

DISSERTATION

Constraint Solving for Model-based Engineering Applications using Relational Aggregation

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften unter der Leitung von

Prof. Dr. Georg Gottlob

E184

Institut für Informationssysteme,
Abteilung für Datenbanken und Artificial Intelligence (E1842)

eingereicht an der Technischen Universität Wien,
Fakultät für Informatik
von

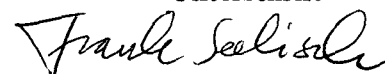
Diplom-Mathematiker Frank Seelisch,

Matrikelnummer: 0027301

89073 Ulm, Platzgasse 9, Deutschland,

Wien, am 23.03.2004

Unterschrift



Deutsche Zusammenfassung dieser Dissertation

Das Thema der vorliegenden Dissertationsschrift ist die Constraint-Verarbeitung für modellbasierte Applikationen im Engineering. Die zugrunde liegende Methodik ist hierbei die *relationale Aggregation*.

Im größeren Kontext ordnet sich diese Dissertation damit ein in Bestrebungen, die die weitere Automatisierung von Standard- und Nichtstandardaufgaben im Engineering zum Ziel haben. Vor dem Hintergrund ständig wachsender Datenbestände besteht heute mehr denn je die Notwendigkeit, Wissen automatisch verarbeiten zu können. Eine Voraussetzung dafür ist, dass dieses Wissen einem entsprechenden Formalisierungsgrad genügt.

Constraints sind eine besonders formale Form von Wissen, welches das statische physikalische Verhalten von einzelnen Bauteilkomponenten mathematisch beschreibt. Sie ermöglichen somit die mathematische Analyse hierarchisch zusammengesetzter Systeme, sobald für jede enthaltene Komponente eine hinreichende mathematische Beschreibung vorliegt.

Die in dieser Dissertation entwickelte Methodik der *relationalen Aggregation* erlaubt es, die wichtigsten Fragen im Bezug auf ein vorgegebenes Constraintproblem zu beantworten. Der präsentierte Ansatz basiert auf einem abstrakten Constraint-Begriff: Jeder Constraint wird aufgefasst als Menge von zulässigen Wertzuweisungen oder - äquivalent dazu - Lösungstupeln. (Neben der Menge der eingeschränkten Variablen spielen bei dieser Betrachtungsweise keine weiteren Merkmale eine Rolle.) Unter Verwendung der aus der Datenbanktheorie bekannten Operationen *join* und *project* können dann sogenannte Aggregationsbäume konstruiert werden, deren Blattmenge mit der Menge aller Ausgangsconstraints koinzidiert. Der Wurzelknoten des Aggregationsbaumes hat nur zwei mögliche Ausprägungen. Die eine zeigt die Konsistenz des analysierten Constraintproblems an; die andere dessen Unerfüllbarkeit. Mithin entscheidet ein Aggregationsbaum die klassische Frage nach der Konsistenz. Im Falle eines konsistenten Problems kann der konstruierte Aggregationsbaum benutzt werden, um Lösungen für alle Unbekannten zu ermitteln; dies implementiert einen zweiten konventionellen Dienst.

Weiterhin erlauben Aggregationsbäume auf natürliche Art und Weise die effiziente Realisierung von Erklärungsmechanismen. Zum einen erlauben sie die Berechnung minimaler Konflikte im Falle eines inkonsistenten Problems und - im konsistenten Fall - die Ermittlung von minimalen Beweisen für die gefundenen Lösungen. Insbesondere im Engineering, wo zumeist mehrere hundert Constraints gleichzeitig durch den Ingenieur in Betracht zu ziehen sind, sind solche Mechanismen unerlässlich, um die Komplexität der Systeme und die Größe der Probleme beherrschen zu können. Diese Anforderung wurde in den letzten Jahren deshalb auch verstärkt durch Forschungsaktivitäten der wissenschaftlichen Community adressiert.

Eine zusätzliche Dimension im Engineering, die in dieser Dissertation vor allem am

Beispiel der modellbasierten Diagnose nachvollzogen wird, ist das effiziente Lösen nicht nur eines Constraintproblems sondern großer Sequenzen ähnlicher Constraintprobleme. Hierbei erweisen sich Algorithmen als unverzichtbar, die eine massive Wiederverwendung früherer Berechnungsschritte erlauben. In der vorgestellten Methodik der relationalen Aggregation wird dies effizient durch die Identifikation wiederverwendbarer Unterbäume existierender Aggregationsbäume bewerkstelligt. Mit der relationalen Aggregation sind also neben konventionellen Diensten auch die Umsetzung effizienter Erklärungsmechanismen und die multikontextuelle Analyse möglich.

Die Hauptbeiträge dieser Arbeit sind

- eine Anforderungsanalyse für Constraintprobleme im Engineering,
- die detaillierte Entwicklung der Methodik der relationalen Aggregation (basierend auf der vorausgegangenen gemeinsamen Forschungsarbeit bei der DaimlerChrysler AG), speziell
 - der Beweis des Theorems zur Berechnung eines minimalen Konflikts,
 - die Herleitung eines Algorithmus zur Berechnung aller minimalen Konflikte sowie
 - von Algorithmen zum Auffinden eines bzw. aller minimalen Erklärungen,
 - eine Darstellung des Zusammenhangs zwischen relationaler Aggregation und bekannten Dekompositionstechniken für Constraintprobleme und
 - die Vorstellung einer (partiellen) Normalform für Constraints in disjunktiver Normalform, die nachfolgende Projektionsschritte trivial macht und das Lösen der Variablen stark vereinfacht,
- die prototypische Umsetzung aller Algorithmen in der objekt-orientierten Programmiersprache Java und
- eine empirische Evaluation der zentralen Algorithmen anhand verschiedener Beispielp Probleme.

Die vorliegende Arbeit folgt einer strikten Trennung zwischen Modulen, die die grundlegenden Operationen *join* und *project* realisieren, und - basierend darauf - einer abstrakteren algorithmischen Schicht, welche Methoden zur Konsistenzprüfung, zum Lösen der Variablen, zum Erklären bzw. zur multi-kontextuellen Analyse implementiert.

Zum anderen wird streng zwischen dem theoretischen Teil und der praktischen Umsetzung in einer objekt-orientierten Programmiersprache unterschieden. Hierbei kann der letztere Teil als konkrete Implementierungshilfe für den vorgestellten relationalen Constraintlöser dienen.

Contact and Addresses

Prof. Dr. Georg Gottlob

Vienna University of Technology,
Computer Science Department,
Institute of Information Systems,
Database and Artificial Intelligence Group,
Favoritenstraße 9-11
Austria - 1040 Vienna
Gottlob@DBAI.TUWien.ac.at

Prof. Dr. Ulrich Geske

Fraunhofer-Gesellschaft,
Fraunhofer FIRST,
Institute of Computer Architecture and Software Engineering,
FIRST.5 PlanT
Kekuléstraße 7,
Germany - 12489 Berlin,
Ulrich.Geske@FIRST.Fraunhofer.de

Dipl.-Math. Frank Seelisch

Vienna University of Technology &
DaimlerChrysler AG,
Research, Information, and Communication,
Austauschgruppe (Knowledge Exchange Group),
Wilhelm-Runge-Straße 11,
Germany - 89081 Ulm,
Frank.Seelisch@DaimlerChrysler.com

Acknowledgements

I would like to thank Prof. Dr. Georg Gottlob and Prof. Dr. Ulrich Geske for supervising my dissertation, and for working through previous versions of this thesis. Interesting discussions at their institutes gave me valuable input for my work.

A cooperation between the DaimlerChrysler Company and the research institute of Prof. Dr. Gottlob set a fruitful environment for investigations on aggregation strategies. Also in this context, the use of previous versions of the prototypic implementation of RCS resulted in several improvements of the Java implementation. I thank the DaimlerChrysler Company for providing me with the infrastructure for my graduation, first of all funding but also a well-equipped workplace. Studying in the Knowledge-based Engineering Group at DaimlerChrysler Research, Information, and Communication in Berlin was very instructive and enjoyable.

Special thanks go to Dr. Volker May, Dr. Mugur Tatar, Dr. Jakob Mauss, and Dr. Andreas Junghanns who have always helped me to set the focus on the important issues of my research topic. Furthermore, Ute John helped me to coordinate the administrative issues with the two practical aspects of my doctoral work, the implementational task and the consulting when it came to integrating the programmed solver into first applications inside the DaimlerChrysler company.

Peter Hamann and Jan Laube supported the programming task by testing and bug-fixing; I am very grateful to them.

Last but not least, I would also like to thank my family - first of all my parents - and all those friends of mine who never stopped to give me mental support for this demanding project. I still remember many intensive talks which helped me to keep at it, and finally complete this dissertation.

