takes advantage of domain specific knowledge as much as possible. Hopefully, this will make the constraint solver sufficiently fast that it will be possible to use it for interactive use. However, there are a number of factors that need to be studied further before the interactive constraint solver can be realised.

As a proof of concept the Erep/IGCS hybrid solver should be implemented and a number of case studies should be tested on it to see how it performs. This work is explored in section 9.1.1.

In order to have an interactive constraint solver consisting of subsolvers communicating with each other, there must be a common standard for describing constraint problems. This is discussed in section 9.1.2.

The case studies presented in this thesis help to demonstrate the power of hybrid constraint solvers. However, more complex case studies need to be explored in order to examine the constraint solver under pathological conditions. This is discussed in

more detail in section 9.1.3.

The incremental nature of constraint solvers such as INCES [62] and IGCS [112] help to improve the efficiency of the solver. Since the interactive constraint solver requires very efficient constraint solution, the use of incremental constraint solvers as part of the interactive constraint solver should be studied in detail. This is described in section 9.1.4.

9.1.1 The Erep/IGCS hybrid solver

The Erep/IGCS hybrid introduced in chapter 8 is an example of the type of interactive constraint solver proposed in this thesis. As such, it should be implemented, if only as a proof of concept solver. In fact, the Erep/IGCS hybrid promises to be quite a powerful solver and so should be tested on a number of case studies.

In particular, the Erep/IGCS hybrid's performance should be studied for constraint problems consisting mainly of constraints that can be solved only by Erep (type α constraints); mainly of constraints that can only be solved IGCS (type β constraints); mainly of constraints that can be solved by either solver (type γ constraints); and mixtures of all three constraints. Problem cases will probably be constraint problems consisting of mainly γ constraints that the constraint solver solves twice.

9.1.2 A standard interface for solvers

In order for an interactive constraint solver comprising of a hybrid of domain specific solvers to be implemented there must first be some form of standardised communication between the interactive constraint solver and the subsolvers used to solve the subproblems. Just as graphical standards were used to allow pictures to be shared and engineering design standards were developed to allow CAD designs to be shared, it will be necessary to have a standard format for describing constraints, entities and constraint problems so that they too can be shared amongst designers.

9.1.3 Complex case studies

The case studies presented in this thesis are not trivial. Case study 2 (section 7.6) has upwards of 2000 entities and 2000 constraints. Case study 2 is also very simple, as the constraints can all be solved using local propagation. However, in order to study the scalability of the concepts in this thesis, even larger case studies must be

considered. Also, more pathological case studies are necessary. The case studies presented here had only a few solutions from each subproblem passed onto the next. Case studies with 100s of solutions from each subproblem should be considered so that techniques of communication and collaboration of such cases can be explored.

9.1.4 Incremental issues

For performance reasons, incremental constraint solvers are becoming significant [62, 112]. Incremental constraint solution involves using previous knowledge to solve a constraint problem when a new constraint is inserted. Typically this means that the constraint problem is analysed so that the smallest part possible of the problem needs to be resolved when a new constraint is inserted or deleted. For obvious reasons, the incremental approach is ideal for interactive constraint solution. Not only do incremental approaches allow for a better average-case speed, but they also provide a natural paradigm for users to progressively add and remove constraints and entities. The other alternative to incrementally defining a model is the specify-then-solve approach used by many other constraint solvers [14, 86].

Incremental constraint solvers have been studied in detail in this thesis. They have great potential for improving the performance of constraint solvers and also of providing more useful feedback to users. In particular, constraint solvers that can follow a constraint problem from being under-constrained to being well-constrained to being over-constrained will be valuable. The constraint solver described in chapter 8 can handle geometric constraint problems as they progress from being under-constrained to being well-constrained. The incremental version of the Erep/IGCS algorithm should therefore be implemented and tested. The next stage would then be to enhance the incremental Erep/IGCS hybrid so that over-constrained constraint problems can be dealt with.

9.2 The mathematical framework

The mathematical framework presented in this thesis is capable of capturing many different types of constraint problem and constraint solver. However, there remain certain types of constraint that are difficult to describe and difficult to solve. These include inequality constraints and probabilistic constraints, which are discussed in sections 9.2.1 and 9.2.2.

9.2.1 Inequality constraints

As noted by Lamounier [64], during the design process many design specifications and performance measures are defined as inequalities. For example, in mathematical programming, many of the constraints are described using inequalities. In the framework given in this thesis the dimension of an inequality constraint is frequently 0 as there will often be a homeomorphism from the original solution space to the restricted solution space caused by the inequality. For example, if a constraint $x < 10$ is imposed on the real line $(-\infty, +\infty)$ to give a solution space of $(-\infty, 10)$, then there is a simple homeomorphism $x \mapsto -e^{-x} + 10, \quad (-\infty, 10) \mapsto (-\infty, +\infty)$. Consequently the dimension of the constraint $x < 10$ is 0.

The fact that most inequality constraints have a dimension of 0 means that they are rarely used to progress towards a well-constrained solution. Instead, inequality constraints are usually used to refine solutions and to select between them. Consequently they form an important type of constraint to deal with. However, little research has been done on the implications of inequality constraints within a larger problem.

9.2.2 Probabilistic constraints

Bistarelli *et al.* [10] use the general semiring framework to describe many different types of constraint problem. One of the types of constraint problem covered by the semiring formalisation is *fuzzy constraint problems*. Fuzzy constraint problems are equivalent to the constraint priority constraint problems presented in section 6.3.1. Bistarelli *at al.* also use the semiring construction to describe *probabilistic constraint problems*. Probabilistic constraints have a certain probability to be part of the given problem. This allows reasoning about problems which are only partly known. It should be possible to model probabilistic constraint problems and solutions using the framework in this thesis.

9.3 The Virtual Environment

Although the Virtual Environment used is not directly significant to this thesis, there are a number of issues that are relevant to both the Virtual Environment and the framework presented in this thesis. These include the use of parallel and concurrent collaboration, discussed in section 9.3.1 and the use of direct manipulation to interact with the interactive constraint solver, explored in section 9.3.2.

9.3.1 Parallel/concurrent collaboration

Sequential collaboration is quite common and most hybrids currently in use adopt this paradigm [15, 62, 87, 112]. However, there are advantages to parallel and concurrent collaboration. Parallel collaboration can be simply implemented on parallel architectures with the obvious performance advantages available from such a move. Concurrent collaboration allows the investigation of design alternatives, an important issue for designers.

The implementation of a parallel collaboration would require careful consideration of the intersection of the solution spaces found by the parallel solvers. Implementation of a concurrent collaboration would require study of the use of the choice function and how it could be incorporated into a real-time algorithm.

9.3.2 Direct manipulation issues

One of the great advantages of ICBSM [26, 32] is that it allows direct manipulation of geometric entities to build up complex assemblies. This proved to be a powerful means for designers to build models. Consequently it would be advantageous if the direct manipulation approach could be used for the interactive constraint solver. Obviously direct manipulation only applies to entities that have some sort of visual representation.

The Erep/IGCS hybrid developed in chapter 8 retains the use of the direct manipulation of IGCS and the root identification of Erep for the subproblems they respectively solve. Research needs to be done to see how well these two approaches interact and whether either can be applied to other constraint problems.

9.4 Summary

The work presented in this thesis describes work done towards the goal of developing an interactive environment for developing engineering designs within a Virtual Environment. The mathematical framework developed here allows the description of complex hybrid constraint solvers consisting of a number of collaborating domain specific constraint solvers. The sound mathematical framework developed allows for an investigation of the structure of hybrid constraint solvers and of the quality of solution provided by hybrid constraint solvers.

The next key step in this research will be to investigate incremental hybrid constraint solvers of domain specific solvers, such as the Erep/IGCS hybrid developed

in chapter 8. This will involve complex engineering design case studies which can be used to investigate the interaction of the domain specific solvers and the power of the hybrid constraint solver.

Chapter 10

Conclusions

The construction of constraint problems in engineering design is supplanting more traditional design techniques [47]. Consequently, many constraint solvers have been developed to handle the constraint problems being constructed. The constraint solvers range from geometric constraint solvers, capable of finding solutions to complex descriptions of geometry and relative positioning of geometric constructs, to functional constraint solvers, capable of finding solutions to descriptions of the function of designs.

It is generally acknowledged that there is a close link between geometric and functional constraint problems [3, 19, 64, 100]. It is inadvisable to separate the geometric and functional parts of the constraint problem as they affect each other closely. Correspondingly, practitioners attempt to solve the combined constraint problem as a whole. In a similar vein, other types of constraint problem, such as finite domain, scheduling and physical constraint problems, should be considered within a general framework.

Constraint solvers handle complex combinations of different types of constraint problem in one of two ways. Either the combined problem is converted into a large system of equations and inequalities and then solved using numerical or symbolic techniques or a part of the combined problem is tackled by a solver good at solving that particular type of problem and when the solver gets stuck, it is assisted by numerical or symbolic techniques.

Numerical solution is slow and prone to numerical convergence problems, as well as a tendency to converge to non-intuitive solutions and only to find one solution. Symbolic solution, using Gröbner bases [17], whilst being much more robust and finding *all* solutions to the combined problem, is far too slow for use in interactive design [16].

The use of a *domain specific* constraint solver as far as possible leads to a significantly faster algorithm [15]. However, the domain specific solver used is usually quite limited in the problems it can handle and the symbolic or numerical backup solver will be consulted frequently. Using two collaborating constraint solvers in this way results in a *hybrid* constraint solver.

The use of domain specific solvers leads to very fast constraint solvers such as Fa's ICBSM [27]. ICBSM is capable of solving geometric constraint problems without loops in time $O(n)$, where n is the number of constraints in the problem. It is therefore suitable for interactive use. In addition, ICBSM is *incremental* and will find solutions to the constraint problem when a new constraint is added in time $O(1)$.

However, ICBSM is very restricted in the type of geometric constraint problems it can solve. ICBSM cannot solve geometric constraint problems with loops in them. ICBSM is also unable to solve functional constraint problems and so is not an ideal candidate for an engineering design constraint solver.

The purpose of this thesis was to study ways of applying Fa's Allowable Motion to more general constraint problems. In particular, engineering design constraint problems consist of loops and functional constraints.

In order to study the possible applications of Allowable Motion to more general constraint problems, it was first necessary to investigate the current state-of-the-art in constraint solution. Chapter 2 presents the result of this study. Constraint solvers in the finite domain, geometric, functional and physical fields were examined in detail and the strengths and weaknesses of each solver were identified. This study identified the specialisation of current solvers in terms of type of problem solved and also the constrainedness of problem handled.

The characterisation of constraint solvers in chapter 2 brought to light the common nature of most constraint solvers. In particular, most existing constraint solvers use a divide-and-conquer approach to solve a constraint problem. First a decomposition strategy is used to split the constraint problem into smaller subproblems. Then the subproblems are ordered. Finally, the subproblems are solved in the order given by the ordering strategy.

Several issues were raised by the divide-and-conquer approach and in order to study these issues the constraint process was examined in detail by dividing it into *constraint definition*, *constraint representation*, *constraint satisfaction* and *representation of solutions*. Since the representation of solutions to constraint problems is a complex issue in itself, it was not studied in this thesis. The reader is referred

to [14] for a discussion of the root identification problem in geometric constraint problems.

The constraint definition phase of the constraint process was studied in chapter 4. Intensive study of many different types of constraint solver and the problems that the solvers can handle led to a coherent, comprehensive definition of general constraint problems. This definition is capable of describing many different types of constraint problem as well as constraint problems that consist of a number of different types of constraint. The set-theoretic approach used in the definition allows the development of a detailed and abstract framework for the constraint satisfaction process. The *dimension* of a constraint was also introduced as a means of identifying the progress towards solution of a constraint process.

Chapter 5 explored the second phase of the constraint process, constraint representation. In order to solve constraint problems efficiently, many constraint solvers rely on a *constraint representation scheme* that allows structure of a constraint problem to be investigated. Most such constraint representation schemes use graph techniques as there already exists a comprehensive selection of literature on the properties of graphs. However, different constraint solvers use different constraint representation schemes. If a constraint solver capable of solving general engineering design constraint problems were to be developed, it would ideally be capable of describing general constraint problems, using the definition developed in chapter 4.

Consequently, chapter 5 studied the various constraint representation schemes currently in use and how they were related to each other. This led to the identification of a *generic* constraint representation scheme, one capable of describing general constraint problems. Using *reductions* it was possible to identify which constraint representation schemes were generic and which were not. Reductions can also be used to translate between representation schemes as and when required.

Different constraint solvers use different solution techniques depending on the structure and type of constraint problem being solved. However, the investigation of the state-of-the-art in chapter 2 indicated that all of the constraint solution techniques studied followed much the same underlying pattern. Chapter 6 explicitly captured this underlying structure by defining *solution steps* and *solution processes*. Using the notion of a *solution space*, it was then possible to describe the properties of a solution process.

The *quality of solution* of a solution process is very important to a designer and is dependent on the properties of the solution process. For example, if a solution process is *consistent*, then the designer knows that there is *a* solution to the con-

straint problem in the terminal solution space. If a solution process is *sound* then the terminal solution space contains *only* solutions to the constraint problem. If the solution process is *complete* then the terminal solution space contains *all* solutions to the constraint problem. However, it was very difficult to determine if a specific constraint solver was sound, complete or even consistent. Theorem 6.1 can be used to identify the quality of solution of a solution process. Theorem 6.1 is also a significant tool in the study of hybrid constraint solvers as it allows designers to identify the quality of solution of hybrid constraint solvers, as well as simple constraint solvers.

The power of the satisfaction abstraction was demonstrated by using it to describe a number of different extensions of the basic constraint problem, including constraint priority problems, incremental constraint satisfaction, backtracking and variable-driven constraint satisfaction.

Using the abstractions of the constraint definition, representation and satisfaction phases, chapter 7 studied the use of hybrids and domain specific constraint solvers. Domain specific solvers were identified as being fast and efficient but not general enough for general engineering design. Hybrid constraint solvers were investigated using an adaptation of Monfroy's BALI environment [84]. Monfroy identifies three different paradigms for the collaboration of constraint solvers. These are sequential, parallel and concurrent collaboration. All three collaboration paradigms were defined in terms of the satisfaction framework of chapter 6. Theorem 6.1 can then be used to study the quality of solution found by sequential collaboration, whilst an equivalent theorem, theorem E.1, can be used to study the quality of solution found by parallel collaboration.

Several examples were then explored using these collaboration paradigms. The example in section 7.3 demonstrated that even a very simple hybrid constraint solver of two domain specific solvers could produce a marked increase in speed for solving a constraint problem consisting of functional and geometric constraints. The example in section 7.6 investigated the asymptotic behaviour of the sequential collaboration and found that the hybrid of domain specific solvers was linear in complexity whereas other solvers were not. A constraint problem with 2000 constraints was solved in 0.25 seconds using the hybrid but 60 seconds using INCES [62] and 180 seconds using a numerical algorithm.

The discussion of the incremental version of DCM in section 3.2.5 along with the solution framework presented in chapter 3 and the work done on hybrid constraint solvers and domain specific solvers in chapter 7 naturally suggested a collaboration

of two existing constraint solvers, IGCS [112] and Erep [14]. The Erep/IGCS hybrid was defined in terms of a simple decomposition strategy, a simple ordering strategy and a description of how solution would proceed. The Erep/IGCS hybrid has not currently been implemented but serves as a useful demonstration of the process of constructing a new constraint management system as discussed in chapter 3. It also has the potential to be a powerful solver in its own right.

In particular, the Erep/IGCS hybrid can be implemented to give a powerful incremental constraint solver. This incremental solver allows a constraint problem to be developed an manipulated from an initial state of being under-constrained to being well-constrained. The incremental solver retains the root identification of Erep and the direct manipulation of IGCS and so allows the user to explore the solution space of the problem as desired.

In chapter 3, a number of issues were raised regarding the decomposition framework:

1. Is it possible to lose solutions by decomposing and recombining?
2. What effect does decomposition have on incremental techniques?
3. Is it more efficient to decompose and recombine or solve as a whole?
4. Is it possible to have a fast decomposition strategy and a fast solution of subproblems or must one always dominate?

These questions have been addressed throughout this thesis. Progress on the answers are presented below:

1. *Is it possible to lose solutions by decomposing and recombining?* It is possible to lose solutions be decomposing and recombining. Theorems 6.1 and E.1 prove that if a constraint solver is not globally consistent then it cannot be linked with other constraint solvers and expect to be globally consistent. In fact, few current constraint solvers are globally consistent and so few hybrid constraint solvers are globally consistent. In particular, INCES and IGCS were both identified as being inconsistent.
2. *What effect does decomposition have on incremental techniques?* It is possible to apply an incremental approach to decomposition, ordering and solution. Incremental decomposition strategies need to reuse information from a previous decomposition as much as possible. Incremental ordering strategies need to order any new subproblems in such a way as to not need to resolve as

many subproblems as possible. Incremental subsolvers can find solutions to problems with newly added constraints quickly. The Erep/IGCS hybrid is a good example of an incremental constraint solver and it makes good use of the natures of both Erep and IGCS to give a powerful solver that can be used interactively.

3. *Is it more efficient to decompose and recombine or solve as a whole?* Generally speaking, it is better to decompose and recombine than to solve as a whole. General constraint solvers such as Gröbner bases or numerical techniques are too slow for interactive use. However, the choice of which approach to take may not be predicated by speed alone. The lack of global consistency in hybrid constraint solvers means that general constraint solvers may be more appropriate for some problems.

4. *Is it possible to have a fast decomposition strategy and a fast solution of subproblems or must one always dominate?* The hybrid Erep/IGCS constraint solver is an example of a constraint solver with a fast decomposition strategy but slow solution. Most other constraint solvers have a slow decomposition strategy but fast subsolvers. The graph in figure 3.9 suggests that there may be a trade-off point where solvers have fast decomposition strategies and fast subsolvers, but the Erep/IGCS hybrid is probably not at this trade-off point.

The contribution of this thesis is the study of hybrids of domain specific constraint solvers using coarse-grain decomposition for solving general engineering design constraint problems. To this end

- A new categorisation of constraint solvers in terms of *type* of problem solved and *constrainedness* of the solver was introduced and used to categorise a number of current constraint solvers.

- The characterisation of constraint solvers led to the identification of the divide-and-conquer approach used by most current constraint solvers and the pros and cons of this approach were studied in detail.

- An abstraction of the constraint process was created that allows the definition, representation and satisfaction of general engineering design constraint problems in a high-level, general fashion.

- A study of the *quality of solution* provided by constraint processes led to theorem 6.1 which allows statements to be made about the quality of solution

of constraint solvers and sequential hybrid constraint solvers by examining the individual steps used by the solver. Theorem E.1 allows similar statements to be made about parallel hybrid solvers.

- A study of the various hybrid collaboration paradigms suggested by Monfroy [84] was made and the advantages and disadvantages of each investigated. The mathematical framework developed allows concrete statements to made about the nature of these collaborations. Monfroy's solver collaboration language was extended to allow for the decomposition framework and to allow conditional application of solvers.

- A new constraint solver has been defined that consists of a hybrid of Erep and IGCS. This constraint solver combines the ability of Erep to solve well-constrained geometric constraint problems with loops and the ability of IGCS to solve under-constrained geometric constraint problems.

The framework developed in this thesis can be used to help bridge the gap between geometric and functional constraints and consequently forms an important tool towards efficient, interactive engineering design by constraints. However, the framework is sufficiently general that it is not restricted to geometric and functional constraints and can also include relational algebra, finite domain constraints, scheduling constraints and physical constraints.

Appendix A

Dimensions

This appendix presents a definition of the dimension function $dim : domain \longrightarrow \mathbb{N}$. The dimension function should have the following properties

$$\begin{gathered} dim(\mathbb{R}) = dim(\mathbb{Q}) = dim(\mathbb{Z}) = dim(\mathbb{N}) = 1 \\ dim(\emptyset) = dim(A) = 0, A \text{ is a finite set,} \\ dim(D_1 \times D_2) = dim(D_1) + dim(D_2), \\ dim(A \cup B) = \max(dim(A), dim(B)), \\ dim(A \cap B) \leq \min(dim(A), dim(B)), \\ dim(A \setminus B) \leq dim(A). \end{gathered}$$

The dimension function defined here is taken from [107]. It uses *manifolds* to capture the above properties. It is assumed that all of the relations and sets used in the constraint problem exist within a metric space M with metric d. It is also assumed that the reader is familiar with the concept of metric spaces. Readers unfamiliar with metric spaces are referred to [109]. Given the metric space M, then the following definitions are important for defining a manifold.

Definition A.1 (Open Ball) *Given a metric space $M = \{A, d\}$, a point a in A and a strictly positive real number ϵ, the* **open ϵ-ball neighbourhood of a in M** *is the set*

$$B_\epsilon(a) = \{x \in A : d(x, a) < \epsilon\}.$$

□

An open ball in the real line is equivalent to an open interval of size 2ϵ. The

interval $(x-\epsilon, x+\epsilon)$ in the metric space $\{\mathbb{R}, d\}$, where $d(x,y) = |x-y|$ is an open ϵ-ball in the neighbourhood of x. In $\mathbb{R}^2$, open ϵ-balls take the form of circles of radius ϵ about a point and in $\mathbb{R}^3$, open ϵ-balls take the form of spheres of radius ϵ about a point.

Definition A.2 (Open Sets) *A set U is* **open** *in metric space M if $\forall y \in U, \exists \epsilon > 0$, such that $B_\epsilon(y) \subset U$.* □

In the metric space $\{\mathbb{R}, d\}$ as above, any open interval (a,b) is an open set. Any half open interval, such as $(a,b]$ or $[a,b)$, or any closed interval $[a,b]$ is not open.

Definition A.3 (Homeomorphisms) *Bijection $f : T_1 \to T_2$ is a* **homeomorphism** *if*

$$U \textit{ open in } T_1 \Leftrightarrow f(U) \textit{ open in } T_2.$$

□

Two sets which are homeomorphic to each other are equivalent in most respects. For example $\mathbb{N}$ is homemorphic to $\mathbb{Z}$ and $\mathbb{Q}$, and the open interval $(0,1)$ is homemorphic to the real line $\mathbb{R}$. However, $\mathbb{R}$ is not homeomorphic to $\mathbb{R}^2$.

Given these definitions, it is possible to define an n-manifold as follows:

Definition A.4 (n-manifold) *An n-manifold is a metric space M such that for all $x \in M$, there is some neighbourhood U of x and some integer $n \geq 0$ such that U is homeomorphic to $\mathbb{R}^n$.* □

An n-manifold is said to have dimension n. Manifolds form an ideal candidate for the dimension function desired.

Definition A.5 (Dimension) *A set S in a metric space M has dimension n, $dim(S) = n$, if and only if n is the smallest number such that S is an n-manifold in M.* □

From this definition, it is immediate that

$$\begin{aligned}
dim(\mathbb{N}) &= 1, \\
dim(\mathbb{Z}) &= 1, \\
dim(\mathbb{Q}) &= 1, \\
dim(\mathbb{R}) &= 1, \\
dim(A) &= 0, (\text{as} \mathbb{R}^0 = \text{finite sets}) \\
dim(\emptyset) &= 0.
\end{aligned}$$

However, by this definition, if a set B is countable, then

$$dim(B \times B) = dim(\mathbb{N}) = 1,$$

as there is always a homeomorphism between a Cartesian product of countable sets and the set of natural numbers. This contradicts property A.1 above.

Standard results from manifold theory can be used to prove that the dimension function above does have all the properties desired. In particular, the following theorem can be used to satisfy property A.1 (taken from [107]).

Theorem A.1 If M_1 and M_2 are manifolds of dimension n_1 and n_2 respectively, then $M_1 \times M_2$ is an $(n_1 + n_2)$-manifold. □

However, note that this theorem does not quite clear up the ambiguity of $\mathbb{Z}^2$. Spivak notes the example of a torus which is the Cartesian product of two circles. Spivak says that the torus is homeomorphic to a subset of $\mathbb{R}^4$, thus making it a 4-manifold. However, it is also homeomorphic to a subset of $\mathbb{R}^3$, meaning that it is also a 3-manifold.

The dimension function clears up this amibiguity by insisting that the set S has dimension equal to the *smallest* n-manifold that it is homeomorphic to. Correspondingly, not all Cartesian products satisfy property A.1 and the property is amended to reflect this (see section 4.5.1).

Appendix B

Constraint Representation Scheme Reductions

This appendix presents the details of the remaining reductions necessary to form the hierarchy of constraint representation schemes in figure 5.2. They are:

- Undirected graph $\longrightarrow$ directed graph,
- Bipartite graph $\longleftrightarrow$ hypergraph,
- Bipartite graph $\longleftrightarrow$ binary graph in finite case.

Theorem B.1 Undirected constraint graphs, for example those used by Owen [86] and Erep [48], are strictly less powerful than the directed graphs used by, for example ICBSM [27]. $\square$

Proof The proof is in two parts. First a reduction is formed from the undirected constraint graph to the directed constraint graph, demonstrating that the directed graph can describe at least as many problems as the directed graph.

The weakness of the undirected graph is that it cannot describe non-symmetric constraints.[†] The vast majority of constraints are non-symmetric. However, there are a significant number of symmetric geometric constraints and some constraint representation schemes, such as undirected constraint graphs ([48, 86]) take advantage of this.

The significance of symmetric constraints is that the ordering associated with the constraint is only a partial ordering. For example, for the constraint $x = y$, it does not matter which order the values of x and y are checked in, the constraint is still the same. Testing $(3, 4)$ and $(4, 3)$ gives the same result.

In an undirected graph there is no concept of ordering of binary constraints. It is not therefore possible to describe anything but symmetric constraints and so it is assumed that all constraints in an undirected constraint graph are symmetric constraints. It is this fact that will be exploited in this proof.

The mapping used to reduce an undirected constraint graph (V, E) to a directed constraint graph (V_D, E_D) is as follows:

> *Create* $V_D = V$.
>
> *For each edge* $e \in E$*, choose an orientation of that edge randomly so that, for example,* (u, v) *in the undirected graph becomes* $[v, u]$*. Place the directed edge in* E_D.

By inspection, the resultant graph is a directed constraint graph. Checking the reduction criterion:

1. The mapping is defined above.
2. Since the constraints in the undirected graph are symmetric and both graphs use the same CTP, any solution to the directed graph is a solution to the undirected graph. Similarly, any solution to the directed graph must be a solution of the undirected graph.
3. Both are valid CRSs.
4. The reduction can be done in linear time in the number of edges in the undirected constraint graph. It is therefore polynomial time.

So the undirected graph can be reduced to a directed graph representation.

Secondly, it is necessary to prove that the directed graph cannot be reduced to the undirected graph. This implies that directed graph representations are strictly more powerful than undirected graph representations as required. It suffices to find a counter example. The weakness of undirected graph representations is that they require all of the constraints to be symmetric.

Consider then, the following problem:

$$\begin{aligned} P &= (E, C), \\ E &= \{(x, \mathbb{R}), (y, \mathbb{R}), (z, \mathbb{R})\}, \\ C &= \{x < y, y = z, z < x\}. \end{aligned}$$

There is an obvious directed graph representation of the problem, but no obvious undirected representation as the two constraints $x < y$ and $z < x$ are non-symmetric. It is necessary to retain an ordering in order to describe these constraint non-ambiguously. Since an undirected graph does not preserve order, not all problems that can be described using a directed graph can be described in an undirected constraint graph representation. Thus, it follows that the directed graph representation is more powerful than the undirected graph representation as desired. □

Theorem B.2 The bipartite graph CRSs, such as the Constraint/Entity graph in this report and the schemes of Tsang [114] and Middleditch and Latham [66], are equivalent to hypergraph representation schemes, such as Serrano's Connectivity Network [101]. □

Proof Again the proof is in two parts. First a reduction is formed from a bipartite graph representation scheme to a hypergraph representation scheme. The mapping used is as follows:

> *Given the bipartite graph (C, V, E), create hypergraph (V_H, HE).*
> *For each vertex $x_i \in V$ create entity vertex x_i in V_H.*
> *For each $c_i \in C$, construct ξ_{c_i} such that $\xi_{c_i} = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$, the set of entity vertices adjacent to c_i in E. Then construct the hyperedge he in HE,*
>
> $$he = \xi_{c_i},$$
>
> *labelled as constraint c_i with CTP f_{c_i} and ordering $<_{c_i}$.*

By inspection, the resulting graph is a hypergraph. Checking the reduction criterion.

1. Map is defined above.
2. Since the relation c_i is borrowed and the ordering of the entities retained, the same problem is being solved, so all solutions in the bipartite graph are solutions in the hypergraph and vice versa.
3. Both are valid CRSs.

4. For each entity in E a vertex in the hypergraph is created, taking O(n). For each constraint in C one hyperedge is created, taking O(mn) in all. This is a polynomial time algorithm.

Secondly, a reduction is formed from the hypergraph to the bipartite graph as follows:

> *Given the hypergraph* (V, HE)*, create bipartite graph* (V_B, C_B, E_B)*.*
> *For each variable* $x_i \in V$*, construct entity vertex* x_i *in* V_B*.*
> *For each constraint* $c_i \in HE$*, construct a constraint vertex labeled* c_i *in* C_B*.*
> *Create edge* $(x_i, c_j, k), x_i \in V_B, c_j \in C_B, k \in \mathbb{Z}$ *if* $he \in HE, x_i \in he$*,* $c_j \in he$ *and* x_i *is in the* k*th position of* $<_{c_j}$*.*

The resultant graph is a bipartite graph. Checking the reduction criterion.

1. Map defined above.

2. Since the constraints are tested using the same test procedures in both representations, a solution in one will be a solution in the other.

3. Both are valid CRSs.

4. For each entity in V, an entity vertex in the bipartite graph is created, taking O(n). For each constraint in C, a constraint vertex in the bipartite graph is created, taking O(m). Each constraint is imposed on at most O(n) entities so at most O(n) edges per constraint are created, hence O(mn) in total. Hence the reduction is polynomial.

Hence the bipartite and hypergraph representation schemes are equivalent. □

Theorem B.3 When dealing only with finite domain problems, hypergraph graph representations and binary constraint graph representations are equivalent. □

Proof This is a proof oft-quoted in the literature, though not explicitly called a reduction. It is usually discussed as a method of describing n-ary constraints in binary graphs, but can be interpreted as a reduction from a hypergraph representation scheme to a binary graph representation scheme (see Tsang [114], p12, for example). The reverse reduction is, of course, trivial, as binary graphs are already hypergraphs. □

Appendix C

Using local properties to draw conclusions about processes

Theorem C.1 We wish to prove the following:

a. Strongly Complete $\Leftrightarrow$ Complete

b. Locally Complete $\not\Rightarrow$ Strongly Locally Complete

   Strongly Locally Complete $\Rightarrow$ Locally Complete

c. Locally Complete $\Leftrightarrow$ Complete

d. Strongly Consistent $\Leftrightarrow$ Consistent

e. Locally Consistent $\Rightarrow$ Strongly Locally Consistent

   Strongly Locally Consistent $\not\Rightarrow$ Locally Consistent

f. Locally Consistent $\Leftrightarrow$ Consistent

g. Strongly Sound $\Leftrightarrow$ Sound

   Sound $\not\Rightarrow$ Strongly Sound

h. Locally Sound $\not\Rightarrow$ Strongly Locally Sound

   Strongly Locally Sound $\Rightarrow$ Locally Sound

i. Locally Sound $\Leftrightarrow$ Sound

$\square$

Lemma C.1 *Given solution process* $\boldsymbol{\mathcal{D}}(0) \stackrel{\Psi}{\longrightarrow}^* \boldsymbol{\mathcal{D}}(\kappa)$ *and constraint problem* $P = (\Phi, \Psi)$

a $\boldsymbol{\mathcal{D}}(l) \subseteq \boldsymbol{\mathcal{D}}(l-1), \quad l = 1..\kappa$

b $A \subseteq B \Rightarrow C \cap A \subseteq C \cap B$

c $A \subseteq B \Rightarrow \boldsymbol{\mathcal{C}}(B) \subseteq \boldsymbol{\mathcal{C}}(A), \quad A, B \subseteq \Psi$

□

Proof

a Trivial from definition of solution process

b Assume $A \subseteq B$

$$\begin{aligned} x \in C \cap A \quad &\Rightarrow \quad x \in C \wedge x \in A \\ &\Rightarrow \quad x \in C \wedge x \in B \text{ (as } A \subseteq B) \\ &\Rightarrow \quad x \in C \cap B \\ &\Rightarrow \quad C \cap B \subseteq C \cap A \text{ (by definition of } \subseteq) \end{aligned}$$

c Let $A = \{\Psi_{A_1}, \ldots, \Psi_{A_k}\}, \; B = \{\Psi_{B_1}, \ldots, \Psi_{B_l}\}$. Then

$$\boldsymbol{\mathcal{C}}(A) = \bigcap_{x \in A} x, \quad \boldsymbol{\mathcal{C}}(B) = \bigcap_{y \in B} y$$

If $A \subseteq B$ then we can reorder $\boldsymbol{\mathcal{C}}(B)$ such that

$$\boldsymbol{\mathcal{C}}(B) = \boldsymbol{\mathcal{C}}(A) \cap \bigcap_{y \in B \setminus A} y$$

Then we have

$$\begin{aligned} x \in \boldsymbol{\mathcal{C}}(B) \quad &\Rightarrow \quad x \in \bigcap_{y \in B} y \\ &\Rightarrow \quad x \in \boldsymbol{\mathcal{C}}(A) \cap \bigcap_{y \in B \setminus A} y \\ &\Rightarrow \quad x \in \boldsymbol{\mathcal{C}}(A) \text{ (from definition of } \cap) \\ &\Rightarrow \quad \boldsymbol{\mathcal{C}}(B) \subseteq \boldsymbol{\mathcal{C}}(A) \end{aligned}$$

□

Proof (of theorem C.1)

a. *Strongly Complete* ⇒ *Complete*

Given

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(l-1) \subseteq \boldsymbol{\mathcal{D}}(l), \forall l = 1..\kappa \tag{C.1}$$

Required to prove

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(\kappa)$$

Proof Assume $\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(l-1),\ \forall l = 1..\kappa$ then

$$\begin{aligned} & \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(l-1) \cap \boldsymbol{\mathcal{C}}(\Psi) \\ \Rightarrow\ & \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(l-1) \cap \boldsymbol{\mathcal{C}}(\Psi) \text{ (as } S \cap S = S) \\ \Rightarrow\ & \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(l) \text{ (by C.1)} \end{aligned}$$

Hence by induction, since $\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(0)$, we have

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(\kappa)$$

as required □

Complete ⇒ *Strongly Complete*

Given

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(\kappa)$$

Required to prove

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(l-1) \subseteq \boldsymbol{\mathcal{D}}(l), \forall l = 1..\kappa$$

Proof

$$\begin{aligned} \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(l-1) &\subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \text{ (since } \boldsymbol{\mathcal{D}}(l-1) \subseteq \boldsymbol{\mathcal{D}}(0)) \\ &\subseteq \boldsymbol{\mathcal{D}}(k) \text{ (as the process is complete)} \\ &\subseteq \boldsymbol{\mathcal{D}}(l) \text{ (by definition of solution process)} \end{aligned}$$

as required □

b. *Strongly Locally Complete* ⇒ *Locally Complete*

Given

$$\boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l-1) \subseteq \boldsymbol{\mathcal{D}}(l),\ \Psi_l \subseteq \Psi', \forall l = 1..\kappa$$

Required to prove

$$\boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(\kappa)$$

Proof Assume $\boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(l-1)$ then

$$\begin{aligned}
\boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) &= \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) \\
&\subseteq \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) \text{ (since } \Psi_l \subseteq \Psi' \Rightarrow \boldsymbol{\mathcal{C}}(\Psi') \subseteq \boldsymbol{\mathcal{C}}(\Psi_l)) \\
&\subseteq \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l-1) \text{ (induction step)} \\
&\subseteq \boldsymbol{\mathcal{D}}(l) \text{ (strong local completeness)}
\end{aligned}$$

Hence $\boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0) \subseteq \boldsymbol{\mathcal{D}}(\kappa)$, as required □

Locally Complete ⇏ *Strongly Locally Complete*

Proof Consider a solution process with step

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$$

where we include $(v_1, \ldots, v_n) \in \boldsymbol{\mathcal{D}}(k-1)$ in $\boldsymbol{\mathcal{D}}(k)$ iff

$$(v_1, \ldots, v_n) \in \boldsymbol{\mathcal{D}}(k-1) \cap \boldsymbol{\mathcal{C}}(\Psi')$$

However, if

$$\boldsymbol{\mathcal{C}}(\Psi') \subset \boldsymbol{\mathcal{C}}(\Psi_k)$$

then

$$\exists (u_1, \ldots, u_n) | (u_1, \ldots, u_n) \in \boldsymbol{\mathcal{C}}(\Psi'), (u_1, \ldots, u_n) \notin \boldsymbol{\mathcal{C}}(\Psi_k)$$

and by the definition of local completeness, this solution step is not locally complete. However, the solution process itself is obviously locally complete.