Contents

1	Introduction	13
2	Constraint Solving in Engineering	19
2.1	Hierarchical System Models	19
2.1.1	Basic Concepts	19
2.1.2	Reusable Components and Constraints	20
2.1.3	Mathematical System Descriptions	21
2.2	System Synthesis & System Analysis	23
2.2.1	Tasks of System Synthesis	23
2.2.2	Tasks of System Analysis	26
2.3	Common Requirements	29
2.3.1	Multi-Contextuality	29
2.3.2	Single Context Analysis	30
2.3.3	Minimal Conflicts and Explanations	30
2.3.4	Rich Constraint Language	31
2.3.5	Uncertain Knowledge	32
2.4	Application of Existing Constraint Solvers	32
2.4.1	General Remarks	32
2.4.2	Detailed Discussion	34
2.4.3	Summary & Outlook on RCS	39
3	Relational Aggregation	42
3.1	Conventions & Notation	42
3.2	Preliminaries	43
3.2.1	Basic Entities	43
3.2.2	Operations on Relations	46
3.3	Solving Single Constraint Problems	50
3.3.1	Aggregation Trees	50
3.3.2	Deciding Consistency	53
3.3.3	Finding all Solutions	54
3.3.4	Providing Minimal Conflicts	57
3.3.5	Providing Minimal Explanations	64
3.4	Solving Multiple Constraint Problems	67
3.4.1	Contexts and Context Spaces	68
3.4.2	Identification of Reusable Aggregation Subtrees	69

3.5	Aggregation Strategies	72
3.5.1	On-The-Fly Strategies	73
3.5.2	Clustered Aggregation Strategies	75
3.5.3	Aggregation Strategies based on Decomposition Methods	76
3.5.4	Generic Aggregation Strategies	81
4	Relational Engine	85
4.1	Overview	85
4.1.1	Approximation of the Formal Framework	87
4.1.2	Soundness and Completeness	89
4.1.3	The Class Engine	90
4.1.4	Heap and AggController	92
4.1.5	Representation of Aggregation Forests	92
4.1.6	Representation of Context Spaces and Strategies	94
4.2	The Forward Phase	94
4.3	The Backward Phase	97
4.4	Minimal Conflicts and Explanations	100
4.4.1	Computing All Minimal Conflicts	100
4.4.2	Computing One Minimal Conflict	102
4.4.3	Providing Minimal Explanations	104
4.5	Analysis of Problem Sequences	106
4.5.1	Context Management	106
4.5.2	Reuse	106
4.6	Utilisation of Aggregation Strategies	109
5	Relational Processor	111
5.1	Overview	111
5.1.1	Services for a Relational Engine	113
5.2	How to Implement the Join Operator	115
5.2.1	Relations in Disjunctive Normal Form	115
5.2.2	Join and Or	117
5.3	Partially Solved Form	119
5.4	Implementation of Projection Based on the PSF	123
5.4.1	Projection	123
5.4.2	Establishing the Partially Solved Form	128
5.5	Managing Arithmetic Terms and Values	132
5.5.1	Overview	132
5.5.2	User-Defined Functions	134
5.5.3	The Representation of Arithmetic Terms	135
5.5.4	Issues of Computer Algebra	138
5.5.5	Operations on Arithmetic Terms	141
5.5.6	Interval Algebra	143
5.6	Constraint Language and Control Aspects	150
5.6.1	Control Facilities	151
5.6.2	Constraint Language	154

6	Experimental Results	160
6.1	Proving the Capabilities of the Prototype	160
6.1.1	A Showcase Example	160
6.1.2	Solving Some Non-Linear Problems	164
6.1.3	Trading Runtime for Accuracy	165
6.2	Selected Problem Families	166
6.2.1	The Family $(R_k)_{k \in \mathbb{N}_+}$	166
6.2.2	The Family $(D_k)_{k \in \mathbb{N}_+}$	168
6.2.3	ATV: A Real-World Application	169
6.3	Minimal Conflicts & Explanations	171
6.3.1	Constraint Suspension	172
6.3.2	Explaining Zero Bridge Currents	174
6.3.3	Conflicts and Explanations for the ATV Family	176
6.4	The Effect of Reuse along Similar Contexts	177
6.4.1	Two Small Context Spaces	178
6.4.2	1000 Contexts: The ATV Family	179
7	Conclusion and Future Work	181
7.1	Conclusion	181
7.2	Major Contributions	183
7.3	Future Work	185
A	Proofs	189
A.1	Lemma 1	189
A.2	Lemma 2	189
A.3	Definition 6	190
A.4	Lemma 3	191
A.5	Lemma 4	191
A.6	Lemma 5	193
A.7	Lemma 6	194
A.8	Lemma 7	196
A.9	Lemma 8	199
A.10	Lemma 9	200
A.11	Lemma 10	202
A.12	Invariants (4.6) and (4.7)	205
A.13	Lemma 11	207
A.14	Lemma 12	210
B	XML String Representations	211
B.1	The 1-Bit Full Adder	211
B.2	Context Space for the 1-Bit Full Adder	215
B.3	XML Strings for Electric Components	216
B.4	A Small Bus Communication Problem	220
B.5	An Electric Circuit with 32 Contexts	221
B.6	Quadratic Resistors	222

C Extensions	224
C.1 An Extension for Bus Communication	224
D Screenshots of the Implementation	226
D.1 A Showcase Example from Electrics	226
D.1.1 The Inspection of Relations	226
D.1.2 Building an Aggregation Forest	226
D.1.3 Obtaining Tightest Variable Restrictions	229
D.1.4 Retrieving a Minimal Explanation	230
D.2 Term Rewrites for the Basic Parallel Circuit	230
D.3 The Utilisation of Monotonous Terms	230
D.4 Minimal Conflicts for the 3-Queens Problem	232
D.5 Solving All Instances in a Context Space	233
E Structural Runtime Figures	236
E.1 Structural Data for $(R_k)_{k \in \mathbb{N}_+}$	236
E.2 Structural Data for $(D_k)_{k \in \mathbb{N}_+}$	236
E.3 Structural Data for the ATV Family	238

List of Figures

1.1	Overview Over This Dissertation's Major Contributions	13
2.1	A Hierarchical System Model of a Coolant Pump	20
2.2	A Simple Model of an Ohmic Resistor with Constraints	21
2.3	Some Basic Electric Components with Constraints	22
2.4	An Electric Circuit and its Mathematical Description	22
2.5	A Problem with Propagation	28
2.6	Derived Catalogue of Requirements	29
2.7	Two Families of Electric Circuits	36
2.8	Constraints for the Problem $D_k, k > 0$	37
3.1	A Graph Colouring Problem with Four Countries	48
3.2	Aggregation of Two Ohmic Resistors	50
3.3	An Aggregation Tree for an Electric Circuit	51
3.4	Explanation of Statements 3. and 4. of Lem. 6	59
3.5	Aggregation Tree for the 3-Queens Problem	62
3.6	Computation Scheme for Finding One Minimal Conflict in Fig. 3.5	63
3.7	All Four Conflict Sets for each Node in Fig. 3.5	63
3.8	Construction of Δ_x from Δ	67
3.9	Repairing the Aggregation Tree of Fig. 3.3 by Reusing Subtrees	70
3.10	Additional Recomputation due to Altered Scopes	71
3.11	One Aggregation Step While Building an Aggregation Forest	72
3.12	An Aggregation Step With Scope Sizes	74
3.13	A Clustered Strategy for the Tree in Fig. 3.3	76
3.14	Hypergraph and Decomposition for the Families in Fig. 2.7	78
3.15	Clustered Strategy Derived from the Decomposition in Fig. 3.14	79
3.16	A Generic Aggregation Strategy for Γ	82
3.17	Illustration of the Scope Property for y in Fig. 3.16	83
4.1	UML Diagram of a Relational Engine	86
4.2	The Relational Engine seen as a Finite State Machine	91
4.3	Pseudo-Code for the Forward Phase	95
4.4	Pseudo-Code for the Backward Phase	98
4.5	A Local View on the Forward and the Backward Phase	99
4.6	Pseudo-Code for Computing All Minimal Conflicts	101
4.7	Pseudo-Code for Computing One Minimal Conflict	103
4.8	Pseudo-Code for Providing One or All Minimal Explanations	104