□

c. *Locally Complete* ⇔ *Complete*

Trivially true if $\Psi' = \Psi$

d. *Strongly Consistent* ⇒ *Consistent*

Trivial. Given $\mathcal{C}(\Psi) \cap \mathcal{D}(l) \neq \emptyset$, let $l = \kappa$.

Consistent ⇒ *Strongly Consistent*

Given

$$\mathcal{C}(\Psi) \neq \emptyset \Rightarrow \mathcal{C}(\Psi) \cap \mathcal{D}(\kappa) \neq \emptyset$$

Required to prove

$$\mathcal{C}(\Psi) \neq \emptyset \Rightarrow \mathcal{C}(\Psi) \cap \mathcal{D}(l) \neq \emptyset, \forall l = 0..\kappa$$

Proof

$$\mathcal{C}(\Psi) \cap \mathcal{D}(\kappa) \subseteq \mathcal{C}(\Psi) \cap \mathcal{D}(l) \text{ (as } \mathcal{D}(\kappa) \subseteq \mathcal{D}(l))$$

Hence

$$\begin{aligned} \exists x \in \mathcal{C}(\Psi) \cap \mathcal{D}(\kappa) &\Rightarrow x \in \mathcal{C}(\Psi) \cap \mathcal{D}(l) \\ &\Rightarrow \mathcal{C}(\Psi) \cap \mathcal{D}(l) \neq \emptyset \end{aligned}$$

as required □

e. *Strongly Locally Consistent* ⇏ *Locally Consistent*

Proof Consider the problem $\{\{(x, \mathbb{Z})\}, \{x \leq 5, x \geq 1\}\}$ and the solution process

$$\begin{aligned} \mathcal{D}(0) = \mathbb{Z} &\xrightarrow{x \geq 1} \mathcal{D}(1) = \{0, 6\} \\ &\xrightarrow{x \leq 5} \mathcal{D}(2) = \{0, 6\} \end{aligned}$$

This is an acceptable solution process by the definition of section 6.6 and $\mathcal{C}(\Psi) \neq \emptyset$. In fact, $\mathcal{C}(\Psi) = \{1, 2, 3, 4\}$. The process is strongly locally consis-

tent, as

$$\begin{aligned} & (x \geq 1) \cap \boldsymbol{\mathcal{D}}(1) = \{6\} \neq \emptyset \\ \text{and} \quad & (x \leq 5) \cap \boldsymbol{\mathcal{D}}(2) = \{0\} \neq \emptyset \end{aligned}$$

But the process is not locally consistent because

$$(x \geq 1) \cap (x \leq 5) \cap \boldsymbol{\mathcal{D}}(2) = \emptyset$$

□

Locally Consistent ⇒ *Strongly Locally Consistent*

Given

$$\boldsymbol{\mathcal{C}}(\Psi') \neq \emptyset \Rightarrow \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(\kappa) \neq \emptyset$$

Required to prove

$$\boldsymbol{\mathcal{C}}(\Psi_l) \neq \emptyset \Rightarrow \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l) \neq \emptyset, \forall l = 0..\kappa$$

Proof

$$\boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l) \ (\text{as } \boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{D}}(l) \text{ and } \boldsymbol{\mathcal{C}}(\Psi') \subseteq \boldsymbol{\mathcal{C}}(\Psi_l))$$

Hence

$$\begin{aligned} \exists x \in \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(\kappa) & \Rightarrow x \in \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l) \\ & \Rightarrow \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l) \neq \emptyset \end{aligned}$$

as required □

f. *Consistent* ⇔ *Locally Consistent*

True if $\Psi = \Psi'$

g. *Strongly Sound* ⇒ *Sound*

Given

$$\boldsymbol{\mathcal{D}}(l) \subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(l-1), \forall l = 1..\kappa$$

Required to prove

$$\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0)$$

Proof Assume $\boldsymbol{\mathcal{D}}(l-1) \subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0)$ then

$$\begin{aligned}\boldsymbol{\mathcal{D}}(l) &\subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(l-1) \text{ (strongly sound)}\\ &\subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \text{ (induction step)}\\ &\subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0) \text{ (as } S \cap S = S)\end{aligned}$$

Therefore, $\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}(0)$ as required □

Sound $\not\Rightarrow$ *Strongly Sound*

Proof Consider the simple solution process β where a solution step is as follows

$$\boldsymbol{\mathcal{D}}(k-1) \xrightarrow{\Psi_k} \boldsymbol{\mathcal{D}}(k)$$
$$\text{where } \boldsymbol{\mathcal{D}}(k) = \boldsymbol{\mathcal{C}}(\Psi_k) \cap \boldsymbol{\mathcal{D}}(k-1)$$

Such a process is obviously sound as $\cup_k \Psi_k = \Psi$, however it will almost certainly not be strongly sound as $\boldsymbol{\mathcal{C}}(\Psi) \subseteq \boldsymbol{\mathcal{C}}(\Psi_k)$ and if $\boldsymbol{\mathcal{C}}(\Psi)$ is a strict subset of $\boldsymbol{\mathcal{C}}(\Psi_k)$, then

$$\begin{aligned}&\boldsymbol{\mathcal{D}}(k-1) \cap \boldsymbol{\mathcal{C}}(\Psi) \subset \boldsymbol{\mathcal{D}}(k-1) \cap \boldsymbol{\mathcal{C}}(\Psi_k)\\ \Rightarrow \quad &\boldsymbol{\mathcal{D}}(k-1) \cap \boldsymbol{\mathcal{C}}(\Psi) \subset \boldsymbol{\mathcal{D}}(k)\end{aligned}$$

That is, there *will* be values in $\boldsymbol{\mathcal{D}}(k)$ which are not valid solutions in $\boldsymbol{\mathcal{C}}(\Psi)$ and so the solution step is not sound and β is not strongly sound. □

h. *Strongly Locally Sound* $\Rightarrow$ *Locally Sound*

Given

$$\boldsymbol{\mathcal{D}}(l) \subseteq \boldsymbol{\mathcal{C}}(\Psi_l) \cap \boldsymbol{\mathcal{D}}(l-1), \forall l = 1..\kappa$$

Required to prove

$$\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0),\ \Psi' = \cup \Psi_i$$

Proof

$$\begin{aligned}\boldsymbol{\mathcal{D}}(\kappa) &\subseteq \boldsymbol{\mathcal{C}}(\Psi_{\kappa-1}) \cap \boldsymbol{\mathcal{D}}(\kappa-1)\\ &\subseteq \boldsymbol{\mathcal{C}}(\Psi_{\kappa-1}) \cap \boldsymbol{\mathcal{C}}(\Psi_{\kappa-2}) \cap \boldsymbol{\mathcal{D}}(\kappa-2)\\ &\subseteq \cdots\\ &\subseteq \boldsymbol{\mathcal{C}}(\Psi_{\kappa-1}) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi_1) \cap \boldsymbol{\mathcal{D}}(0)\end{aligned}$$

But $\boldsymbol{\mathcal{C}}(\Psi_{\kappa-1}) \cap \boldsymbol{\mathcal{C}}(\Psi_{\kappa-2}) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi_1) = \boldsymbol{\mathcal{C}}(\Psi')$, as $\Psi' = \cup \Psi_i$ and by definition of $\boldsymbol{\mathcal{C}}$.

Hence, $\boldsymbol{\mathcal{D}}(\kappa) \subseteq \boldsymbol{\mathcal{C}}(\Psi') \cap \boldsymbol{\mathcal{D}}(0)$. □

Locally Sound $\not\Rightarrow$ *Strongly Locally Sound*

Proof Consider the following solution process.

Find $(v_1, \ldots, v_n)$ such that $(v_1, \ldots, v_n) \notin \boldsymbol{\mathcal{C}}(\Psi')$ and order $\Psi' = \{\Psi'_1, \ldots, \Psi'_\kappa\}$ such that $(v_1, \ldots, v_n) \notin \boldsymbol{\mathcal{C}}(\Psi' \setminus \Psi'_1)$, ie $(v_1, \ldots, v_n)$ is not a valid solution to the constraint problem. We then define the following solution process:

$$\begin{aligned}\boldsymbol{\mathcal{D}}(0) &\xrightarrow{\Psi'_1} \boldsymbol{\mathcal{D}}(1)\\ &\xrightarrow{\Psi'_2} \boldsymbol{\mathcal{D}}(2)\\ &\longrightarrow^* \cdots\\ &\xrightarrow{\Psi'_\kappa} \boldsymbol{\mathcal{D}}(\kappa)\end{aligned}$$

Where

$$\begin{aligned}\boldsymbol{\mathcal{D}}(1) &= (\boldsymbol{\mathcal{D}}(0) \cap \boldsymbol{\mathcal{C}}(\Psi'_1)) \cup (v_1, \ldots, v_n)\\ \boldsymbol{\mathcal{D}}(i) &= (\boldsymbol{\mathcal{D}}(i-1) \cap \boldsymbol{\mathcal{C}}(\Psi'_i))\,,\ \forall i = 2..\kappa\end{aligned}$$

Such a process is locally sound as

$$
\begin{aligned}
\boldsymbol{\mathcal{D}}(\kappa) &= \boldsymbol{\mathcal{D}}(\kappa-1) \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa) \\
&= \boldsymbol{\mathcal{D}}(\kappa-2) \cap \boldsymbol{\mathcal{C}}(\Psi'_{\kappa-1}) \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa) \\
&= \cdots \\
&= ((\boldsymbol{\mathcal{D}}(0) \cap \boldsymbol{\mathcal{C}}(\Psi'_1)) \cup (v_1, \ldots, v_n)) \cap \boldsymbol{\mathcal{C}}(\Psi'_2) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa) \\
&= (\boldsymbol{\mathcal{D}}(0) \cup (v_1, \ldots, v_n)) \cap (\boldsymbol{\mathcal{C}}(\Psi'_1) \cup (v_1, \ldots, v_n)) \\
&\quad \cap \boldsymbol{\mathcal{C}}(\Psi'_2) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa) \\
&= \boldsymbol{\mathcal{D}}(0) \cap ((\boldsymbol{\mathcal{C}}(\Psi'_1) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa)) \cup \\
&\quad ((v_1, \ldots, v_n) \cap \boldsymbol{\mathcal{C}}(\Psi'_2) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa))) \\
&\qquad\qquad (\text{as } \boldsymbol{\mathcal{D}}(0) \cup (v_1, \ldots, v_n) = \boldsymbol{\mathcal{D}}(0)) \\
&= \boldsymbol{\mathcal{D}}(0) \cap (\boldsymbol{\mathcal{C}}(\Psi') \cup ((v_1, \ldots, v_n) \cap \boldsymbol{\mathcal{C}}(\Psi'_2) \cap \cdots \cap \boldsymbol{\mathcal{C}}(\Psi'_\kappa))) \\
&\qquad\qquad (\text{by definition of } \boldsymbol{\mathcal{C}}) \\
&= \boldsymbol{\mathcal{D}}(0) \cap (\boldsymbol{\mathcal{C}}(\Psi') \cup \emptyset) \;\; (\text{by definition of } (v_1, \ldots, v_n)) \\
&= \boldsymbol{\mathcal{D}}(0) \cap \boldsymbol{\mathcal{C}}(\Psi') \\
&\subseteq \boldsymbol{\mathcal{D}}(0) \cap \boldsymbol{\mathcal{C}}(\Psi')
\end{aligned}
$$

However, the process is not strongly locally sound as $\boldsymbol{\mathcal{D}}(1)$ contains a value not in $\boldsymbol{\mathcal{C}}(\Psi'_1) \cap \boldsymbol{\mathcal{D}}(0)$. □

i. *Locally Sound* ⇔ *Sound*

Trivially true if $\Psi' = \Psi$.

□

Appendix D

Enhanced solution spaces

This appendix defines enhanced and embedded solution spaces. Enhanced solution spaces allow discussion of a solution space within a larger solution space. Embedded solution spaces allow discussion of subsets of solution spaces.

Definition D.1 (Enhancement of solution spaces) *Given two sets of entities Φ and Φ' with solution spaces $\boldsymbol{\mathcal{D}}_{\Phi}(0)$ and $\boldsymbol{\mathcal{D}}_{\Phi'}(0)$ respectively, the* **enhancement of $\boldsymbol{\mathcal{D}}_{\Phi}(0)$ with respect to Φ'** *is the solution space of Φ within the solution space $\Phi \cup \Phi'$. This is written*

$$\begin{aligned}\boldsymbol{\mathcal{D}}_{\Phi}\big|^{\Phi\cup\Phi'}(0) &= \boldsymbol{\mathcal{D}}_{\Phi}(0) \times \boldsymbol{\mathcal{D}}_{\Phi'\setminus\Phi}(0), \\ \boldsymbol{\mathcal{D}}_{\Phi}\big|^{\Phi\cup\Phi'}(k) &= \boldsymbol{\mathcal{D}}_{\Phi}(k) \times \boldsymbol{\mathcal{D}}_{\Phi'\setminus\Phi}(0).\end{aligned}$$

Since it is natural to want

$$\boldsymbol{\mathcal{D}}_{\Phi}\big|^{\Phi\cup\Phi'}(0) = \boldsymbol{\mathcal{D}}_{\Phi'}\big|^{\Phi\cup\Phi'}(0),$$

an ordering, $<$, is enforced on the labels in $\Phi \cup \Phi'$ so that

$$\begin{aligned}\boldsymbol{\mathcal{D}}_{\Phi}\big|^{\Phi\cup\Phi'}(0) &= \big(\boldsymbol{\mathcal{D}}_{\Phi}(0) \times \boldsymbol{\mathcal{D}}_{\Phi'\setminus\Phi}(0)\big)_{<} \\ &= \big(\boldsymbol{\mathcal{D}}_{\Phi'}(0) \times \boldsymbol{\mathcal{D}}_{\Phi\setminus\Phi'}(0)\big)_{<} \\ &= \boldsymbol{\mathcal{D}}_{\Phi'}\big|^{\Phi\cup\Phi'}(0).\end{aligned}$$

In this thesis the ordering is implied unless explicitly stated. It is omitted for clarity.

Conversely, it is sometimes necessary to take the enhanced solution space of a superset Φ and examine the solution space of a subset Φ' of Φ. This is called **the embedded solution space of Φ' in Φ** *and is equivalent to the embedded domains*

in [84]. The embedded solution space of Φ' in Φ is denoted $\boldsymbol{\mathcal{D}}_\Phi|_{\Phi'}(0)$ as below

$$\boldsymbol{\mathcal{D}}_\Phi|_{\Phi'}(0) \;=\; \bigotimes\{\boldsymbol{\mathcal{D}}_\phi(0)|\phi \in \Phi'\}\,.$$

□

For relational algebra, the enhanced solution space is equivalent to a natural join. The embedded solution space is similar to a projection. The ordering is similar to the notion of an *attribute*. Enhanced solution spaces are also extensions of Monfroy's structure embedding and constraint system enrichment [84].

Example D.1 Let $\Phi = \{(x, D_x), (y, D_y), (z, D_z)\}$ and $\Phi' = \{(y, D_y), (z, D_z), (w, D_w), (a, D_a)\}$, with a lexicographic ordering $<_{lex} = a < w < x < y < z$. Then

$$\begin{aligned}
\boldsymbol{\mathcal{D}}_\Phi(0) &= D_x \times D_y \times D_z, \\
\boldsymbol{\mathcal{D}}_{\Phi'}(0) &= D_y \times D_z \times D_w \times D_a \\
&= D_a \times D_w \times D_y \times D_z, \text{ under } <_{lex}.
\end{aligned}$$

Consequently,

$$\begin{aligned}
\boldsymbol{\mathcal{D}}_\Phi|^{\Phi\cup\Phi'}(0) &= D_x \times D_y \times D_z \times D_w \times D_a \\
&= D_a \times D_w \times D_x \times D_y \times D_z, \text{ under } <_{lex}, \\
\boldsymbol{\mathcal{D}}_{\Phi'}|^{\Phi\cup\Phi'}(0) &= D_a \times D_w \times D_y \times D_z \times D_x \\
&= D_a \times D_w \times D_x \times D_y \times D_z, \text{ under } <_{lex}.
\end{aligned}$$

□

A stated aim of Monfroy's enrichments is that no solutions should be lost. However, Monfroy treats constraint solvers as black boxes and does not consider solution spaces at all. The terminal solution space produced by a solver may contain any number of solutions. However, all solutions that it *does* contain should be preserved. Consequently, the enhanced solution space within a solution process is defined as follows.

Definition D.2 (Enhancement of solution spaces) *Given two constraint pro-*

cesses α and β such that for constraint problems $P_1 = (\Phi, \Psi)$ and $P_2 = (\Phi', \Psi')$,

$$\begin{array}{rcl} \mathcal{D}_{\Phi}(0) & \stackrel{\Psi}{\longrightarrow}^{*}_{\alpha} & \mathcal{D}_{\Phi}(k) \\ \mathcal{D}_{\Phi'}(0) & \stackrel{\Psi'}{\longrightarrow}^{*}_{\beta} & \mathcal{D}_{\Phi'}(l), \end{array}$$

the **enhanced solution space at step k in process α and step l in process** *β is:*

$$\mathcal{D}_{\Phi}|^{\Phi \cup \Phi'}(k,l) = \left(\mathcal{D}_{\Phi}(k) \times \mathcal{D}_{\Phi' \setminus \Phi}(0)\right) \cap \left(\mathcal{D}_{\Phi \setminus \Phi'}(0) \times \mathcal{D}_{\Phi'}(l)\right).$$

□

Example D.2 Let $\Phi = \{(w, \mathbb{R}), (x, \mathbb{R}), (y, \mathbb{R})\}$ with solution spaces

$$\begin{array}{rcl} \mathcal{D}_{\Phi}(0) & = & \mathbb{R} \times \mathbb{R} \times \mathbb{R}, \\ \mathcal{D}_{\Phi}(10) & = & \{\{w = \{1\}, x = \{2\}, z = \{3\}\}, \{w = \{3\}, x = \{4\}, z = \{5\}\}, \\ & & \{w = \{5\}, x = \{6\}, z = \{7\}\}\}, \end{array}$$

and $\Phi' = \{(x, \mathbb{R}), (y, \mathbb{R}), (z, \mathbb{R})\}$ with solution spaces

$$\begin{array}{rcl} \mathcal{D}_{\Phi'}(0) & = & \mathbb{R} \times \mathbb{R} \times \mathbb{R}, \\ \mathcal{D}_{\Phi'}(11) & = & \{\{x = \{1\}, y = \{2\}, z = \{3\}\}, \{x = \{2\}, y = \{3\}, z = \{4\}\}, \\ & & \{x = \{3\}, y = \{4\}, z = \{5\}\}, \{x = \{4\}, y = \{5\}, z = \{6\}\}\}, \end{array}$$

with lexicographic ordering $<_{lex} = w < x < y < z$. Then

$$\begin{array}{rcl} \mathcal{D}_{\Phi}|^{\Phi \cup \Phi'}(10,11) & = & \left(\mathcal{D}_{\Phi}(10) \times \mathcal{D}_{\Phi' \setminus \Phi}(0)\right) \cap \left(\mathcal{D}_{\Phi \setminus \Phi'}(0) \times \mathcal{D}_{\Phi'}(11)\right) \\ & = & (\{\{w = \{1\}, x = \{2\}, y = \{3\}\}, \{w = \{3\}, x = \{4\}, y = \{5\}\}, \\ & & \{w = \{5\}, x = \{6\}, y = \{7\}\}\} \times \mathbb{R}) \cap \\ & & (\mathbb{R} \times \{\{x = \{1\}, y = \{2\}, z = \{3\}\}, \\ & & \{x = \{2\}, y = \{3\}, z = \{4\}\}, \{x = \{3\}, y = \{4\}, z = \{5\}\}, \\ & & \{x = \{4\}, y = \{5\}, z = \{6\}\}\}) \\ & = & \{\{w = \{1\}, x = \{2\}, y = \{3\}, z = \{4\}\}, \\ & & \{w = \{3\}, x = \{4\}, y = \{5\}, z = \{6\}\}\}. \end{array}$$

Thus $\boldsymbol{\mathcal{D}}_{\Phi}|^{\Phi\cup\Phi'}(10,11)$ only contains configurations that are possible solutions to the whole problem. For example, $\{1,2,3,5\}$ is only a solution to Φ and not Φ' and cannot be a solution to the whole problem. $\square$

Appendix E

Paradigms of collaboration

Monfroy [84] has presented three paradigms of collaboration for hybrid constraint solvers. Section 7.4.1 has discussed the serial collaboration paradigm in detail. This appendix discusses the parallel and concurrent paradigms in terms of the constraint satisfaction framework. The advantages and disadvantages of each paradigm are discussed, particularly in terms of practical issues.

E.1 Parallel hybrids

The second collaboration paradigm suggested by Monfroy is parallel collaboration. In terms of the framework of chapter 6, parallel solvers work in a slightly different fashion to sequential solvers. A parallel hybrid splits a large constraint problem into a number of subproblems and then subsolvers solve the subproblems independently of all other subproblems in parallel. The solution spaces found by the subsolvers are combined into a solution space for the problem as a whole using a combination function.

In diagrammatical form (see figure E.1) constraint problem $P = (\Phi, \Psi)$ is split into subproblems $P_1 = (\Phi_1, \Psi_1), P_2 = (\Phi_2, \Psi_2), \ldots, P_n = (\Phi_n, \Psi_n)$ to be solved by solvers $(S_1, S_2, \ldots, S_n)$. For the purposes of this section, the decomposition of P to the various P_i is assumed. Entities Φ_i are found by taking the imposed sets $\xi(\Psi_i)$. The initial solution space $\boldsymbol{\mathcal{D}}(0)$ of P is used as input to the subsolvers. However, the solvers are each only concerned with the embedded solution spaces $\boldsymbol{\mathcal{D}}_\Phi|_{\Phi_i}(0)$. Each solver then initiates the solution process

$$\boldsymbol{\mathcal{D}}_\Phi|_{\Phi_i}(0) \xrightarrow{\Psi_i}_{S_i} \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_i}(\kappa) .$$

The combination function, f, combines the terminal solution spaces of the subproblems into a solution space for P. Choosing the subproblems P_i is critical for an efficient hybrid. The combination function is also vital to determining the terminal solution space of the hybrid and also to the efficiency of the hybrid. Monfroy does not discuss the practicalities of a parallel hybrid, but it seems likely that the combination function will have complexity worse than linear in the number of constraints, thus destroying the advantages of using hybrids, as the combination function may have to find the intersection of two infinite, implicitly defined sets. Effectively this means solving another nonlinear system of equations. If there are only a finite number of solutions to each subproblem, then the recombination function will be of linear complexity in the number of solutions.

A theorem similar to theorem 6.1 can be constructed for parallel constraint satisfaction.

Theorem E.1 (Relation between local and global properties for parallel collaboration)

Let solution process S act on problem $P = (\Phi, \Psi)$ such that

$$\boldsymbol{\mathcal{D}}_\Phi(0) \stackrel{\Psi}{\longrightarrow}{}^*_S \boldsymbol{\mathcal{D}}_\Phi(\kappa).$$

Let S be a parallel collaboration between two solvers S_1 and S_2 such that, for $\Phi_1, \Phi_2 \subseteq \Phi$ and $\Psi_1, \Psi_2 \subseteq \Psi$,

$$\begin{array}{rcl} \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}(0) & \stackrel{\Psi_1}{\longrightarrow}{}^*_{S_1} & \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}(\kappa_1), \\ \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}(0) & \stackrel{\Psi_2}{\longrightarrow}{}^*_{S_2} & \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}(\kappa_2), \\ \boldsymbol{\mathcal{D}}_\Phi(\kappa) & = & f(\boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}(\kappa_1), \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}(\kappa_2)), \end{array}$$

where f is a combination function.

Then

1. S is (globally) sound if S_1, S_2 and f are locally sound.

2. S is (globally) complete if S_1, S_2 and f are locally complete.

□

Proof The proof of this theorem is presented after lemma E.1. □

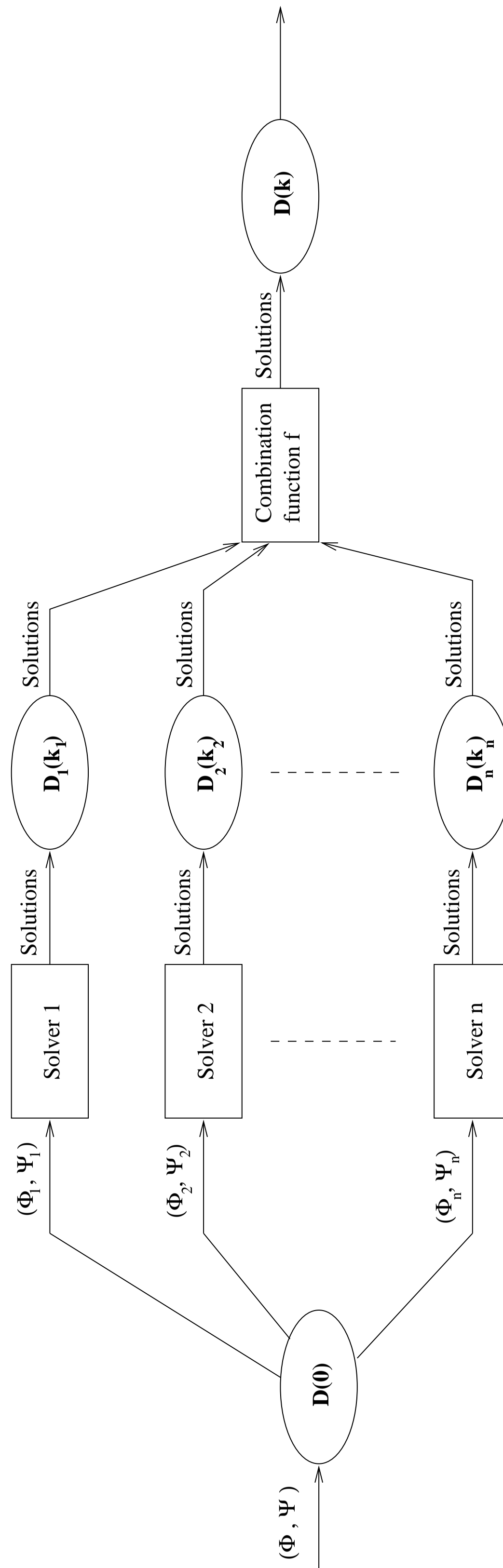


Figure E.1: Parallel Collaboration

The reader is reminded that a solution process

$$\boldsymbol{\mathcal{D}}(0) \xrightarrow{\Psi}{}^{*} \boldsymbol{\mathcal{D}}(\kappa)$$

is *consistent* if $\boldsymbol{\mathcal{D}}(\kappa)$ contains *a* solution to the constraint problem formed by the set of constraints Ψ. The process is *sound* if $\boldsymbol{\mathcal{D}}(\kappa)$ contains *only* solutions to the constraint problem and the process is *complete* if it contains *all* of the solutions to the constraint problem. For parallel collaboration, the combination function f satisfies these properties if it preserves the properties of the solution spaces it acts on.

By theorem E.1, if the terminal solution spaces provided by processes S_1, S_2 and combination function f contain *only* solutions to the subproblems then S contains *only* solutions to the whole problem. Similarly, if the terminal solution spaces provided by processes S_1, S_2 and combination function f contain *all* of the solutions to the subproblems, then S contains *all* of the solutions to the whole problem.

Notice that, unlike the sequential theorem, theorem 6.1, global consistency is not decided by this theorem. Even if S_1 and S_2 are globally consistent, because they are operating independently of each other it is possible that both solvers could find a single globally consistent solution, but that both find a different one. Consequently, S is not globally consistent. Building a globally consistent parallel hybrid will be difficult.

Lemma E.1 $A \subseteq B, C \subseteq D \Rightarrow A \cap C \subseteq B \cap D$. □

Proof $A \subseteq B \Rightarrow x \in A \Rightarrow x \in B$.

Similarly,

$$C \subseteq D \Rightarrow x \in C \Rightarrow x \in D.$$

Therefore

$$\begin{aligned} x \in A \cap C &\Rightarrow x \in A \wedge x \in C \\ &\Rightarrow x \in B \wedge x \in D \\ &\Rightarrow x \in B \cap D. \end{aligned}$$

□ **Proof (of theorem E.1)**

1. S_1, S_2 and f sound $\Rightarrow$ S sound.

If S_1 is sound then

$$\boldsymbol{\mathcal{C}}(\Psi_1) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}(0) \supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}(\kappa_1).$$

Similarly, S_2 and f sound implies that

$$\begin{aligned} \boldsymbol{\mathcal{C}}(\Psi_2) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}(0) &\supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}(\kappa_2), \\ \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2) &\supseteq \boldsymbol{\mathcal{D}}_\Phi(\kappa), \end{aligned}$$

where

$$\boldsymbol{\mathcal{D}}_\Phi|_{\Phi'}^{\Phi}(k) = (\boldsymbol{\mathcal{D}}_\Phi|_{\Phi'}(k))|^{\Phi}.$$

Given these, it remains to prove that S is sound, i.e.

$$\boldsymbol{\mathcal{C}}(\Psi) \cap \boldsymbol{\mathcal{D}}_\Phi(0) \supseteq \boldsymbol{\mathcal{D}}_\Phi(\kappa).$$

By the definition of enhanced solution spaces

$$\begin{aligned} \boldsymbol{\mathcal{C}}(\Psi_1) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}^{\Phi}(0) &\supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1), \\ \boldsymbol{\mathcal{C}}(\Psi_2) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}^{\Phi}(0) &\supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2). \end{aligned}$$

Also by the definitions of enhanced and embedded solution spaces,

$$\boldsymbol{\mathcal{D}}_\Phi|_{\Phi_i}^{\Phi}(0) = \boldsymbol{\mathcal{D}}_\Phi(0), \quad i = 1, 2.$$

Consequently,

$$\begin{aligned} \boldsymbol{\mathcal{C}}(\Psi_1) \cap \boldsymbol{\mathcal{D}}_\Phi(0) \supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1), \\ \boldsymbol{\mathcal{C}}(\Psi_2) \cap \boldsymbol{\mathcal{D}}_\Phi(0) \supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2). \end{aligned}$$

Then

$$\begin{aligned} \boldsymbol{\mathcal{C}}(\Psi) &= \boldsymbol{\mathcal{C}}(\Psi_1) \cap \boldsymbol{\mathcal{C}}(\Psi_2) \cap \boldsymbol{\mathcal{D}}_\Phi(0) \\ &\supseteq \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1) \cap \boldsymbol{\mathcal{D}}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2) \text{ (By lemma E.1 and commutativity of } \cap) \\ &\supseteq \boldsymbol{\mathcal{D}}_\Phi(\kappa) \text{ (By combination function } f). \end{aligned}$$

2. S_1, S_2 and f complete $\Rightarrow$ S complete.

If S_1 is complete then

$$\mathcal{C}(\Psi_1) \cap \mathcal{D}_\Phi|_{\Phi_1}(0) \subseteq \mathcal{D}_\Phi|_{\Phi_1}(\kappa_1).$$

Similarly, S_2 and f complete implies that

$$\begin{aligned} \mathcal{C}(\Psi_2) \cap \mathcal{D}_\Phi|_{\Phi_2}(0) &\subseteq \mathcal{D}_\Phi|_{\Phi_2}(\kappa_2), \\ \mathcal{D}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1) \cap \mathcal{D}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2) &\subseteq \mathcal{D}_\Phi(\kappa). \end{aligned}$$

Given these, it remains to prove that S is complete, i.e.

$$\mathcal{C}(\Psi) \cap \mathcal{D}_\Phi(0) \subseteq \mathcal{D}_\Phi(\kappa).$$

By the definition of enhanced solution spaces

$$\begin{aligned} \mathcal{C}(\Psi_1) \cap \mathcal{D}_\Phi|_{\Phi_1}^{\Phi}(0) &\subseteq \mathcal{D}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1), \\ \mathcal{C}(\Psi_2) \cap \mathcal{D}_\Phi|_{\Phi_2}^{\Phi}(0) &\subseteq \mathcal{D}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2). \end{aligned}$$

Consequently,

$$\begin{aligned} \mathcal{C}(\Psi_1) \cap \mathcal{D}_\Phi(0) \subseteq \mathcal{D}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1), \\ \mathcal{C}(\Psi_2) \cap \mathcal{D}_\Phi(0) \subseteq \mathcal{D}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2). \end{aligned}$$

Then

$$\begin{aligned} \mathcal{C}(\Psi) &= \mathcal{C}(\Psi_1) \cap \mathcal{C}(\Psi_2) \cap \mathcal{D}_\Phi(0) \\ &\subseteq \mathcal{D}_\Phi|_{\Phi_1}^{\Phi}(\kappa_1) \cap \mathcal{D}_\Phi|_{\Phi_2}^{\Phi}(\kappa_2) \text{ (By lemma E.1 and commutativity of } \cap) \\ &\subseteq \mathcal{D}_\Phi(\kappa) \text{ (By combination function } f). \end{aligned}$$

□

Parallel solution does not suffer from the same problems as sequential satisfaction. Since the output of the various solvers is combined using the function f, it does not matter whether inputs and outputs of solvers are compatible - only that f can handle the various types of output.