4.9	Pseudo-Code for Collecting Reusable Subtrees	107
4.10	Pseudo-Code for Processing a Generic Aggregation Strategy	110
5.1	UML Diagram of a Relational Processor	112
5.2	The Bulb Constraint of Fig. 2.3 in Disjunctive Normal Form	115
5.3	Combination of Two Relations via Join and Or	117
5.4	Advantage of the PSF in the RCS Framework	122
5.5	Elimination of Basic Variables from a Conjunction	125
5.6	Pseudo-Code for Eliminating a Set of Basic Variables	126
5.7	Pseudo-Code for Computing Projections	128
5.8	Pseudo-Code for Making a Set of Goal Variables Basic	129
5.9	UML Diagram with the Architecture of Real-Valued Terms	133
5.10	Building a Term by Maximal Reuse of Identical Subterms	136
5.11	Simplification of the Term in Fig. 5.10 for $y = \frac{\pi}{2}$	139
5.12	A Term Rewrite for the Reformulation of a Quadratic Term	140
5.13	UML Diagram of Value Classes for Representing Subsets of \mathbb{R}	143
5.14	An Extended Arithmetic Constraint that is a Linear Band	145
5.15	A Typical Problem with Interval Algebra	146
5.16	The Relationship between Term Depth and Overestimation	148
6.1	An Electric Circuit with 32 Contexts	161
6.2	Increasing the Algebraic Degree - Quadratic Resistors	165
6.3	Runtimes for the Family $(R_k)_{k \in \{10, 20, \dots, 100\}}$	167
6.4	Scope Sizes for the Analysis of $(R_k)_{k \in \{10, 20, \dots, 100\}}$	167
6.5	Runtimes for the Family $(D_k)_{k \in \{1, 2, \dots, 10\}}$	168
6.6	Scope Sizes for the Analysis of $(D_k)_{k \in \{1, 2, \dots, 10\}}$	169
6.7	Four Propulsion Systems with Increasing Redundancy	170
6.8	Runtimes for Non-Incrementally Solving the ATV Systems	171
6.9	Pseudo-Code for Explanatory Services Using Constraint Suspension	173
6.10	Explaining Zero Bridge Currents	175
6.11	Node Statistics for the Explanation of the Zero Bridge Current	176
6.12	Conflict Computation for the Four ATV Systems	176
6.13	Explanations for the Four ATV Systems	177
6.14	The Effect of Reuse for the 1-Bit Full Adder	178
6.15	The Effect of Reuse for the Example Circuit in Fig. 6.1	178
6.16	Runtimes for Incrementally Solving the ATV Systems	179
6.17	Comparison of Incremental and Non-Incremental ATV Runtimes	180
6.18	ATV Runtimes Acquired by MDS	180
A.1	The Trees Υ' and Υ'' for the Strategy Υ in Fig. 3.16	201
B.1	The 1-Bit Full Adder	211
C.1	Aggregation Tree for Two Computers Exchanging Messages	225
D.1	A Screenshot Taken During the Inspection of Constraints	227
D.2	An Aggregation Forest for the Circuit in Fig. 6.1	228

D.3	Solutions for All Unknowns	228
D.4	Solutions for the Altered Problem with the Bulb Being Lit	229
D.5	A Minimal Explanation of Size 4 for the Switch Being Closed	229
D.6	All Term Rewrites Applied to Solve a Simple Quadratic Circuit	230
D.7	Non-Monotonous Solutions for the Bridge Circuit with one Box	231
D.8	Monotonous Solutions for the Bridge Circuit with one Box	231
D.9	Both Minimal Conflicts for the 3-Queens Problem	232
D.10	Context Space for the 1-Bit Full Adder in Fig. B.1	233
D.11	Solution Table for the Context Space in Fig. D.10	234
E.1	Number of Atoms per Node; $(R_k)_{k \in \{10, 20, \dots, 100\}}$	236
E.2	Average Number of Atoms per Node; $(D_k)_{k \in \{1, 2, \dots, 10\}}$	237
E.3	Maximum Number of Atoms per Non-Leaf Node; $(D_k)_{k \in \{1, 2, \dots, 10\}}$	237
E.4	Average Number of Disjuncts per Node; $(D_k)_{k \in \{1, 2, \dots, 10\}}$	238
E.5	Maximum Number of Disjuncts per Non-Leaf Node; $(D_k)_{k \in \{1, 2, \dots, 10\}}$	238
E.6	Structural Data for the Analysis of the ATV Systems	239

Chapter 1

Introduction

The main goal of this work is to contribute to and support the further automation of standard and non-standard engineering tasks by proposing a *novel concise framework for constraint solving*, the so-called *Relational Constraint Solver (RCS)*.¹ It supports classical tasks, as *deciding consistency*, *finding all solutions* and *providing minimal conflicts*, as well as tasks that have not yet gained a similar recognition: *providing explanations for all findings* and the *efficient analysis of problem spaces* as opposed to single problem instances.

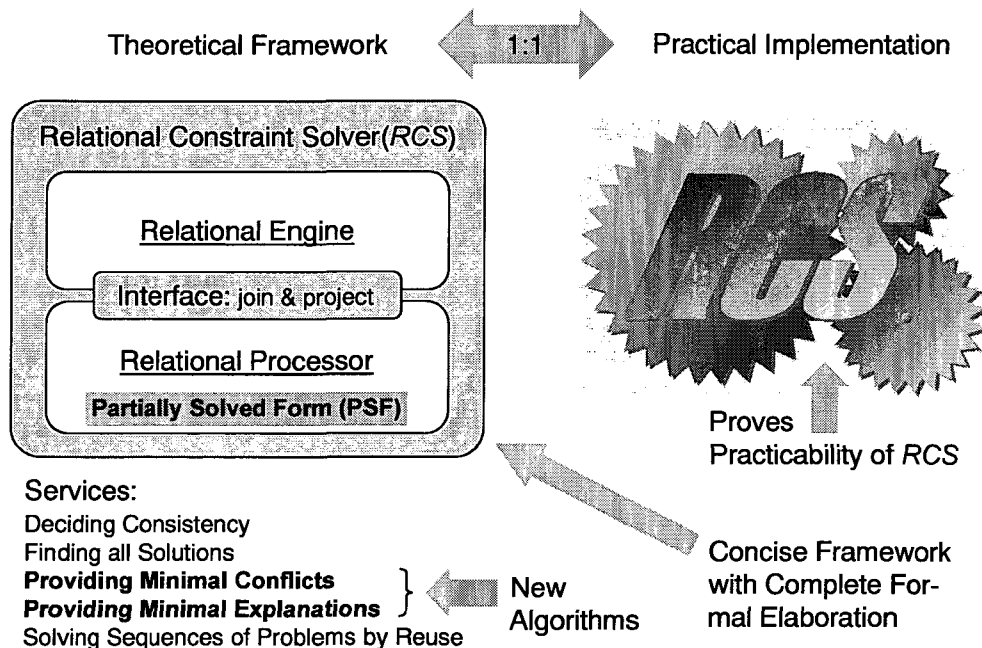


Figure 1.1: Overview Over This Dissertation's Major Contributions

¹Throughout this thesis, the distinction between the theoretical framework of RCS and its practical realisation by means of a computer program shall be made very clear. So the term RCS is to identify the former, whereas the latter will be noted by "prototype" or "prototypic implementation".

A special focus is going to be on *model-based* engineering applications, a field that has significantly gained in importance over the last decade. The major contributions of this dissertation, as partly shown in Fig. 1.1, are

- a requirements analysis for constraint solving in engineering applications,
- a thorough theoretical elaboration of a relational framework for solving heterogeneous constraint problems, based upon joint work, especially
 - the proof of a theorem concerning the computation of one minimal conflict, which had been hypothesised in the course of joint work,
 - an algorithm for finding all minimal conflicts, and its proof,
 - algorithms for finding one and all minimal explanations, respectively, and their proofs,
 - the relationship to decomposition techniques and
 - the development of the partially solved form,
- the practical realisation of all theoretical concepts by means of a prototypic implementation in Java and
- an empirical evaluation on a diverse set of problem instances.

In our highly industrialised world we are currently witnessing two antagonistic trends. On the one hand, products as well as processes tend to become more and more complex. A competitive engineering solution will usually involve a variety of leading-edge solutions coming from distinct engineering domains. Contrariwise, engineers and scientists have nowadays become specialists who have been educated and trained in a particular field of work. This calls for efficient tools that help aid complex processes and automatise the design, production and maintenance of desirable products.

In order to master those increasingly difficult engineering tasks, we need to manage our knowledge and formalise it so that it can be processed by machines and computers. Seen from this point of view, *constraints* are a highly formalised form of knowledge that can be reused, analysed, manipulated, and rearranged by computers. *Constraint solvers* are a means of realising this idea.

In this work, a new kind of constraint solver, RCS, is going to be presented, that incorporates new ideas and those already realised in existing solvers. It has been implemented in Java and is currently being used as a constraint server in some first engineering software clients that deploy constraint solving.