Unfortunately, this means that the operation of f is critical. The most powerful combination function is simply the intersection of the various solution spaces. Intersection is sound and complete. However, the intersection of infinite sets is a

non-trivial problem .

Another possible combination function is to find a single member of the intersection of solution spaces. This would be a solution to the overall problem, P.

The combination function is a constraint solver in itself. It takes as input two embedded solution spaces and tries to produce a solution space that is an intersection of these two solution spaces. The two solution spaces provided as input can be thought of as constraints and so the combination function is trying to find solutions to two complicated constraints. Consequently, the operation of the combination function is vital to the operation of the parallel hybrid. The construction of a combination function is very hard in general.

The chief advantage of a parallel hybrid is that it is trivial to implement on a parallel processor. Since the subproblems are solved independently, they can be solved on separate processors and then recombined as necessary. Consequently, if the combination function is fast and efficient, then the hybrid should be fast and efficient.

Parallel collaboration should be considered if:

1. Use of parallel processors to improve performance is possible and important.
2. Subproblems are expected to be relatively independent of each other.
3. An efficient combination function is available.

The following section presents an example of a parallel collaboration.

E.1.1 An example of solvers in parallel

When joining two solvers in parallel, the most important issues to consider are the intersection of the entities of the two problems and the nature of the recombination function. The intersection of the two sets of entities gives an indication as to how difficult it will be to find solutions to the general problem. The recombination function, as discussed in section E.1 is critical for identifying solutions to the general problem.

Consider the geometric problem $P_{geom} = (\Phi_{geom}, \Psi_{geom})$ described in figure E.2,

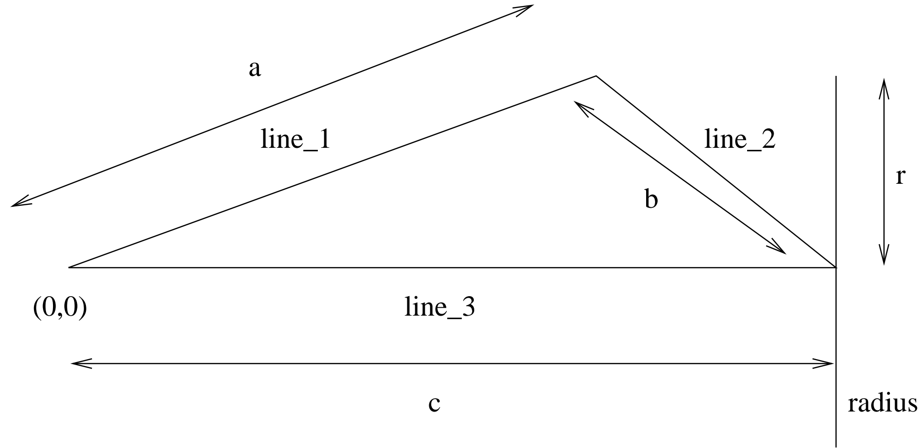


Figure E.2: A simplification of the internal combustion engine

defined by the constraint problem below

$$
\begin{aligned}
\Phi_{geom} \;=\; & \{(line_1_end_1, (0,0,2)), (line_1_end_2, (0,0,2), \\
& (line_2_end_1, (0,0,2)), (line_2_end_2, (0,0,2)), \\
& (line_3_end_1, (0,0,2)), (line_3_end_2, (0,0,2)) \\
& (radius_end_1, (0,0,2)), (radius_end_2, (0,0,2)), \\
& (a, \mathbb{R}), (b, \mathbb{R}), (c, \mathbb{R}), (r, \mathbb{R})\}, \\
\Psi_{geom} \;=\; & \{line_1_end_1 = (0,0), line_1_end_2 = line_2_end_1, \\
& line_2_end_2 = line_3_end_1, line_3_end_2 = line_1_end_1, \\
& midpoint(radius_end_1, radius_end_2) = line_2_end_2, \\
& distance(line_1_end_1, line_1_end_2) = a, \\
& distance(line_2_end_1, line_2_end_2) = b, \\
& distance(line_3_end_1, line_3_end_2) = c, \\
& distance(radius_end_1, radius_end_2) = 2r, b = 10, c = 15\},
\end{aligned}
$$

where Φ_{geom} is a set of entities and Ψ_{geom} is a set of constraints.

Simple geometric reasoning, by considering the furthest possible points on $line_1$, determines that

$$
\begin{aligned}
a &\leq c + b, \\
a &\geq c - b,
\end{aligned}
$$

which in this case corresponds to a range of values for a such that $5 \leq a \leq 25$. There

is no restriction on the size of r from the geometric problem. Correspondingly, there are an infinite number of solutions to this problem corresponding to the line *line_2* rotating about the point (15,0) and *line_1* stretching so that it remains coincident. The length of *radius* is undetermined.

Consider also the algebraic problem, $P_{alg} = (\Phi_{alg}, \Psi_{alg})$ with

$$\begin{aligned}
\Phi_{alg} &= \{(p, \mathbb{R}), (\alpha, \mathbb{R}), (d, \mathbb{R}), (cr, \mathbb{R}), (r, \mathbb{R}), (t, \mathbb{R}), (\pi, \mathbb{R}), (n, \mathbb{R})\}, \\
\Psi_{alg} &= \{p = \alpha.d.cr.r, cr = \frac{2\alpha + t}{t}, d = 2\alpha\pi r^2 n, 1 < \alpha < 3, 4 < n < 12, \\
&\quad p = 1, d = 1, t = 1\},
\end{aligned}$$

where Φ_{alg} is a set of entities and Ψ_{alg} is a set of constraints.

Solving this problem results in values for cr, r and a, as below

$$\begin{aligned}
cr &= \frac{2\alpha + t}{t} \\
&= 2\alpha + 1 \\
&\Rightarrow 3 < cr < 7. \\
1 &= \alpha.1.cr.r \\
&= \alpha.cr.r \\
&\Rightarrow \frac{1}{3} > r > \frac{1}{21}. \\
1 &= 2\pi r^2 an \\
&\Rightarrow \frac{3}{8\pi} < a < \frac{441}{8\pi}.
\end{aligned}$$

Now consider the general problem, $P = (\Phi, \Psi), \Phi = \Phi_{geom} \cup \Phi_{alg}, \Psi = \Psi_{geom} \cup \Psi_{alg}$. An interval solver such as ILOG Solver [92] can solve P but only by converting the geometric constraints in P_{geom} into a system of equations, thus losing any geometric knowledge implicit in the problem.

However, the problem can be solved using domain specific knowledge in a more efficient manner. If P_{geom} is studied using ICBSM or IGCS, then the range of solutions found will be

$$5 \leq a \leq 25,$$
$$r > 0.$$

The domain specific knowledge used here is that the rigid bodies can only move according to their allowable motions. Consequently, the solutions to the problem

consist of all the allowable motions of the rigid bodies.

If P_{alg} is solved in parallel, independently of P_{geom}, then the solutions to P_{alg} are

$$3 < cr < 7,$$
$$\frac{1}{21} < r < \frac{1}{3},$$
$$\frac{3}{8\pi} < a < \frac{441}{8\pi}.$$

The recombination function f is then used to find solutions to P by finding the intersection of the solutions to P_{geom} and P_{alg}

$$3 < cr < 7,$$
$$\frac{1}{21} < r < \frac{1}{3},$$
$$5 \leq a < \frac{441}{8\pi}.$$

Parallel solution of P is very efficient and makes good use of domain specific knowledge.

The decomposition strategy used in this case study is to split $P = (\Phi, \Psi)$ into $P_1 = P_{geom}$ and $P_2 = P_{alg}$. The hybrid can then be described in the solver collaboration language of appendix F as

$$((IGCS, P_{geom})||(ILOGSolver, P_{alg}))\,.$$

E.2 Concurrent hybrids

The third collaboration primitive that Monfroy suggests is concurrency. A concurrent hybrid takes as input a constraint problem $P = (\Phi, \Psi)$. The hybrid has a number of solvers at its disposal as well as a choice function ψ. The hybrid solves problem P simultaneously on all of the subsolvers. The choice function ψ is then used to decide at run-time which solver and terminal solution space to use. For example, a choice function may be to take the solution space of the first subsolver that terminates or the most complete solution space, which will be the largest if all solvers are sound.

In diagrammatical form (see figure E.3) constraint problem $P = (\Phi, \Psi)$ is copied to subsolvers $S_1, \ldots, S_n$. The subsolvers are initiated on P. The choice function is then used to decide when to terminate and what solution space is the output of the hybrid.

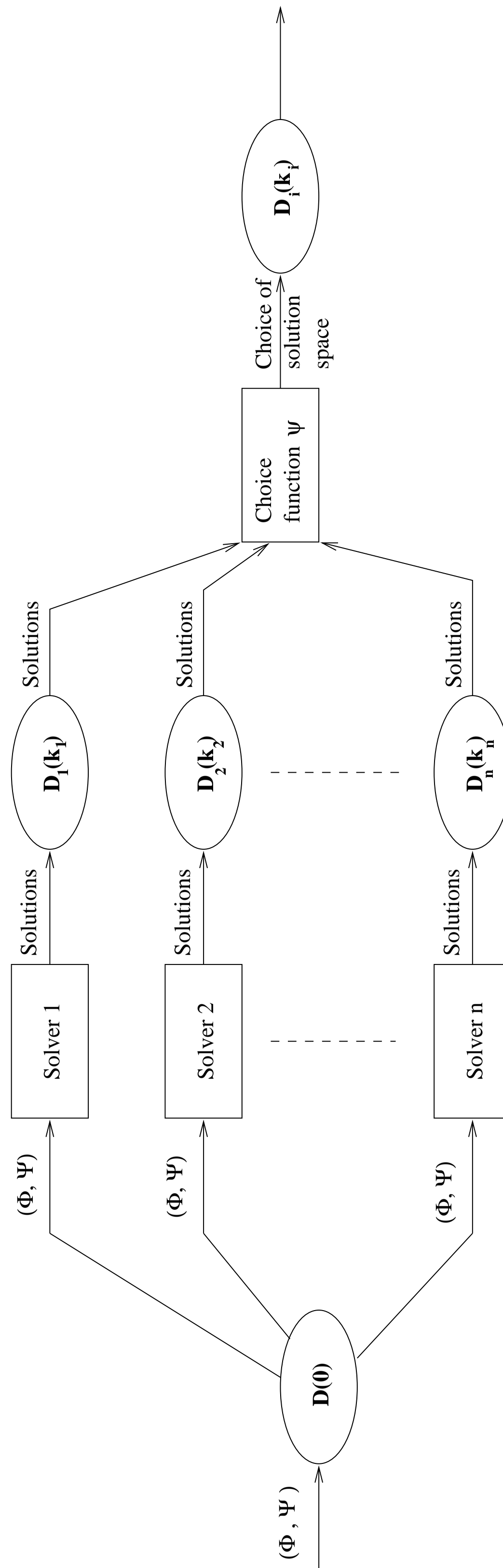


Figure E.3: Concurrent Collaboration

Similar to the parallel hybrid, the choice function ψ is critical to the terminal solution space of the hybrid. However, in this case the ψ function only selects a solution space from amongst n possible solution spaces. Indeed, since ψ is also time dependent, in that it may wish to choose the *first* solver to terminate with a certain property, it is not possible to say with certainty what properties the terminal solution space may have, unless the ψ function enforces a property such as consistency or completeness. Enforcing such a property will mean that the choice function becomes a constraint solver itself and this will be difficult to implement.

The advantages of concurrent hybrids are that they can guarantee certain properties of the terminal solution space, by judicious use of the ψ function, and that, since the solvers act independently, the concurrent subsolvers can be run in parallel. If a choice function is selected that selects the first solver to finish and solvers are run in parallel, then the concurrent collaboration will reduce the real time needed to solve a problem. Note that if knowledge of the problem can be used to determine the best solver to be used before solution commences, this sort of concurrent collaboration is wasted.

However, concurrent hybrids involve massive duplication of effort. Since effectively each solver is trying to do the same job as every other solver, that is find solutions to P, then they will find the same solutions and duplicate the effort.

Concurrent hybrids also do not improve the expressiveness of individual solvers. Since concurrent solvers are only dealing with the same problems, it is not possible to improve the expressiveness of an individual solver using the concurrent paradigm, unlike sequential and parallel hybrids.

Currently no concurrent hybrids exist in the literature.

Concurrent collaboration should be considered if:

1. It is unknown which of many solvers will work best.

2. A particular property of a solution space is desired.

3. Speed is important and there is spare parallel processing power.

Appendix F

Solver collaboration language

The extended solver collaboration language used in this thesis is presented in this appendix. The extended solver collaboration language is based on Monfroy's solver collaboration language [84] but has been extended to allow subsolvers to act on subproblems and also includes a conditional branch statement to initiate backup solvers when a first attempt fails.

The extended solver collaboration language is presented in table F.1. The language is discussed in more detail below.

The set Id of identifiers is used to name solver collaborations and identify them. The symbol S denotes a set of constraint solvers. Note that Monfroy uses only a single solver in each basic part of a collaboration. It is more suitable here to have a number of constraint solvers as a decomposition strategy may identify a number of possible constraint solvers that can be applied to a subproblem.

ψ represents a choice function used in a concurrent collaboration to select from

$$
\begin{array}{rcl}
Id & \in & \text{Identifiers}, \\
S & \subseteq & \text{Solvers}, \\
\psi & \in & \text{Conditional selection}, \\
P & \in & \text{Constraint problems}, \\
\boldsymbol{\mathcal{D}} & \in & \text{Solution spaces}, \\
Col & ::= & Id = E, \\
E & ::= & \diamond | Id | B | E_1 ; E_2 | EP | \text{ repeat } (E) | \psi(EC) | \text{ if } T \text{ then } E_1 \text{ else } E_2, \\
T & ::= & E = \boldsymbol{\mathcal{D}}, \\
B & ::= & (P,S) | (P,S) \parallel B, \\
EP & ::= & E | E \parallel EP, \\
EC & ::= & E | E ? EC.
\end{array}
$$

Table F.1: Solver collaboration language (adapted from BALI [84])

a number of solution spaces as described in appendix E.2. The symbol P represents a constraint problem. In the terms of solver collaboration, each P will typically be a subproblem of a larger constraint problem. Consistent with the rest of this thesis, solution spaces form an important part of solver collaboration and are shown as a $\mathcal{D}$.

The remaining terms in table F.1 define how the basic symbols can be formed into solver collaborations. A solver collaboration is described in the solver collaboration language as an *expression*, E. An expression is assigned a name using the form '$Id = E$'. An expression consists of one of the following:

- The identity solver $\diamond$ which does not refine a constraint problem at all.
- An identifier indicating a previously defined solver collaboration.
- A parallel collaboration. A parallel collaboration consists of a sequence of constraint problem-solver pairs or more complicated expressions, linked with a '$\|$' symbol. This is interpreted that each constraint problem-solver pair is solved independently and the solution spaces combined, as described in section E.1. Thus in the parallel collaboration

  $$(P_1, S_1) \parallel (P_2, S_2) \parallel (P_3, S_3),$$

  problem P_1 is solved by a solver in S_1; problem P_2 is solved by a solver in S_2 and problem P_3 is solved by a solver in S_3 independently of each other. Solutions are then recombined to give a solution space for the combined constraint problem formed by $P_1 \cup P_2 \cup P_3$.
- A serial collaboration. A serial collaboration consists of a sequence of constraint problem-solver pairs or more complicated expressions linked with a ';' symbol. This is interpreted that each constraint problem-solver pair is solved in sequence and that information from the first solution is used to solve the second and so on as described in section 7.4.1. Thus, in the serial collaboration

  $$(P_1, S_1); (P_2, S_2); (P_3, S_3),$$

  problem P_1 is solved by a solver in S_1; problem P_2 is solved by a solver in S_2 and problem P_3 is solved by a solver in S_3. P_1 is solved first and information obtained from this is used in solving P_2 and so on. Solutions to P_3 are then

combined with the solutions obtained from P_1 and P_2 to give solutions to $P_1 \cup P_2 \cup P_3$.

- Repeated application of a solver collaboration. Sometimes information can be obtained during a solution process that can be used to refine the solution even further. However, this information is only apparent after the solution process has completed. Repeated application of a solver collaboration, $Repeat(E)$, can be used to force a solver to repeat a solution process until it can progress no further. Monfroy demonstrates this in [84] by defining the solver collaboration

$$S_{inc} = Maple_NF; Repeat(Lin_{Eq}).$$

  $Maple_NF$ transforms polynomials so they can be used by Lin_{Eq}. Lin_{Eq} is a solver for linear equalities and inequations using an extension of Gaussian elimination. It is repeatedly applied until it cannot simplify the linear equations any more.

- A concurrent collaboration. A concurrent collaboration consists of a number of constraint problem-solver pairs or more complex expressions linked by a '?' symbol and selected from using a choice function ψ. This is equivalent to the concurrent collaboration presented in section E.2. Thus,

$$\psi_f\left((P_1, S_1)?(P_1, S_2)?(P_1, S_3)\right)$$

  is a concurrent collaboration where P_1 is solved by one of S_1, one of S_2 and one of S_3 concurrently. The choice function ψ_f selects the concurrent solution space given by the first solver to terminate.

- A conditional branch. The conditional branch statement has been included in the language to allow the possibility that a constraint solver expression may produce a value different than expected. Should this occur, the user of the constraint solver may wish to activate backup solvers.

  For example, suppose that two solvers S_1 and S_2 can be applied to find solutions to problem P. Solver S_1 is much more efficient than S_2 but can miss solutions occassionally, whereas solver S_2 is much more thorough but considerably slower. In this case, the user may wish to build a hybrid constraint

solver as below

$$\mathcal{S} = [\text{ if } (S_1, P) = \emptyset \text{ then } (S_2, P) \text{ else } \diamond]\,,$$

where $\diamond$ is the identity solver $\diamond(P, \boldsymbol{\mathcal{D}}) = \boldsymbol{\mathcal{D}}$.

The hybrid solver $\mathcal{S}$ will attempt to solve problem P using solver S_1. However, if no solutions are found, then S_2 is used to find solutions to P. Note that several assumptions are made in the definition of $\mathcal{S}$:

1. S_1 is sound. If S_1 is not sound then it may return a solution space which is not identically empty but also does not contain any solutions to P. If S_1 is sound and returns an empty set, then S_1 has failed to find any solutions and S_2 should be used.
2. Finding a single solution to P is sufficient. The conditional test does not count the number of solutions in order to activate S_2. It is possible to imagine a conditional branch statement that is predicated by the number of elements in the solution space, and in particular by the number of *solutions* in the terminal solution space of solver S_1. However, such a test would likely be extremely expensive, as estimating the number of solutions to even simple constraint problems is difficult.

Note that, other than the conditional guard operator which has been omitted for clarity, the language in table F.1 contains Monfroy's BALI.

The operational semantics of the statements in table F.1 are straight-forward to convert from Monfroy's descriptions into the terminology of chapters 4- 6. The conditional branch is described below as an example and because it is not part of Monfroy's basic language.

Definition F.1 (Conditional branch) *Successful branch:*

$$\frac{\boldsymbol{\mathcal{D}}(k) \stackrel{E_0}{\longrightarrow}{}^* \boldsymbol{\mathcal{D}}(k+1), \quad \boldsymbol{\mathcal{D}}(k+1) = \boldsymbol{\mathcal{D}}}{P : (\ \mathit{if}\ E_0 = \boldsymbol{\mathcal{D}}\ \mathit{then}\ E_1\ \mathit{else}\ E_2; E, \boldsymbol{\mathcal{D}}(k)) \longrightarrow P.\,(E_1, \boldsymbol{\mathcal{D}}(k)) : (E_1; E, \boldsymbol{\mathcal{D}}(k))}$$

Unsuccessful branch:

$$\frac{\boldsymbol{\mathcal{D}}(k) \stackrel{E_0}{\longrightarrow}{}^* \boldsymbol{\mathcal{D}}(k+1), \quad \boldsymbol{\mathcal{D}}(k+1) \neq \boldsymbol{\mathcal{D}}}{\begin{array}{c} P : (\ \mathit{if}\ E_0 = \boldsymbol{\mathcal{D}}\ \mathit{then}\ E_1\ \mathit{else}\ E_2; E, \boldsymbol{\mathcal{D}}(k)) \longrightarrow \\ P.\,(E_2, \boldsymbol{\mathcal{D}}(k)) : (E_2; E, \boldsymbol{\mathcal{D}}(k+1)) \end{array}}$$

□

Note that if the conditional branch test is satisfied, the solver expression E_1 operates on the old solution space $\boldsymbol{\mathcal{D}}(k)$. If the test fails, the solver expression E_2 operates on the new solution space $\boldsymbol{\mathcal{D}}(k+1)$. This produces the desired outcome as in the example above.

The solver collaboration can be used as Monfroy does to produce hybrid constraint solvers. Some examples of this use are demonstrated in section G and E.1.1. Moreover, the extensions to the solver collaboration language discussed in this section, combined with a decomposition strategy, allow more complex and potentially more powerful hybrids.

Appendix G

An example of many solvers in serial

This appendix describes in more detail the experiment described in section 7.6. This experiment was carried out to study the asymptotic behaviour of various constraint satisfaction algorithms on a simple case study. The algorithms studied were INCES [62], a numerical algorithm [46] and a sequential hybrid. The purpose of the experiment was to give empirical as well as theoretical evidence that the hybrid algorithm was sound, complete and more efficient than the other two algorithms as well as to investigate sequential collaboration. It was anticipated that the hybrid would be approximately linear in complexity, whilst the other two algorithms would be quadratic. This would help to underline the advantages of using hybrid algorithms.

For convenience, the description of the case study is repeated in the following section. A detailed description of the manner in which the experiment was conducted is then explained and the problems encountered are described in full.

G.1 Case study

The case study chosen was an extension of Lamounier's internal combustion engine case study (see [64] and section 7.3). That problem studied the integration of some algebraic equations with the geometric constraints describing the construction of the piston. The two problems were linked so that the size of the piston and the length of the crankshaft were variables both in the functional problem and also in the geometric problem. However, this is a fixed size of problem. In order to study the asymptotic behaviour of the algorithms, n piston problems were joined together,

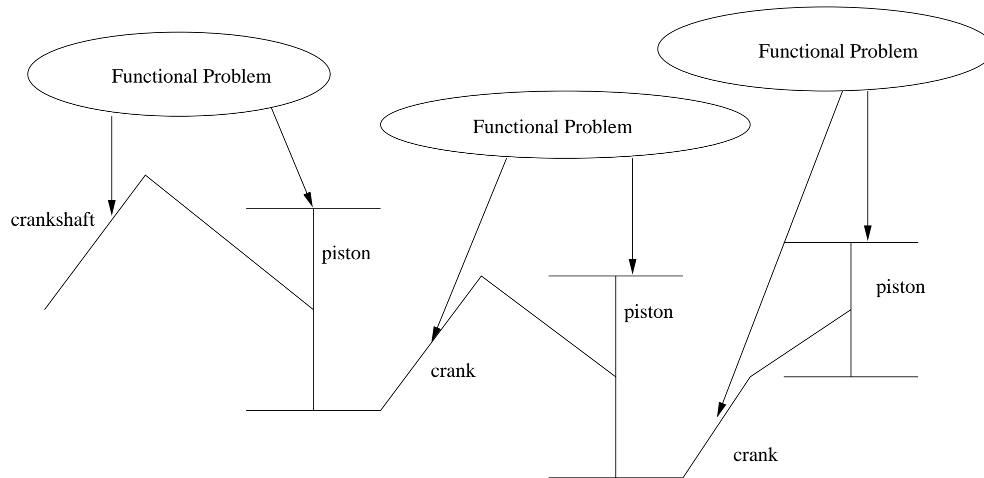


Figure G.1: Case Study of n Piston Problems Linked Together

as in figure G.1.

In this case study, the n pistons are linked by coincident constraints at each end of the piston. Thus the problems are all connected and the complexity of the problem does increase as the size of n increases. The functional problems are not linked and are effectively lots of small, fixed problems solved independently.

G.2 The solvers used

Three solvers were examined. The NAG C05NBC function [46] was used as a numerical solver. It was passed the whole set of constraints and used numerical techniques to converge towards a solution. The speed of convergence depended heavily on the initial guess, but the best case complexity of the NAG function is $\Omega(n^2)$, where n is the number of constraints.

Lamounier's INCES solver [62] is also capable of solving the problem as a whole. However, INCES deals only with equations and not geometric constraints. Geometric constraints can be handled if they are reduced to the constituent equations. INCES was expected to be quadratic, as it dealt with the problem as a whole and resorted to numerical solvers if loops appeared.

These constraint solvers were compared with a hybrid formed from combining the functional solver INCES and the geometric solver IGCS, much as in case study 1 (section 7.3). Each functional problem was solved using INCES and the results were passed to IGCS by varying the size of the lines in IGCS (see figure G.2). It was hoped that the hybrid would be able to take best advantage of the domain-specific

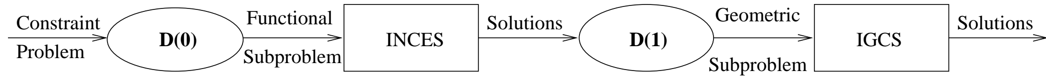


Figure G.2: Serial Hybrid used to Solve n Piston Problems Linked Together

knowledge incorporated in the INCES and IGCS solvers and would be linear. The theoretical complexity analysis of the hybrid system is presented in the next section.

The decomposition strategy, De, used in this case is to decompose problem $P = (\Phi, \Psi)$ into a set $\{(S_i, P_i)\}$, where solver S_i is IGCS if subproblem P_i is geometric and S_i is INCES if subproblem P_i is algebraic. Decomposition is performed by first identifying constraints as geometric or algebraic. These form two sets of constraints Ψ'_1 and Ψ'_2. Constructing Ψ'_1 and Ψ'_2 takes time $\mathrm{O}(n)$, where n is the number of constraints.

Set Ψ'_i is then decomposed further into sets Ψ''_j of connected components, where $\Psi_1 \in \Psi'_i$ and $\Psi_2 \in \Psi'_i$ are connected if there is a path between Ψ_1 and Ψ_2 in the constraint/entity graph of constraint problem $(\Phi, \Psi \setminus \Psi'_j), j \neq i$. Finding the connected components can be done in a simple graph traversal algorithm that takes time $\mathrm{O}(m)$, where m is the number of edges in the constraint/entity graph. Since the imposed sets of constraints are usually quite small, m will typically be a multiple of the number of constraints in P. Thus decomposition of P takes time $\mathrm{O}(n)$.

With this decomposition strategy, the hybrid can be described in the solver collaboration language of section 7.5 as

$$((S_1, P_1); (S_2, P_2); (S_3, P_3); \ldots ; (S_n, P_n))\,.$$

G.2.1 Expected behaviour of hybrid

Each functional problem is solved individually and independently of any other and the results of each problem are then used as input to the appropriate line segments of the geometric problem. Since each problem is independent, it is solved in constant time. Therefore n such problems take $\mathrm{O}(n)$ time, where n is the number of constraints.

The geometric problem increases in complexity as the number of problem instances increases. However adding a new instance of the problem, due to the nature of IGCS, should only take a constant amount of time to translate each new line segment so that it is coincident to the previous line segment. Since the constraints are processed in the order they are positioned, the algorithm should take $\mathrm{O}(n)$ time overall, where n is the number of constraints.

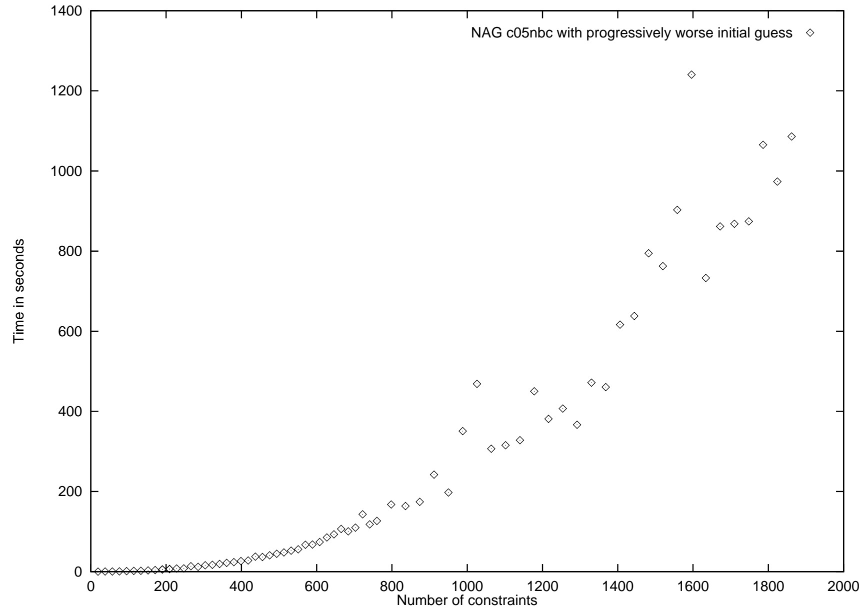


Figure G.3: NAG function with bad initial guess

Since there are no loops in each constraint subproblem, the hybrid of INCES and IGCS should have a linear behaviour. Since the functional problems only provide one solution to the geometric problem, the hybrid solver should be $O(n)$. This compares favourably with the other two constraint solvers.

G.3 Results

The case study was run for problem sizes between 1 and 200. This gave problems with between 19 and 3800 variables. For all of the following graphs, the x-axis is the number of variables in the problem and the y-axis is the amount of time taken to solve the problem in seconds. All case studies were run on a Silicon Graphics Indy with an R4600 100MHz IP22 processor and 32 Mbytes of memory.

The NAG C05NBC function gave the results in figure G.3 and in figure G.4. Figure G.3 shows that with an initial guess that is progressively further and further away from the solution, the time that the NAG algorithm takes to solve the problem increases in a nonlinear fashion. Divergent results, such as those at $n = 1000$ were identified by the NAG algorithm as 'Not Improving' and had not converged to a solution by the time the algorithm terminated. However, underlying the divergent behaviour, a nonlinear pattern is apparent. For a problem size of 1900 variables, the NAG solver takes approximately 1100 seconds, more than 20 minutes.

Figure G.4 shows the NAG C05NBC function with a consistently good guess.

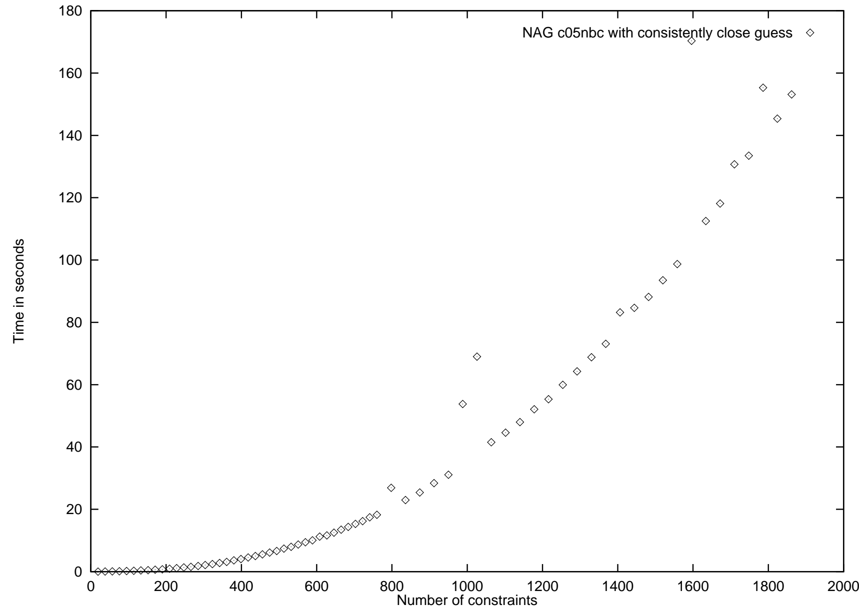


Figure G.4: NAG function with good initial guess

The curve shown is much more regular but the numerical solver still diverges occasionally, for example when $n = 988$. However, even with a good initial guess, the NAG function is still nonlinear, probably quadratic. For a problem size of 1900, the NAG function takes 155 seconds.

Figure G.5 shows the results for the INCES solver on the problem. Again, the solver is obviously nonlinear, but the solution is always found. Note that the INCES algorithm appears to be about 3 times as fast as the NAG solver. For a problem size of 1900, INCES took about 55 seconds.

Figure G.6 shows the results for the hybrid solver. Even though the curve is nonlinear, the time taken to solve a 1900 variable problem is less than a second, two orders of magnitude faster than the INCES algorithm. However, the nonlinear curve does not agree with the expected behaviour of the hybrid as a linear function was expected. Further analysis of the INCES and IGCS algorithms discovered that the parametric constraint list and dependency hierarchy lists respectively were reducing the two algorithms to quadratic behaviour.