The programming language Java has been chosen for different reasons. First of all, it has nowadays become a popular object-oriented language which suits the modular structure of RCS. There exist good development environments for distributed programming based upon shared code repositories. And finally, Java provides useful facilities for building web-based applications.²

²This point is especially interesting when we think of providing web-services, e.g. for diagnosis, which will later turn out as an important client that may utilise a suitable software module implementing RCS, as a server for its constraint solving tasks.

RCS has been especially designed to meet a certain catalogue of engineering requirements, and exploits a characteristic common to many constraint problems arising from model-based engineering applications, the *low density assumption*: Each variable appears in only few constraints. This suggests the usage of variable elimination techniques in order to gradually simplify any given constraint problem.

A line of distinction can be drawn between existing constraint solvers and RCS: Most existing solvers tend to be applicable only to a certain class of constraint problems, such as finite domain constraints (for example in scheduling applications) or systems of linear equations and inequalities. RCS has been built to cover a wider range of constraint problems, although this may sometimes be at the price of incompleteness. Those constraints will usually involve finite domain variables, real-valued ones, and even variables that take as values structured objects. Furthermore, constraints understood by RCS can be equations, disequations and inequalities. Referring to this band width, RCS is said to be capable of dealing with *heterogeneous constraints*. Concerning arithmetic constraints, RCS allows also for non-linear constraints. As opposed to other solvers which would basically postpone all non-linear arithmetic constraints until they have simplified to linear ones, RCS is able to symbolically manipulate non-linear constraints. This is supported by a special representation for constraints in disjunctive normal, called *partially solved form (PSF)*, which can be seen as an extension of known normal forms for arithmetic constraints.

Also, in contrast to most existing implementations, RCS is, to a degree, capable of dealing with *uncertain knowledge*. This means the ability to process sets of possible value assignments rather than just single values. For real-valued variables, the special instantiation of that notion is interval arithmetic. Important engineering tasks require us to impose interval bounds on a certain variable since this might be the only piece of information available. We might come across that situation when designing a new artifact, where we may want to express certain tolerances of a part, a subcomponent, a feature or a parameter.

Apart from heterogeneity and uncertainty, RCS addresses conventional basic services such as *deciding consistency* of a given problem instance, *finding all solutions* in case of consistency, and otherwise providing *minimal conflicts*. The algorithm used to extract minimal conflicts in RCS is new. A small extension of it allows also for the retrieval of *minimal proofs* for the solutions of a variable computed by RCS. This function may be used to build explanation services which can help prevent the user from getting lost in too many details. The need for such explanation facilities has been stressed for long, and is currently gaining increasing recognition by researchers of the constraint community.

Another requirement supported by RCS is what may be referred to as *multi-contextuality*. In this dissertation, *model-based diagnosis* will serve as a prominent engineering application in which multi-contextuality plays an important role. Diagnosis is a special task of system analysis in the context of product maintenance, where we need to solve large numbers of constraint problems rather than just a single one. Due to issues of runtime and space consumption, it is hereby essential and inevitable for any underlying constraint solver, to enable the reuse of previous computations. Thanks to the similarity of the problem instances to be investigated during diagnosis, the potential for reuse is in general very high.

Whereas other built-in solvers of diagnosis tools make use of truth maintenance systems (also for deriving explanations), RCS deploys reuse based on so-called *aggregation trees*. Since constructing a forest of aggregation trees is anyway the initial step when analysing a constraint problem, RCS has almost no overhead when identifying reusable computations and generating explanations. Thus, updating effort for otherwise necessary maintenance modules can be saved.

Aggregation trees constitute the core concept of the Relational Constraint Solver. Although this concept is not new (cf. [58]), RCS is the first realisation of a constraint solver that covers all above requirements exclusively by means of constructing aggregation trees. Viewed from a software engineering perspective, RCS consists of two main layers: the *relational engine* and the *relational processor*. Whereas the latter provides the operators *join* and *project*, the former encapsulates all high-level algorithms that make sole use of those two operators. This strict separation will ease code management in any object-oriented programming environment. Also, it provides a good infrastructure for comparing alternative high-level algorithms regardless of the respective realisation of the join and project operators. In other words, one may examine alternative relational engines *modulo* the underlying relational processor.

Aggregation, the process of building an aggregation tree, is controlled by a so-called *aggregation strategy*. This might be produced either on the fly, or provided by some preprocessing strategy module that analyses the respective *constraint hypergraph* using known decomposition techniques, and, based on that, proposes a certain plan of aggregation steps. Albeit the latter kind of strategy must - in most cases - stick to purely topological properties of the constraint hypergraph, any on-the-fly strategy can also take into account the actual data encoded in the respective constraints. To a degree, this comparison will also be undertaken in this thesis.

RCS can be seen as a new constraint solver that combines well-known approaches with new algorithms. It will be shown to have a greater applicability than most existing solvers. However, when applied to pure problem instances, i.e. to those that are made up of constraints of one particular type, the figures prove the inferiority of RCS, and thus the superiority of tailor-made constraint solvers for the particular problem class at hand. Nevertheless, the prototypic implementation produces good results proving the practicability of the relational approach; a fact that is also supported by ongoing applications of RCS. Beyond the status of implementation presented in this dissertation, RCS shall be developed further, according to the requirements that are certainly going to be set by new and more demanding engineering applications.

Synopsis

Chapter 2 gives an overview over some common, well-known features of model-based engineering applications, and how constraint solving can contribute to handle them efficiently. The fundamental ideas of hierarchical system models, components, ports, and assemblies will be recapitulated. *Diagnosis* is to serve as an example for model-based system analysis. Examples of diagnosis applications will be used as well as other problems to clarify formal work and illustrate algorithms. The derived require-

ment catalogue for constraint solving in model-based engineering is however mainly due to diagnosis. Chapter 2 sketches also briefly matters of constraint solving for system synthesis tasks, such as configuration and reconfiguration, but some of the corresponding requirements will not be paid attention to in subsequent chapters. Still, the typical requirements set by diagnosis suffice to show the shortcomings of existing solvers, and motivate the development of RCS.

In Chap. 3, a formalisation of *relational aggregation* is presented, starting from basic concepts and entities, and proceeding to the important high-level algorithms situated in the *relational engine* layer of RCS, cf. Fig. 1.1. Here we are also going to state three theorems concerning the new algorithms for retrieving minimal conflicts and explanations. The chapter elaborates furthermore all relevant concepts for multi-contextual analysis and the reuse facilities of RCS, as well as aggregation strategies.

Chapter 3 states the main results of this dissertation, and provides detailed proofs to the respective lemmas and theorems.

The following Chap. 4 is dedicated to the high-level architecture of a concrete implementation of RCS as it has actually been carried out in Java, in order to verify the theoretical results of this dissertation. That *relational engine* assumes serviceable join and project operators for relations, which are pointed out, by Fig. 1.1, as the interface to the lower implementational level of a *relational processor*.

Chapter 4 includes investigations concerning the complexity of most of the presented pseudo-code.

The thorough elaboration of the two core operators *join* and *project* is the subject of Chap. 5. The corresponding layer of our Java implementation is called *relational processor*. It deploys a *partially solved form (PSF)* for constraints in disjunctive normal form, cf. Fig. 1.1. Properties of that form which is strongly related to well-known normal forms, are going to be stated, explained and proved.

The building blocks of arithmetic constraints, as combined by *join* and affected by *project*, are arithmetic terms. Consequently, this chapter also deals with arithmetic terms and issues of their automated simplification using so-called term rewrites. The chapter ends with an overview over the constraint language supported and processed by our prototype.

Empirical results derived with the prototypic implementation of RCS will be presented in Chap. 6. Their primary goal is to prove the practicability of our prototype and hence of the framework of RCS. To this end, a real-world problem is going to be analysed, that arises from a simplified model of a small unmanned spaceship, the so-called *automated transfer vehicle*.

Besides those experiments, there are several groups of additional measurements for evaluating the reuse facilities of RCS, and trading off runtime for computational accuracy.

The structural decomposition of RCS into two layers enables us to test alternative high-level implementations that rely on the same relational processor, see Fig. 1.1. We will utilise that feature to compare our new algorithms for deriving minimal conflicts and explanations with naive constraint suspension.

After concluding, items of future work are going to be sketched in Chap. 7.

Chapter 2

Constraint Solving for Applications in Model-based Engineering

After explaining hierarchical system models, and showing where - in this framework - constraints enter the scene, light will be thrown on two major engineering tasks, system synthesis and system analysis. The main focus will however be on the latter, with diagnosis as a prominent example. As will turn out, one can identify a certain catalogue of common requirements. Having collected those requirements, a new kind of constraint solver, the relational constraint solver (RCS), will be motivated.