In INCES, the parametric constraint list is used as a global data structure to describe the entire constraint problem. Whenever a constraint is referenced, the INCES solver searches through the parametric constraint list to find the reference to the appropriate constraint. INCES is an incremental solver and correspondingly, each time a constraint is added, the parametric constraint list is checked. The constraint list is of O(n) size, where n is the number of constraints in the problem,

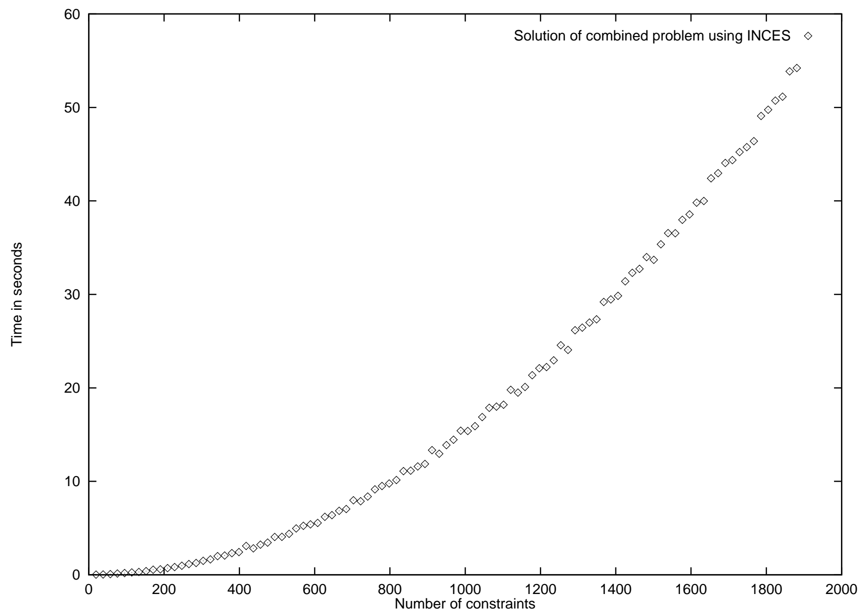


Figure G.5: Solving with INCES

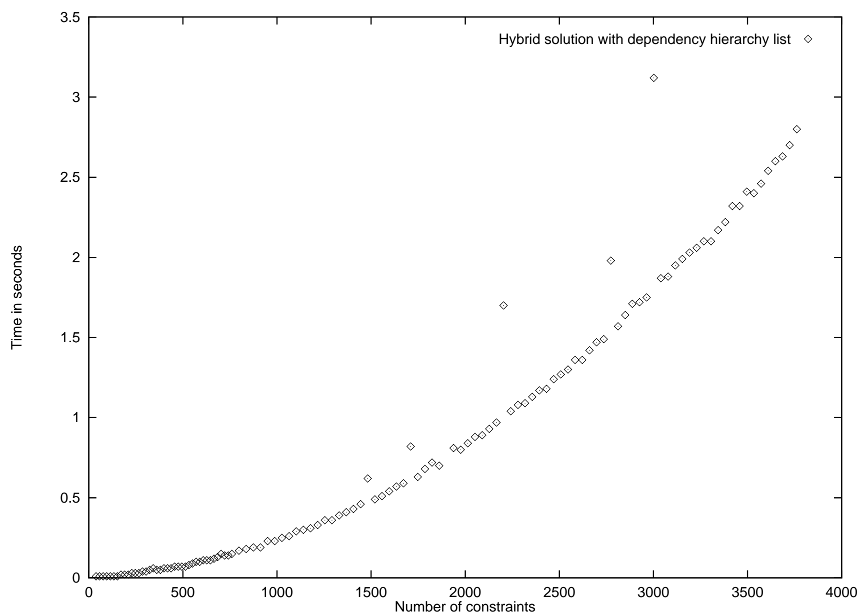


Figure G.6: Hybrid solution using IGCS and INCES

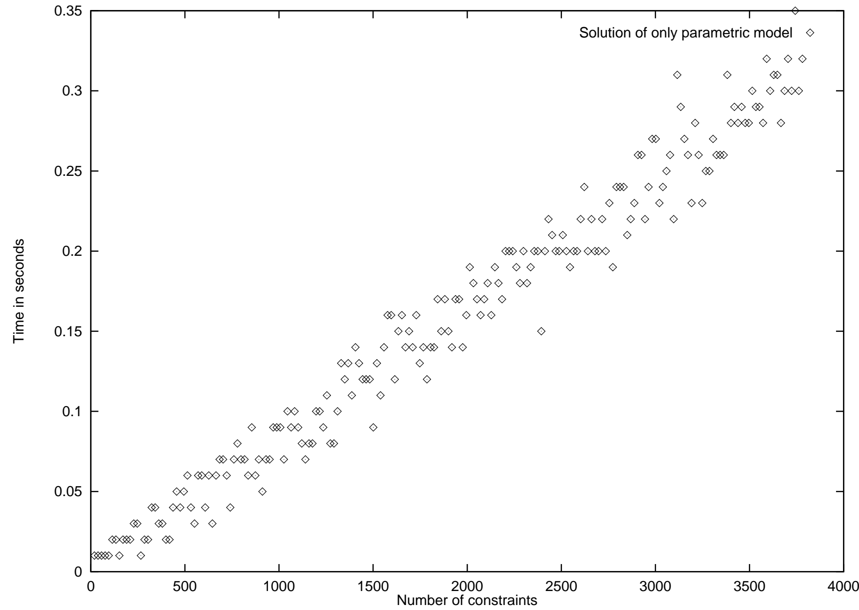


Figure G.7: Solving the functional problem only using INCES without global parametric constraint list

and since each time a constraint is added the parametric constraint list is studied, by the time n constraints have been added to the problem, the parametric constraint list has been consulted n times. The first such consultation takes 1 lookup, the second 2 lookups, until the final constraint insertion takes n lookups. Thus, just studying the parametric constraint list takes $O(n^2)$.

Removing the parametric constraint list each time the functional problem was solved resulted in the graph in figure G.7 showing the time taken to solve only the algebraic model.

In IGCS, the dependency hierarchy list is used to search for loops in the constraint graph. The dependency hierarchy of an entity is the list of entities for which solutions must be found before solutions to the entity itself can be found. In order to find whether a constraint problem has a loop or not, when a constraint between two entities is added, the dependency hierarchy of both entities is compared. If they share a common ancestor in the dependency hierarchy, then a loop has been created. Unfortunately, the implementation of this dependency hierarchy list results in the quadratic behaviour of IGCS.

As a new constraint is added between two entities, the dependency hierarchy lists of the two entities are compared. In the long chain of constraints used in this case study, the constraint is always between an entity with a dependency hierarchy of the whole problem and an entity with a dependency hierarchy of only a few entities.

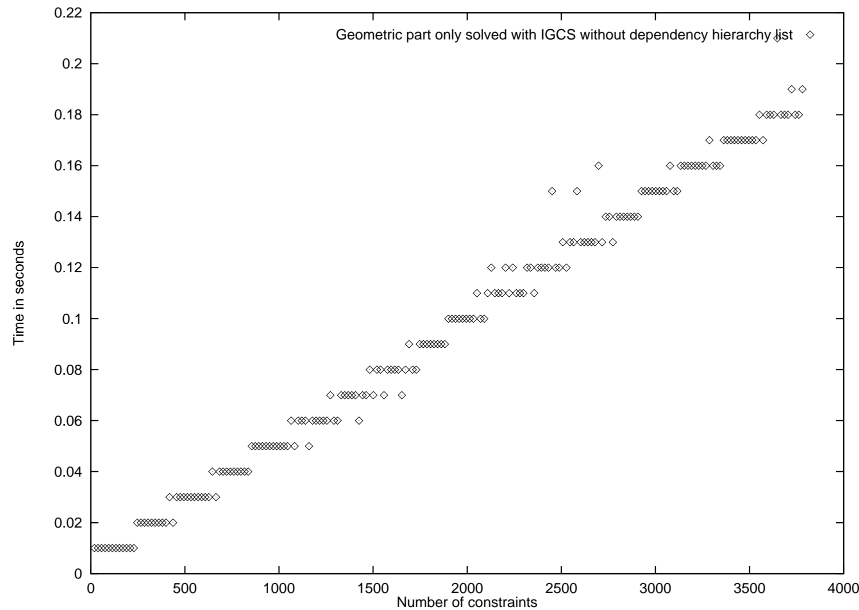


Figure G.8: Solving the geometric problem only using IGCS without the dependency hierarchy list

However, IGCS as currently implemented searches through the whole set of entities in each dependency hierarchy to find common elements. This takes O(m), where m is the number of entities in the problem. This is repeated for each new constraint as it is inserted and since there are n constraints, the operation as a whole takes O(mn). Since the number of constraints is usually similar to the number of entities simply consulting the dependency hierarchy list in IGCS takes O(n^2).

Removing the dependency hierarchy list altogether resulted in the graph of figure G.8 showing the time taken to solve only the geometric problem. Note that both of these graphs are roughly linear. Irregularities can be associated with the coarseness of the timing function available.

Recombining the new, linear versions of IGCS and INCES results in a hybrid algorithm with results as in figure G.9. Note that the hybrid is approximately linear in complexity and that the time taken to solve a 1900 variable problem is approximately 0.25 seconds, an order of magnitude improvement over the previous version.

G.4 Conclusions

The hybrid constraint solver is very fast indeed. It is linear, whereas the other solvers compared were quadratic at best. It is three or four orders of magnitude

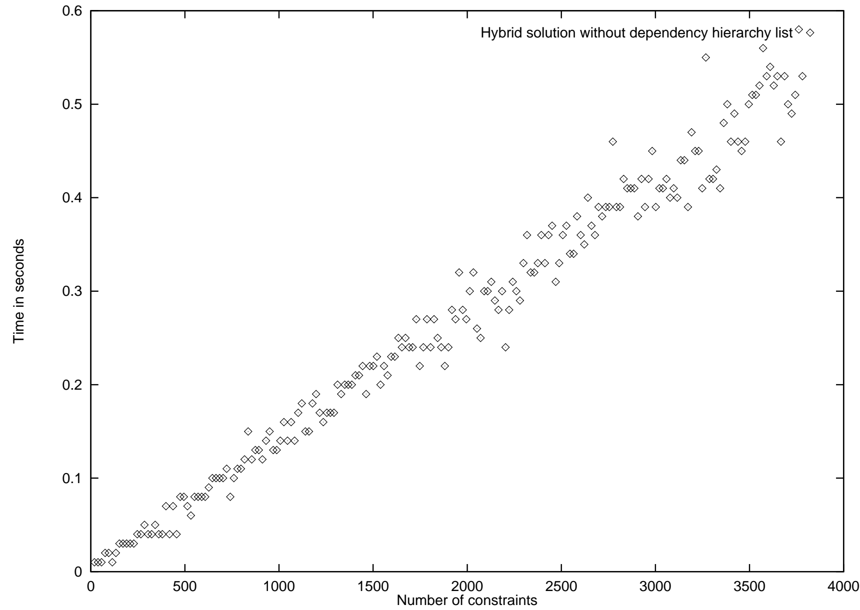


Figure G.9: Solution using a hybrid algorithm of IGCS without depH list and INCES without global parametric constraint list

faster than the NAG function. Figure G.10 shows the various graphs in relation to each other. Even for problems of 100 or so variables, the hybrid is much faster. It is also significant that the hybrid with the depH list, whilst nonlinear, is still much faster than INCES and C05NBC. The depH list is used to identify loops and, as IGCS cannot handle loops without it, something equivalent will have to be implemented. Disjoint forests [21] would increase the complexity of the current $O(n)$ algorithm to $O(n \log n)$ for example, rather than $O(n^2)$, where n is the number of constraints.

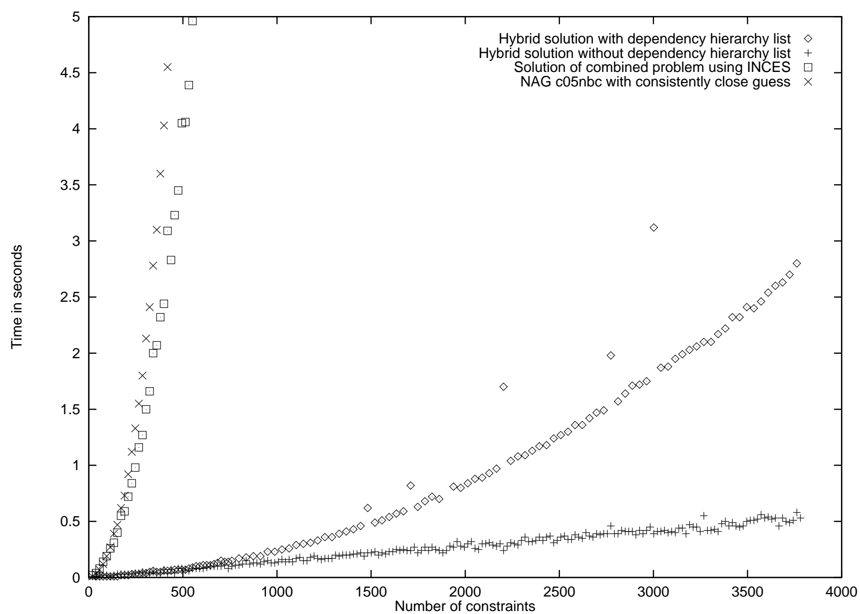


Figure G.10: A comparison of the C05NBC function and INCES algorithm with the hybrid solver

Appendix H

Glossary

Articulation pair	A pair of vertices (A, B) in a graph G are an articulation pair if removal of A and B from G, along with all edges incident to A or B, disconnects G.
Bijections	A bijection is a function that is injective and surjective. If f is bijective then f has well-defined inverse.
Bipartite graph	A bipartite graph is a graph (V, E) such that $V = U \cup W$, $U \cap W = \emptyset$ and $\forall (a, b) \in E$, $(a \in U \wedge b \in W)$.
Cartesian product	The Cartesian product of two sets A and B is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.
Connected graph	A graph $G = (V, E)$ is connected if for every pair of vertices u and v in V, there exists a path from u to v.
Countable sets	A set A is countable if it is finite or if there exists a bijection $f : \mathbb{N} \to A$.
Directed edges	An edge e of graph G is directed if there is an order placed on the vertices in the edge. Directed edges are represented using the notation $[u, v]$ where u and v are vertices in G.
First order formulae	First order predicate logic formulae,
Graph	Let V be a finite set of vertices and E be a subset of the unordered pairs of vertices. Then a graph is the ordered pair (V, E).
Hyperedge	A hyperedge is a finite set of vertices.

Hypergraph	A hypergraph (V, HE) is an ordered pair of a finite set of vertices V and a finite set of hyperedges HE.
Injective functions	A function $f : A \rightarrow B$ is injective if $f(x) = f(y)$ implies $x = y$.
Labelled graph	A graph $G = (V, E)$ is labelled if each edge $e \in E$ is labelled with a symbol.
Loop	A loop in a graph G is an edge (u, u) where u is a vertex in G.
Path	A path in a graph $G = (V, E)$ is an alternating sequence of vertices and edges $v_0, e_1, v_1, \ldots, e_n, v_n$ where $e_i = (v_i, v_{i+1})$, all edges are different and no vertices are repeated except possibly that the last and the first are the same.
Quantifier free	A first order formula lacking the symbols $\forall$ and $\exists$,
Simple graphs	A simple graph has at most one edge between any two vertices.
Strongly connected components	A graph is a strongly connected component if there exists a path from every vertex to every other vertex.
Surjective functions	A function $f : A \rightarrow B$ is surjective if $f(A) = B$, where $f(A) = \{f(a), a \in A\}$.
Symmetric constraints	A constraint C is symmetric if $\xi(C) = \{x_{i_1}, \ldots, x_{i_j}\}$ is the imposed set on C, $v_{i_1}, \ldots, v_{i_j}$ are values in $D_{i_1}, \ldots, D_{i_j}$ of $x_{i_1}, \ldots, x_{i_j}$ respectively and $$(v_1, \ldots, v_{i_1}, \ldots, v_{i_j}, \ldots, v_n) \in C \Rightarrow (v_1, \ldots, \pi_l(v_{i_1}), \ldots, \pi_l(v_{i_j}), \ldots, v_n) \in C$$ for all permutations π_l of $(v_{i_1}, \ldots, v_{i_j})$. A constraint is non-symmetric if it is not symmetric.
Undirected edges	An edge e of graph G is undirected if there is no order placed on the vertices in the edge. Undirected edges are represented using the notation (u, v) where u and v are vertices in G.
Triconnected components	A graph is a triconnected component if it contains no articulation pairs.

Bibliography

[1] B. Aldefeld. Variation of geometries based on a geometric-reasoning method. *Computer-Aided Design*, 1988.

[2] Ram Anantha, Glenn A. Kramer, and Richard H. Crawford. Assembly modelling by geometric constraint satisfaction. *Computer-Aided Design*, 28(9):707–722, 1996.

[3] R. Anderl and R. Mendgen. Modelling with constraints: Theoretical foundation and application. *Computer-Aided Design*, 28(3):155–168, 1996.

[4] Farhad Arbab and Bin Wang. A constraint-based design system based on operational transformation planning. In *Proceedings of the 4th International Conference on the Applications of Artificial Intelligence in Engineering*, pages 405–426, Cambridge, UK, July 1989.

[5] P. Atzeni and V. De Antonellis. *Relational Database Theory*. The Benjamin/Cummings Publishing Company Inc, 1993.

[6] F. Baader and K. Schulz. On the combination of symbolic constraints, solution domains and constraint solvers. In *Proceedings of the first International Conference on Principles and Practice of Constraint Programming - CP95*, volume 976 of *Lecture Notes in Computer Science*, pages 380–397. Springer-Verlag, 1995.

[7] F. Baader and K. U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1995.

[8] D. Baraff. Interactive simulation of solid rigid bodies. *IEEE Computer Graphics and Applications*, pages 63–74, May 1995.

[9] Sanjay Bhansali, Glenn A. Kramer, and Tim J. Hoar. A principled approach towards symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, 1996.

[10] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201, March 1997.

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

[12] Alan Borning. Thinglab - a constraint-oriented simulation laboratory. Technical Report SSL-79-3, Xerox Palo Alto Research Center, July 1979.

[13] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A Geometric Constraint Solver. Technical Report CSD-Tr-93-054, Department of Computer Science, Purdue University, August 1993.

[14] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *Computer-Aided Design*, 27(6):487–501, June 1995.

[15] Mark W. Brunkhart. Interactive geometric constraint systems. Master's thesis, Computer Science Division, Deparment of Electrical Engineering and Computer Science, University of California, Berkeley, 1994.

[16] S.A. Buchanan and A. de Pennington. Constraint Definition System: a Computer-Algebra based Approach to Solving Geometric-Constraint Problems. *Computer-Aided Design*, 25(12):741–750, December 1993.

[17] B. Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional systems theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[18] Bruce W Char, Keith O Geddes, Gaston H Gonnet, Benton L Leone, Michael B Monagan, and Stephen M Watt. *Maple V Library Reference Manual.* Springer-Verlag, 1991.

[19] J.C.H. Chung and M.D. Schussel. Technical evaluation of variational and parametric design. In *Proceedings of Autofact '89*, pages 5/27–5/44, 1989.

[20] C. Clarke. Pro-engineer. *CAD/CAM*, 12(1), January 1993.

[21] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[22] Maurice Dohmen. A survey of constraint satisfaction techniques for geometric modeling. *Computers and Graphics*, 19(6):831–845, 1995.

[23] Jean-Francois Dufourd, Pascal Mathis, and Pascal Schrek. Formal resolution of geometric constraint systems by assembling. In *Proceedings of Solid Modelling 97*, 1997.

[24] Lynn Eggli, Ching yao Hsu, Beat Brüderlin, and Gershon Elber. Inferring 3d models from freehand sketches and constraints. *Computer-Aided Design*, 29(2):101–112, 1997.

[25] M. Fa. *Interactive Constraint-based Solid Modelling*. PhD thesis, School of Computer Studies, University of Leeds, September 1993.

[26] M. Fa, T. Fernando, and P. M. Dew. Direct 3D Manipulation Techniques for Interactive Constraint-based Solid Modelling. *Computer Graphics Forum, Proc. of EuroGraphics'93*, 12(3):237–248, September 1993.

[27] M. Fa, T. Fernando, and P. M. Dew. Interactive Constraint-based Solid Modelling using Allowable Motion. *Proc. of ACM/SIGGRAPH Symposium on Solid Modelling and Applications*, pages 243–252, May 1993.

[28] J.-C. Faugere. *Résolution des systèmes d'équations algébriques*. PhD thesis, Université Paris 6, 1994.

[29] L.T.P. Fernando, P.M. Dew, and F. Gao. Constraint-based interaction techniques for supporting a distributed collaborative engineering environment. In *Proceedings of the First Workshop on Simulation and Interaction in Virtual Environments - SIVE '95*, pages 265–270, 1995.

[30] L.T.P. Fernando, M. Fa, P.M. Dew, and M. Munlin. Constraint-based 3d manipulation techniques within virtual environments. In R.A. Earnshaw, editor, *Virtual Reality Applications*, pages 71–89. Academic Press, 1995.

[31] T. Fernando, P. M. Dew, M. Fa, J. Maxfield, and N. Hunter. A Shared Virtual Workspace for Constraint-based Solid Modelling. *EuroGraphics Workshop on Virtual Environments*, September 1993.

[32] T. Fernando, M. Fa, P. M. Dew, and Mudarmeen Munlin. Constraint-based 3D Manipulation Techniques for Virtual Environments. In *Proc. of International State of the Art Conference (BCS) on Applications of Virtual Reality, Ed. by R. A. Earnshaw, J. Vince and H. Jones*, 1994.

[33] R. Fraïssé. *Theory of Relations*, volume 118 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers, Amsterdam, 1986.

[34] B. N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1), January 1990.

[35] E. Freuder and P. Hubbe. A disjunctive control schema for constraint satisfaction. In V. J. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*. MIT Press, 1995.

[36] E. C. Freuder and P.D. Hubbe. Extracting constraint satisfaction subproblems. In *14th International Joint Conference on Artificial Intelligence*, 1995.

[37] I. Fudos. Editable Representations for 2D Geometric Design. Master's thesis, School of Computer Studies, Purdue University, 1993.

[38] I. Fudos and C. Hoffmann. Correctness Proof of a Geometric Constraint Solver. Technical Report CSD 93-076, Department of Computer Science, Purdue University, December 1993.

[39] Ioannis Fudos and Christoph Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179, April 1997.

[40] Esther Gelle and Ian Smith. Dynamic constraint satisfaction with conflict management in design. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in LNCS, pages 237–252. Springer, 1996.

[41] M. Gleicher. Integrating Constraints and Direct Manipulation. *1992 Symposium on Interactive 3D Graphics*, pages 171–174, 1992.

[42] M. Gleicher. A Graphics Toolkit Based on Differential Constraints. *UIST '93*, pages 109–120, November 1993.

[43] M. Gleicher and A.Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[44] Michael Gleicher. *A Differential Approach to Graphical Interaction.* PhD thesis, School of Computer Science, Carnegie Mellon University, November 1994. CMU-CS-94-217.

[45] Sreenivasa R Gorti and Ram D Sriram. From symbol to form: a framework for conceptual design. *Computer-Aided Design*, 28(11):853–870, November 1996.

[46] Nottingham Algorithms Group. Nag library manual : Mark 5, 1976. Fortran Edition.

[47] Christoph Hoffmann and Jaroslaw Rossignac. A road map to solid modeling. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):3–10, March 1996.

[48] C.M. Hoffmann and R.Juan. Erep - An Editable, High-level Representation for Geometric Design and Analysis. Technical report, Department of Computer Sciences, Purdue University, 1994.

[49] J.E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal of Computation*, 2(3):135–158, September 1973.

[50] H. Hosobe, K. Miyashita, S. Takahashi, S. Matuoka, and A. Yonezawa. Locally simultaneous constraint satisfaction. In Alan Borning, editor, *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA, May 1994.

[51] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Beyond finite domains. In Alan Borning, editor, *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA, May 1994.

[52] Joxan Jaffar and Michael Maher. Constraint logic programming: a survey. *Journal of Logic Programming, Special 10th Anniversary Issue,*, 19/20, May/July 1994.

[53] Narendra Jussien and Patrice Boizumault. Implementing constraint relaxation over finite domains using ATMS. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in LNCS, pages 265–280. Springer, 1996.

[54] N.P. Juster. Modelling and Representation of Dimensions and Tolerances : A Survey. *Computer-Aided Design*, 24(1):3–17, January 1992.

[55] H. Kirchner and C. Ringeissen. Combining symoblic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[56] K. Kondo. Algebraic Method for Manipulation of Dimensional Relationships in Geometric Models. *Computer-Aided Design*, 24(3):141–147, March 1992.

[57] G. A. Kramer. Using Degrees of Freedom Analysis to Solve Geometric Constraints. In J. Rossignace and J. Turner, editors, *Proceedings Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 371–378, 1991.

[58] G. A. Kramer. A Geometric Constraint Engine. *Artificial Intelligence*, 58:327–360, 1992.

[59] Glenn A. Kramer. *Solving Geometric Constraint Systems.* MIT Press, 1992.

[60] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, pages 32–44, Spring 1992.

[61] Timo Laakko and Martti Mäntylä. Incremtnal constraint modelling in a feature modelling system. In J. Rossignac and F. Sillion, editors, *EUROGRAPHICS '96*, volume 15. Eurographics Association, Blackwell Publishers, 1996.

[62] E. Lamounier, T. Fernando, and P. Dew. An Incremental Constraint Equation Solver for Variational Design. In *Proceedings of the Fourth International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS'95)*, pages 81–90, December 1995.

[63] Edgard Lamounier. First Year Report. Technical report, School of Computer Studies, University of Leeds, 1994.

[64] Edgard Lamounier. *An incremental constraint-based approach to support engineering design.* PhD thesis, School of Computer Studies, University of Leeds, 1996.

[65] Hervé Lamure and Dominique Michelucci. Solving geometric constraints by homotopy. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):28–33, March 1996.

[66] R. Latham and A. Middleditch. Connectivity Analysis : A Tool for Processing Geometric Constraints. Technical report, Brunel University, UK, August 1994.

[67] Richard Latham and Alan Middleditch. Connectivity analysis: a tool for processing geometric constraints. *Computer-aided Design*, 28(11):917–928, November 1996.

[68] Richard Samuel Latham. *Combinatorial algorithms for the analysis and satisfaction of geometric constraints*. PhD thesis, Brunel University, 1996. PhD L356.

[69] Robert Light and David Gossard. Modification of geometric models through variational geometry. *Computer-Aided Design*, 14(4):209–214, July 1982.

[70] V.C. Lin, D.C. Gossard, and R.A. Light. Variational Geometry in Computer Aided Design. *Computer Graphics*, 15(3):171–175, August 1981.

[71] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope : A Constraint Imperative Programming Language. Technical Report UW-CSE-93-09-04, University of Washington, 1993.

[72] Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, 32(2):108–120, February 1983.

[73] D-Cubed Ltd. The 2-d dcm technical overview. 68 Castle Street, Cambridge, CB3 0AJ, England.

[74] D-Cubed Ltd. 3-d dcm technical overview. 68 Castle Street, Cambridge, CB3 0AJ, England.

[75] D-Cubed Ltd. An overview of d-cubed and the dcm. 68 Castle Street, Cambridge, CB3 0AJ, England.

[76] M. Mantyla. A Modelling System for Top-down Design of Assembled Products. *IBM J. Res. Develop.*, 34(5):636–658, 1990.

[77] M. Mantyla. WAYT: Towards a Modelling Environment for Assembled Products. *Intelligent CAD III*, pages 187–203, 1991.

[78] J. Maxfield, L.T.P. Fernando, and P.M. Dew. A distributed virtual environment for collaborative engineering. In *Proceedings Virtual Reality Annual International Symposium - VRAIS'95*, pages 162–170, 1995.

[79] John Maxfield. *A Distributed Virtual Environment for Synchronous Collaboration in Simultaneous Engineering*. PhD thesis, School of Computer Studies, University of Leeds, July 1996.

[80] M.Bouzoubaa, B. Neveu, and G.Hasle. Houria : A solver for equational constraints in a hierarchical system. In *Proceedings of the OCS workshop in conjunction with CP-95*, Cassis, France, 1995.

[81] Pedro Meseguer. Constraint Satisfaction Problems: An Overview. *AI Communications*, 2(1):3–17, March 1989.

[82] E. Monfroy and C. Ringeissen. Domain-independent constraint solver extension. Technical report, Centre de Recherche en Informatique de Nancy, Vandoeuvre-ls-Nancy, 1996.

[83] E. Monfroy, M. Rusinowitch, and R. Schott. Implementing non-linear constraints with cooperative solvers. Technical Report Technical Report 95-R-110, Centre de Recherche en Informatique de Nancy, Vandoeuvre-ls-Nancy, 1995. Also as INRIA Technical Report RR-2747.

[84] Eric Monfroy. An environment for designing/executing constraint solver collaborations. Technical report, Centre de Recherche en Informatique de Nancy, Vandoeuvre-ls-Nancy, 1996.

[85] Eric Monfroy, Michael Rusinowitch, and Rene Schott. Implementing non-linear constraints with cooperative solvers. In K. M. George, J. H. Carrolland D. Oppenheim, and J. Hightower, editors, *Proceedings of ACM Symposium on Applied Computing, SAC '96*, pages 63–72, February 1996.

[86] J. Owen. Algebraic Solution for Geometry from Dimensional Constraints. *Symposium on Solid Modelling Foundations and CAD/CAM Applications*, June 1991.

[87] J. Pabon, R. Young, and W. Keirouz. Integrating Parametric Geometry, Features and Variational Modeling for Conceptual Design. *International Journal of Systems Automation: Research and Applications (SARA)*, 2:17–36, 1992.

[88] G. Pahl and W. Beitz. *Engineering Design*. Design Council, 1984.

[89] John Platt. A generalization of dynamic constraints. *CVGIP: Graphical Models and Image Processing*, 54(6):516–525, November 1992.

[90] William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. *Numerical Recipes : The Art of Scientific Computing*. Cambridge University Press, 1986.

[91] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[92] Jean-Francois Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS-94 (Singapore International Conference on Intelligent Systems)*, 1994.

[93] A. A. G. Requicha. Representations of Tolerances in Solid Modelling: Issues and Alternative Approaches. In *Solid Modelling by Computers From Theory to Applications*, 1984.

[94] M. Sanella. The SkyBlue Constraint Solver and its Applications. *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, 1994.

[95] M. Sanella and R.A. Borning. Multi-Garnet - Integrating Multi-Way Constraints with Garnet. Technical Report UW-CSE-92-07-01, University of Washington, 1992.

[96] M. Sanella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-Way vs One-Way Constraints in GUIs - Experience with the Deltablue Algorithm. Technical Report UW-CSE-92-07-05a, University of Washington, 1993.

[97] M. Sapossnek. Research on constraint-based design systems. In *Applications of AI '89*, 1989.

[98] D. Serrano. Managing Constraints in Concurrent Design : First Steps. *ASME 90*, pages 159–164, 1990.

[99] D. Serrano. Automatic dimensioning in design for manufacturing. *SM '91*, pages 379–385, 1991.

[100] David Serrano. *Constraint Management in Conceptual Design*. PhD thesis, Department of Mechanical Engineering, MIT, 1987.

[101] David Serrano and David Gossard. Tools and Techniques for Conceptual Design. In *Artificial Intelligence in Engineering Design*, volume 1, chapter 3, pages 71–116. Academic Press, Inc, 1992.

[102] Shuichi Shimizu and Masayuki Numao. Constraint-based Design for 3D Shapes. *Artificial Intelligence*, 91(1):51–69, 1997.

[103] Barbara Smith. A Tutorial on Constraint Programming. Technical Report 95-14, University of Leeds, April 1995.

[104] Barbara Smith and Martin Dyer. Locating the phase transition in constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996. Special issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

[105] Barbara Smith and Stuart Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of IJCAI-95*, volume 1, pages 646–651, August 1995.

[106] W. Sohrt and J.D. Bruderlin. Interaction with Constraints in 3D Modelling. *International Journal of Computational Geometry and Applications*, 1(4):405–425, 1991.

[107] Michael Spivak. *A Comprehensive Introduction to Differential Geometry*, volume One. Publish or Perish, Inc, 2nd edition, 1979.

[108] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, MIT, Cambridge, Mass., 1963.

[109] W. A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford Science Publications, 1987.

[110] Martin Thompson. Techniques for supporting maintenance analysis in virtual environments. First Year Transfer Report, School of Computer Studies, University of Leeds, 1996.

[111] J-C Tsai, R. Konkar, and M.R. Cutkosky. Issues in Incremental Analysis of Assemblies for Concurrent Design. *2nd International Conference on AI in Design*, 1992.

[112] Y. T. Tsai, T. Fernando, and P.M. Dew. Exploiting degrees of freedom analysis for interactive constraint-based design. In N. M. Thalmann and V. Skala, editors, *The Fourth International Conference in Central Europe on Computer Graphics andVisualization'96 (WSCG '96)*, pages 377–387, Plzen, Czech Republic, February 1996.

[113] Yung-Teng Tsai. *Incremental Geometric Constraint Satisfaction Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1996.

[114] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[115] Edward Tsang and Alvin Kwan. Mapping constraint satisfaction problems to algorithms and heuristics. Technical Report CSM-198, Department of Computer Science, University of Essex, 1993.

[116] Edward P. K. Tsang, James Borrett, and Alvin C. M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In J. Hallam, editor, *Proceedings AISB-95*, pages 203–216. IOS Press, Amsterdam, 1995.

[117] EDS Unigraphics. Unigraphics cad/cam/cae. World Wide Web Page. `http://www.ug.eds.com/ug/`.

[118] G. Vanecek, Jr and J. F. Cremer. Project isaac: Building simulations for virtual environments. Technical report, Department of Computer Science, Purdue University, 1994.

[119] A. Verroust, F. Schonek, and D. Roller. Rule-oriented method for parameterised computer-aided design. *Computer-Aided Design*, 24(10):531–540, October 1992.

[120] Roger Westbrook. *Structural Engineering Design in Practice.* Construction Press, 1984.

[121] Kevin Wise. Using multidimensional csg models to map where objects can and cannot go. Technical Report 001/1996, University of Bath, January 1996. `http://www.bath.ac.uk/~enskdw/Tech_rep_001_96/trans_rep.html`.

[122] Andrew Witkin, David Baraff, and Michael Kass. *An Introduction to Physically Based Modeling*, chapter Constrained Dynamics. World Wide Web Page `http://www.cs.cmu.edu/~baraff/pbm/pbm.html`, 1997.

[123] Armin Wolf. Transforming ordered constraint hierarchies into ordinary constraint systems. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in LNCS, pages 171–188. Springer, 1996.

[124] Anthony Wren and Jean-Marc Rousseau. Bus driver scheduling - an overview. Technical Report 93.31, School of Computer Studies, University of Leeds, 1993.

[125] Yasushi Yamaguchi and Fumihiko Kiumra. A constraint modeling system for variational geometry. In J. Wozny, J. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 221–233. North Holland, 1990.