2.1 Hierarchical System Models

2.1.1 Basic Concepts

Nowadays, many software tools used in engineering environments make use of hierarchical system models as representation of physical artifacts. Engineers think of a technical device as composed of sub-systems in a recursive manner. The terminal elements of such a tree-shaped assembly are basic parts which are usually called *components*¹. This basic concept is captured by the notion of a *part-of tree*. Many copies of the same component might occur severalfold in a large number of different engineered devices.

Figure 2.1 depicts - besides a photograph - an abstract view of a coolant pump as it is used in cars. The casing holds subsystems realising drive, suspension and the actual delivery of the fluid. The below tree shows also the next refinement of the part-of tree.

Device-specific functions of an assembled physical system are realised by a certain way of connecting components and subsystems. For this, components are modelled to have *ports*, see Fig. 2.1. Those are sometimes also called *terminals*, cf. [34] which provides a good overview over the fundamental concepts of *model-based diagnosis*

¹Unless ambiguous, this term will be used for both the physical component and the digital component model.

and introduces most of the terminology also used in this section; see especially [15]. In electrics, ports can be thought of as pins, otherwise they may be sockets, bushings or the like. A *connector*, i.e. a wire, hose, pipe, or shaft, will always link exactly two components by identifying one port of the first component with one of the second. A component model will usually comprise some specific piece of information modelling the very nature of that component. Sometimes we might also want to attach information of that sort to a connector, e.g. stating that a pipe may be leaking with a certain probability, or that an electric wire may be broken. It is however easy to see that we can get by with “plain” connectors, i.e. those that comprise no additional information. In order to do that, we just need to introduce one new component for each non-trivial connector, cf. the model of a non-trivial electric wire in Fig. 2.3. Being prepared like that, it suffices to use only “plain” connectors to model the entire physical system.

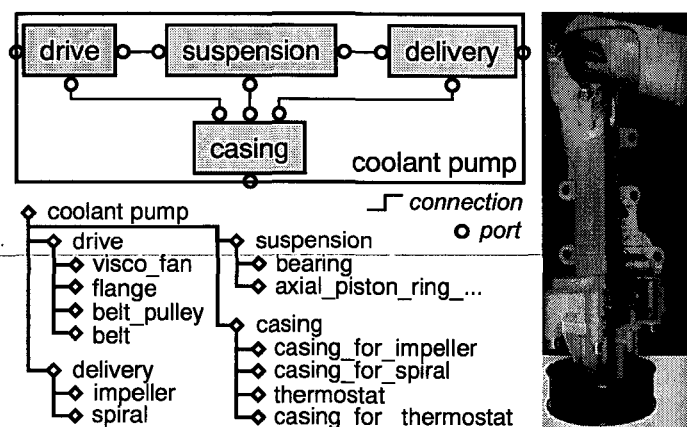


Figure 2.1: A Hierarchical System Model of a Coolant Pump

Following the outlined procedure and summarising, modelling a system basically boils down to modelling all components, and placing “plain” connectors accordingly. The major advantage is, of course, that components, modelled once and stored in a component library, can be reused in an arbitrary number of system models, cf. [24]. This will eventually justify the initial modelling effort, and reduce the cost of future modelling tasks remarkably.

2.1.2 Reusable Components and Constraints

However, in order to guarantee the reusability of a component model, it is crucial that all information attached to it be *context-free*. Context-free information attached to a component representing an electric resistor could for instance be a mathematical constraint capturing Ohm’s law. But in terms of which variables is this constraint going to be stated?

The answer to this question as well as to how to establish a context-free formulation lies in the ports of a component. Associated with each port, there will be a set of

variables; one for each item that needs to be transmitted by any linking connector. For instance, in an electric circuit one can think of a wire as transmitting an electric current and propagating a voltage. Thus there should be exactly two variables associated with each electric port, as the pairs of variables (i_k, v_k) , $k \in \{1, 2\}$, in Fig. 2.2. By attaching additional, component-specific *internal variables*, as the resistance R ,

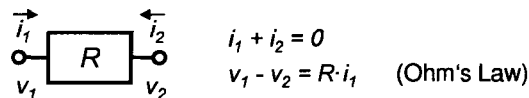


Figure 2.2: A Simple Model of an Ohmic Resistor with Constraints

we can then state a component's physical behaviour as exemplified in Fig. 2.2.² Internal variables can be seen as *component parameters*, thus allowing the modelling engineer to define a whole class of similar components. It is intuitively clear that the set of constraints attached to a component model should always be consistent, and that the set of involved variables will in general be underconstrained, i.e. allow for more than just one solution, due to the presence of parameters and the intended degree of freedom of the component's behaviour.

It is easy to check that the constraints in Fig. 2.2 are oblivious to permuting the port identifiers. We expect this always to be so when a component's behaviour does not depend on its orientation with respect to the modelled system. An electric diode is, in that sense, an example for an oriented component. Here, a turn-over would in general imply a different physical behaviour of the modelled electric circuit.

2.1.3 Mathematical System Descriptions

Let us suppose we are given a library of reusable component models, and that the physical behaviour of each component has been attached as a set of context-free constraints in terms of internal variables and port variables.

As an example, Fig. 2.3 presents models of the electric components bulb, ideal voltage source, wire, ground and Kirchhoff node, including constraints that reflect their physical behaviour. The bulb model includes a threshold current, C , that needs to be superseded in order to lit the bulb, i.e. to ensure $L = \text{on}$. Note that the constraints may also involve disjunctions, modelling alternative possible behaviours like a *nominal* and one or even more *fault* behaviours. In Fig. 2.3, each disjunct refers to a so-called *mode* variable, M , that takes its values in a finite domain of symbols. Here, this domain is the set $\{\text{ok}, \text{broken}\}$. For a detailed discussion of the pros and cons of modelling *behavioural modes*, see [34], especially [17] on explicit models for distinct modes of component failure.

It is then clear how to derive a mathematical description of any system that we might model using those component models: We simply have to collect, for each component, an instantiation of its constraints, and add all identification constraints

²Note that a current is a directed value. We should thus commit once and for all to having directed values always pointing inward with respect to the modelled component.

imposed by the connections in the system. This is exemplified in Fig. 2.4 which makes use of the components of Fig. 2.3: The conjunctive constraint in the grey box is due to the connection between port 2 of node $N1$ and port 1 of bulb $B2$. The other six constraints of that kind, are not presented in the figure. All other constraints are instantiations of the constraints in Fig. 2.3. For example, $\text{Source}^{\text{SRC}}$ stands

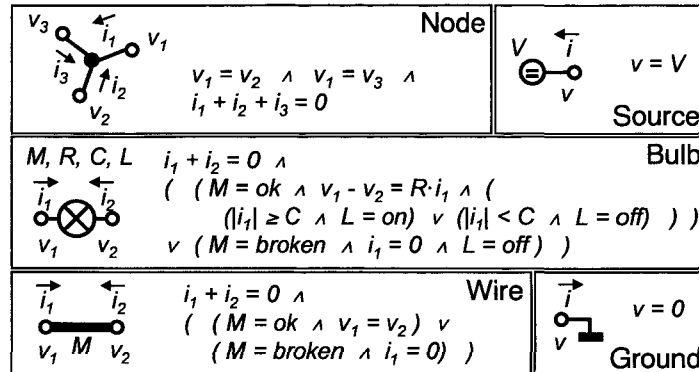


Figure 2.3: Some Basic Electric Components with Constraints

for the constraint of an ideal voltage source with each variable top-indexed by the component's name "SRC", producing $v^{\text{SRC}} = V^{\text{SRC}}$. Hence, instantiation is done by renaming all variables. Obviously, the process of collecting all system-relevant constraints can be automatised.

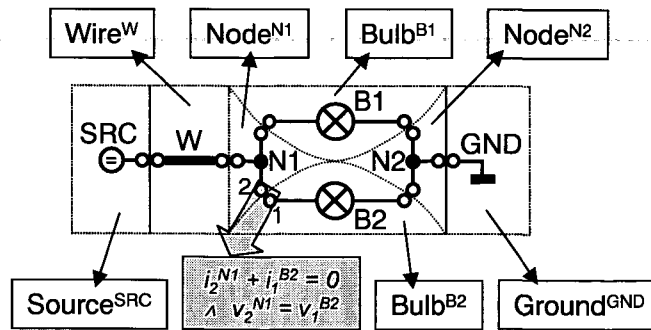


Figure 2.4: An Electric Circuit and its Mathematical Description

We will thus finally end up with a set of constraints that provide a mathematical description of our modelled system. The complexity and accuracy of this description will directly depend on the component models.

Mathematical system descriptions are the starting point that naturally suggests the use of constraint solving methods in order to automatise and computer-aid the synthesis and analysis of technical systems.

2.2 System Synthesis & System Analysis

Along the so-called *product life cycle*, there are various tasks of model-based engineering. The overall idea, sought after by the discipline of *engineering knowledge management*, is to support all those engineering tasks preferably by the same digital component and system models.

A great deal of research has been done in the last few years on design-supporting methods, cf. [21], on software tools, and on enhanced configuration and reconfiguration facilities. See e.g. [48] for a focused view on the connection to constraint solving. It has become obvious how crucially important the early phases of the product life cycle are, in order to cut costs for all subsequent phases. This insight also contributed to the push of the emerging subject of engineering knowledge management. [19] contains a collection of publications concerned with knowledge management for model-based engineering, and covers a spectrum of relevant tasks ranging from design to product maintenance issues.

Although engineering knowledge management suggests the overall applicability and compatibility of digital models, the reality looks somewhat different. For historical reasons, most tasks along the product life cycle happen to find support in one or even more specific software tools. In general, the respective data are syntactically as well as semantically incompatible. Moreover, those tools will usually contain their own problem-solving and constraint-processing software modules. These will therefore usually make assumptions about the problem instances at hand. Certain limitations, concerning the applicability of the solver modules, are the logical consequence.

The next subsection is to give a brief summary of the early phases of the product life cycle, including tasks as *specification-driven design*, *configuration* and *reconfiguration*. As will become clear, the latter two involve optimisation which is neither in the scope of RCS, nor of this dissertation.

Secondly, *diagnosis* is discussed in more detail, which provides in practice valuable services in the field of *product maintenance*.

2.2.1 Specification-driven Design, Configuration and Reconfiguration

There are basically two main ideas when talking about specification-driven design on the one hand, and (re-)configuration on the other.

Specification-driven design assumes some initial system design, based upon the reuse of previous, approved designs. It hence deals mainly with *parameter adjustments* in order to satisfy a set of user requirements. The typical situation would be that the designer is provided with a set of parameter tolerances, e.g. given as intervals, and has to find a consistent parameter instantiation. Additionally, there may exist preferences for a subset of parameters, often expressed in terms of probabilities.

As to configuration and reconfiguration, the situation is different in that now the focus is no longer on a parametric yet otherwise fixed assembly, but on the usually vast space of all possible assemblies. This time, the task encloses an optimisation

problem, namely to pick an *optimal variant of a system*, subject to a set of one or more objective functions.³ The search space relates basically to the cartesian product of all component alternatives. These are usually organised in so-called *taxonomies*, sometimes also called *kind-of trees*. Naturally, here is a great similarity to the object-oriented approach in software engineering.

Most configuration tools will, for reasons of efficiency, decide the consistency of a candidate system design only during the process of optimisation itself. Obviously, deciding consistency can be accomplished by similar methods as in the case of specification-driven design. In the dissertation [48], configuration and reconfiguration are looked at more closely; the most important methodic frameworks and tools are discussed. It states *rule-based*, *case-based* and *constraint-based problem solving*, and *genetic algorithms* as the main approaches to (re-)configuration.

[48, Sect. 5.5] elaborates a method for designing new components, by deriving necessary conditions from the set of constraints that have been imposed for the entire system.

An additional view on the outlined problem can be found in [3] which also characterises configuration as a necessarily interactive task since requirements may become explicit only during modelling and actually solving the problem. This work also stresses the need to provide advanced explanation facilities to guide the user's backtracking when approaching an inconsistent system configuration. The proposal is here a so-called Assumption-based Constraint Satisfaction Problem (A-CSP), based on *Assumption-based truth maintenance systems (ATMS)*; see also the next subsection on tasks of system analysis. Furthermore, [3] discusses a precompilation of the solution space, in order to guarantee acceptable response times in the range of a few seconds.

The widespread design tool CATIA is used mainly for specification-driven design. Its built-in constraint solver addresses, first of all, geometric conditions, and detects overlapping or interpenetration of parts; see [8]. Here, the designer will strongly depend on the solver as it is. He is to think of the built-in constraint solving facilities in rather abstract terms, as for instance imposing a hole to be bored "in the middle of a steel plate". He will not be bothered with mathematical formalisations of those conditions. On the other hand, and as mentioned before, this puts certain limitations on the applicability of the built-in constraint solver to other tasks, and consequently this is also not intended.

Latest versions of CATIA deploy tools, that allow also to model interval bounds for selected geometric parameters, for example for the width of a car door and the matching frame in the car body. This is a facility which becomes more and more important in the context of *tolerance management*: Decreasing tolerances is a means to ensure quality but at the same time considered very expensive.

One such supportive tool is 3DCS, [1]. Herein, the user can impose several prominent probability distributions for each mutable parameter, such as Gaussian, Laplacian or a triangular distribution. After having modelled several parts in this manner,

³Engineers will in fact often be happy to find just a sub-optimal design, due to the complexity of the problem.

3DCS can then compute probability distributions for the entire modelled artifact, e.g. the total length. In order to do that, 3DCS randomly chooses n system instances, i.e. n situations in the parameter space, according to the given probability distributions, where $n \in \mathbb{N}$ is a large number provided by the user. After that, 3DCS solves all dependent variables in all n situations. The results yield the sought-after distribution. Additionally, 3DCS provides percentages for each individual parameter distribution, telling to what degree it affects the overall distribution. Unfortunately, the interpretation of those data is not clear and turned out illogical for rather small test cases.

In [54], the coolant pump of Fig. 2.1 is used as an illustrating example to point out requirements of specification-driven design on a less geometric and more functional level.⁴ Most of the stated requirements can be mapped to specific requirements for constraint solving. Among others, those requirements are the ability to add/remove a set of constraints (in order to investigate alternative component variants), and - in connection to that - to reuse previous computations and partial solutions corresponding to consistent subsystem designs. Again, the ability to reason with intervals of parameters, and - more generally - with sets of possible values rather than just singleton values, is pointed out.

The constraint management in the iViP tool allows for an investigation on the consistency of a design by means of propagation. This introduces difficulties with cyclic constraint problems, i.e. with problems that have a cyclic *constraint hypergraph*, see [55, p. 448] or [2, p. 130]). That well-known problem is exemplified in Fig. 2.5 for a small example with arithmetic constraints. (The crux is elaborated in the following subsection.)

Apart from that, the built-in solver of the iViP tool is not incremental in that it does not deploy reuse as requested above. Nor does it provide a good support for the processing of parameter intervals.

What seems to be a common characteristic of most built-in solvers in software tools supporting specification-driven design, is that constraint solving works by propagation. The application of known constraint solving techniques is hence restricted to those used in simulation tools, which becomes especially evident in the iViP tool.⁵ Consequently, when working with multi-state systems in which the topology, the direction of flows, or other causal dependencies may change due to closing / opening switches, relay coils, brakes or clutches, propagation-like constraint solvers will only work if the designer provides one system model for each system state. Clearly, for complex applications, this will in general fail to be workable.

Another design-supporting software prototype that focusses more on functional dependencies, is the System Design for Reusability (SDR). It stresses the concept of part-of trees and model-based system design; for a conceptual look at SDR, see [25], or [4] which uses again the coolant pump in Fig. 2.1 as a pilot application.

In SDR, constraint processing provides conflict detection and solving variables, respectively, by means of the well-known *Waltz algorithm*. It works by a more sophisticated form of propagation. In [27] the reader can find a short description of

⁴[54] resulted from the German research project "Integrated Virtual Product Creation (iViP)" with over 40 industrial partners.

⁵Moreover, not all relevant paths of propagation will necessarily be investigated.

the *Waltzer constraint engine*, and its application to some classical problems. The Waltz algorithm may also encounter difficulties with cyclic constraint problems since termination conditions are hard to define and will usually be problem-specific. This becomes even harder to resolve when there are parameter intervals in the constraints: The algorithm might narrow those intervals. But how to define sensible generic termination conditions in terms of changes imposed on those intervals? Especially for the coolant pump application, the built-in Waltzer constraint engine has, due to those reasons, not consequently been used: For certain layout subtasks, SDR uses tailor-made, problem-specific computation procedures instead, which need to be re-run several times in order to adjust and synchronise parameters.

In SDR, the designer may postpone refinements of abstract subsystems. For example, the decision which actual realisation of a drive he prefers to use in the new collant pump, need not be made right away. There are constraints that hold for the abstract concept of a drive. Reasoning with those generic constraints might already exclude numerous choices for some of the components. This is a means to prune the normally very large spaces of design variants. In this respect, SDR goes beyond specification-driven design, and deploys also simple methods of (re-)configuration. Once the user specifies an actual drive, for instance by choosing the model of an electric motor from a taxonomic library, particular constraints describing its physical behaviour will be added to the pool of active constraints that have to be considered by SDR.

2.2.2 Tasks of System Analysis

Whereas in design and (re-)configuration the challenge comes - besides optimisation - from the vast spaces of variants, it comes in analysis tasks, such as diagnosis, from the numerous potential states in which a system may be. Figure 2.4 already presents a small system with $2^3 = 8$ states: The wire and both bulbs are modelled to either work or fail. For a complex diagnosis application with n components each of which may behave according to m alternative modes, this gives m^n system states.

In model-based diagnosis, we are given a system description, sometimes also called *background theory*, as for example the set of constraints in Fig. 2.4. Furthermore, observations can be made, e.g. bulbs found to be lit or not. The task is then to provide one or more *diagnoses*, i.e. vectors of behavioural modes for all components, consistent with the system description and sufficient to justify all observations. For instance, the observation that $B1$ is not lit (in spite of a sufficiently small threshold C^{B1}) can be justified by the *single-fault* diagnoses $M^{B1} = broken$, and $M^W = broken$. The latter suffices also to explain why neither $B1$ nor $B2$ is lit, being the only single-fault diagnosis for that observation. $L^{B1} = L^{B2} = off$ may furthermore be justified by the *double-fault* diagnosis $M^{B1} = M^{B2} = broken$. This is however, in practice the far less probable situation; hence good diagnosis tools allow to attach probabilities to a component's behavioural modes, and can produce diagnoses according to their probabilities rather than their cardinalities.

An exhaustive introduction to the field of diagnosis can be found, for example, in [34]. In [15] a very concise summary of model-based diagnosis and alternative approaches

to diagnosis are presented. This work especially stresses the advantages of *model-based diagnosis* as opposed to diagnostics, fault dictionaries, rule-based systems, and decision trees. One advantage is that most other approaches owe their ability to the incorporation of application-specific and problem-specific knowledge. This will have a negative effect whenever the diagnosed system is altered since underlying knowledge bases will need revision. Also, as is especially the case for decision trees and rule-based systems, other approaches tend to lack transparency, and knowledge revision will become even harder. Maintaining a diagnosis application along all minor and major system changes is altogether going to cost more.

Another advantage is gained due to the fact that once an artifact had been designed in a model-based framework, diagnosis happens to be yet another application based on the digital model of that artifact.⁶

From the very beginning of model-based diagnosis until today, there have been great achievements. One example is the detection of multiple faults with a causal nature, see [68]. For instance, reporting a blown fuse will not satisfy the user, since replacing it will not fix the actual problem in an electric circuit. The user needs to be told where and why there had been an overcurrent. Such subtle multiple faults can only be diagnosed with a program that facilitates simulation and computations over a sequence of time slices.

The *Model-based Diagnosis System (MDS)*, is a software tool that is able to detect single faults as well as multiple faults which may even be of such a time-causal nature; see [57], and also [62] for the application of MDS to a tram lighting system in the context of augmented-reality-based service and training. MDS has been proved applicable to large-scale engineering applications with a few hundred components and a few thousand variables and constraints. It produces diagnoses based on a special form of constraint solving. A constraint needs to be modelled as a set of rules, one of each enabling the computation of one occurring variable provided all other variables have been assigned a value. The process of constraint processing itself is then performed in a rule-propagating manner. MDS keeps track of all computational dependencies using an ATMS. It is thus able to detect conflicts, which can be further minimised, and to reuse all unaffected computations when discarding previous assumptions. Truth maintenance systems (TMS) have been and are still very popular in diagnosis tools. In the dissertation thesis [69] relevant aspects of the application and implementation in MDS of *reason maintenance systems*, as for example ATMS, lazy ATMS, and justification-based TMS, are elaborated. A concise overview over TMS's can be found in [27].

Moreover, due to the variety of systems that have been diagnosed using MDS and, connected with that, thanks to the continuous development of it, MDS is today able to process very different sorts of values and constraints. Also, the transformation of constraints to rules makes user-defined code fit in the framework more easily.

However, MDS reveals problems and sometimes impractical runtimes with mutable parameters that are only known to lie within some interval. In order to make the rule-based constraint propagation work, those intervals must sometimes be split into sufficiently small ones, causing the original problem to branch into a large number

⁶Obviously, this fits nicely in the concept of engineering knowledge management that is to support a product life cycle without gaps.

of disjuncts. The rule-based approach holds thus the disadvantage that the modeller of those rules will sometimes have to know a great deal about the built-in constraint processing.

Figure 2.5 depicts another problem with pure propagation. Suppose, the constraint

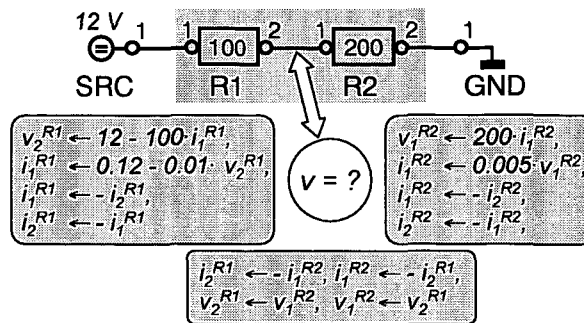


Figure 2.5: A Problem with Propagation

propagator is fed with the constraint rules describing the given electric circuit with an ideal voltage source of 12V, and two resistors of 100Ω and 200Ω in series. After propagating those constants as far as possible, a simple propagator would end up with the rule store shown in Fig. 2.5. The intermediate voltage $v_2^{R1} = v_1^{R2}$ is unknown, and the current can only be fixed after determining that voltage. In MDS this problem, which will obviously occur rather frequently, is resolved by the use of current-voltage-pairs and a bit of symbolical manipulation, which will clearly recover the situation in Fig. 2.5.

The design of MDS allows furthermore to tackle other analysis tasks:

- **Failure mode and effects analysis (FMEA):** Here, the user can simulate the effects of faulty or worn components on other components and - more general - on desired functions of the modelled system.
- **Test Proposal:** Given an unknown faulty component and some initial observations, MDS proposes a “best” measurement with highest discriminating power (according to costs and probabilities). After entering the measurement as an additional observation, the process is repeated until the faulty component can be identified.
- **Sensor Placement:** Here, MDS computes critical points in the system where additional measurement facilities would help to increase an otherwise poor or insufficient discriminating power during troubleshooting.

Mentionable work has been done on diagnosing tree-structured systems. This is mainly due to the fact that acyclicity seems to guarantee better time and space complexity than in the general case. ([67] suggests that linear time complexity, as proved there for single-fault diagnoses, may be preserved even in the case of finding all double-fault diagnoses.) [23] introduces an efficient algorithm, SAB, that works

by assigning costs for faulty components. Finding minimal diagnoses coincides then with minimising the overall cost. This work also compares SAB with MBD2 which goes back to *Reiter* and follows the standard approach of assuming a certain vector of behavioural modes for all components.

In [67] diagnoses are computed according to the algorithms TREE and TREE*, up to a given size. They work by recursion, seeding in components for which expected and actual behaviour differ. This work also presents a detailed comparison of TREE and TREE*, respectively, with SAB.

An interesting combination of the configuration domain and diagnosis can be found in [26]. Here, diagnosis is adopted to debug knowledge bases: Besides unachievable requirements, an inconsistent knowledge base, e.g. after alterations, can often be the reason for a configuration task to fail.

2.3 Common Requirements

This section is going to summarise the requirements for constraint-based specification-driven design and system analysis, derived from the above remarks and examples. Although there might be approaches other than constraint solving techniques, it had been made clear that modelling physical behaviour by constraints comes natural. In a highly complex engineering environment, it turns out expensive and - seen from the perspective of engineering knowledge management - disadvantageous to be forced, as

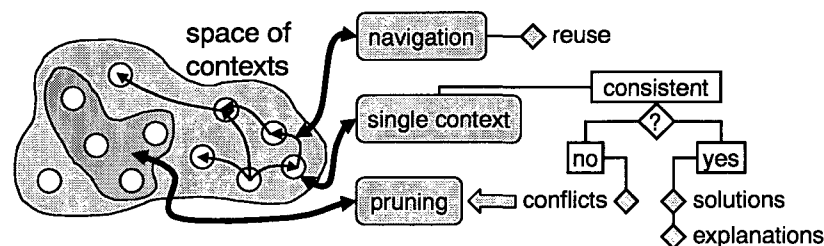


Figure 2.6: Derived Catalogue of Requirements

a modeller, to state a constraint in some tool-specific representation, as for example by a set of rules. Stating constraints in a more tool-independent, declarative way will help bridge the gap between an engineer's manner of thinking and the computer interfaces he needs to deal with. Constraints embody engineering knowledge of a very formal kind, enabling computers to efficiently aid today's engineering tasks. In Fig. 2.6 the most relevant top-level requirements and their interrelations are sketched. The following subsections are going to explain the overall picture and clarify the respective requirements.

2.3.1 Multi-Contextuality

First of all, the aim is to analyse entire spaces of distinct constraint problems rather than just single problem instances. It is worth mentioning that most constraint

solvers do not address this task, but concentrate rather on efficiently solving single problem instances. In Fig. 2.6, this *context space* is depicted as a two-dimensional area of distinct constraint problems called *contexts*, symbolised by white dots. As seen above, in design those spaces stem from the many variants one has to handle when laying out a new product using alternative subcomponents; in diagnosis a context is a certain state of the modelled system. Indeed, the larger the context space, the less important become the performance figures for the analysis of each single context. This argumentation assumes however that the average figures will remain good-natured which strongly depends on the “density” of the underlying context space: If one can find, for each context, a “neighbouring” one which differs in only few constraints, one should be able to cut the effort for the latter by means of reusing computations carried out for the former; see item “navigation” in Fig. 2.6. In order to accomplish an efficient analysis of the whole *context space*, it is thus a non-trivial task how to traverse it in order to maximise that reuse. However, that optimisation problem is not addressed in this dissertation.⁷

2.3.2 Single Context Analysis

A solver supporting model-based applications needs to be capable of accomplishing the classical tasks of *checking consistency*, and - in case of consistency - *finding solutions* for all variables. Confer to the centre item “single context” in Fig. 2.6.

Finding solutions will, in the presented setup, explicitly mean to find *all* solutions for all variables: For a consistency-enforcing tool that supports specification-driven design, the designer wishes to survey not only one but numerous consistent layouts for a new system, if not all.⁸ As for diagnosis, it is in fact not so important to find solutions, since producing diagnoses is mainly based on deciding consistency. But regarding for example FMEA, it should be clear that computing all solutions is essential in order to treat all possible effects of component failures on the modelled system.

Another aspect of single context analysis is the *trade-off between computational accuracy and runtime*: In order to arrive at acceptable response times of a solver, it should be able to gradually sacrifice certain levels of accuracy. Approximations and incomplete data processing are means to implement such trade-off facilities.

2.3.3 Minimal Conflicts and Explanations

The computation of conflicts, for any given inconsistent context is a rewarding measure to prune the context space: Other contexts that also refer to the conflicting set of constraints, can right away be deduced to be also inconsistent; see item “pruning” in Fig. 2.6. Therefore, those contexts need not be considered by the constraint solver, and their number will in general be the greater, the denser (in the sense of

⁷It should nevertheless be easy to define an appropriate metric, μ , for a context space, considering - besides maybe other characteristics - the number of differing constraints between two contexts. The task of maximising overall reuse relates then to computing an μ -shortest path that visits each context.

⁸The situation changes, of course, when there are cost functions, and the designer looks just for one (sub-)optimal layout; see Subsect. 2.2.1.

the above loose definition) the context space is. Considering again the typical size of a context space, it becomes worthwhile to not only compute small but even *minimal conflicts*.

For consistent constraint problem instances, the pendant of a minimal conflict is a minimal *explanation* or *proof* of the value restriction obtained for some variable. As will become clear, with relational aggregation, both tasks can indeed be tackled by the same algorithmic idea. The benefit of providing minimal explanations for the user is that he will be more likely to understand what is going on.

For example, after specifying a set of system parameters of a new coolant pump, the designer may learn that he can no longer use an electric motor as drive. He might want to enable this type of drive again by undoing some of his previous parameter settings. For this, he needs to take a look at an explanation, i.e. a minimised (if not minimal) set of parameter settings that enforce the exclusion of the electric motor. This will help him identify the right parameter setting(s) to be suspended, cf. again the work done in [3].

2.3.4 Rich Constraint Language

Figure 2.3 has already hinted that constraints, in order to express the physical behaviour(s) of a component, may involve *disjunctions* and *conjunctions* of *equations*, *disequations*⁹, and *inequalities*. The latter kind of binary relation makes sense only for domains where at least some partial ordering can be imposed. In this work it will be used exclusively in connection with the set of reals, \mathbb{R} . Apart from that, (dis-)equations can be formulated for terms taking values either in \mathbb{R} or in some finite domain of symbols, see for example the equations in Fig. 2.3 involving the mode variable M . (It should be clear, however, that variables and terms are not allowed to take their values in two mutually distinct domains, as for instance in \mathbb{R} and in some finite domain.)

Ohm's law, see Fig. 2.3, is an example for a non-linear arithmetic equation. A solver that is to serve engineering tasks needs to be able to process such non-linear constraints, not to mention linear ones.

Another important requirement in the context of engineering tasks is to enable the incorporation of *procedural constraints*. Those are usually provided by a piece of executable code that will, for a given vector of input values, produce an output value. Normally, the functional relation captured by such a procedure can, due to its complexity, not be formulated in terms of other language features provided by the constraint language. We are going to further clarify this requirement in Chap. 5, where we also present a possible and practicable approach as realised in our prototype.

In what follows, the term *heterogeneous constraints* will be used to refer to all the different types of constraints that have been mentioned.

In addition to logic and-or-junctions of atomic constraints, it is often more favourable, for reasons of clarity, to use implications. For example, in the case of the bulb in

⁹I will consequently stick to this term for reference to the binary relation $\neq \subseteq \Gamma \times \Gamma$ for any set $\Gamma \neq \emptyset$, where $(a, b) \in \neq \iff \neg(a = b)$

Fig. 2.3, the implication $M = ok \wedge |i_1| \geq C \implies L = on$ would be more intuitive. In general, the implication $c \implies d$, where c, d are boolean formulas, can be translated to $\neg c \vee d$. Thus, implication is a trivial language feature, provided that the underlying parser module is able to compute a representation of the negation of any given condition. The parser of RCS “understands” implications, but is able to produce such negations only for a certain subclass of atomic constraints. This is going to be discussed in Chap. 5.

In engineering, man-machine-interfaces and communication facilities play a more and more grave roll. Highly specialised experts from different domains need to combine their work by means of shared digital models. This sets challenging language requirements for any supporting software tool, including constraint-based tools for model-based engineering. Therefore, the aspect of a rich constraint language must not be underestimated.

2.3.5 Uncertain Knowledge

As already pointed out, it will often be the case that certain component parameters will only be known to lie within some interval. This might be due to fabrication tolerances, or - in case those parameters have been measured - to measuring errors. A well-conditioned constraint solver will yet have to deal with this sort of *vague, uncertain, or erroneous data*. Little attention has been paid to the constraint problems that typically arise; see e.g. [76] for an introduction. [5] deals with uncertainties in the context of configuration, presenting examples with parameter intervals.

As an easy example, regard again the circuit in Fig. 2.4. Suppose the bulbs’ resistances have been measured to lie within the interval $[1000 - \Delta, 1000 + \Delta] \Omega$, where $\Delta > 0$. Determining the electric current that flows to ground, i^{GND} , will depend on an evaluation of the overall resistance $R = (R^{B1} \cdot R^{B2}) / (R^{B1} + R^{B2})$. This reveals that managing uncertain parameters can naturally lead to optimisation problems. In the example, that problem consists of finding the minimum and maximum values of the function $f(x, y) = (x \cdot y) / (x + y)$ on the square $[1000 - \Delta, 1000 + \Delta]^2$. Although it is easy to see that, in the given setting, evaluating R by means of *naive interval arithmetic* produces tightest bounds, this method is known to be incomplete and will in general only yield supersets of the set of values that can actually be attained by the given arithmetic function. This issue will be discussed in Chap. 5.

More generally, the addressed requirement can be stated as the ability to simultaneously process and reason about sets of possible value assignments, rather than just singleton values. For arithmetic terms, this will, as seen above, often result in hard optimisation problems. For reasons of efficiency, those need to be avoided.

2.4 Application of Existing Constraint Solvers

2.4.1 General Remarks

The purpose of this section is to motivate and provide good arguments for the development of a new kind of constraint solver, namely RCS, in spite of the many

existing ones. The argumentation will be based on the above requirements. The respective shortcomings of existing solvers shall be pointed out. Before, some general remarks are worth mentioning.

Today, the constraint community can roughly be divided into researchers working on *constraint satisfaction* and *constraint solving / constraint programming* respectively. The former focus on problems defined over finite domain variables. The corresponding solvers are characterised by a high degree of maturity, and implement very efficient algorithms, for example for scheduling, task and job assignment problems, planning and propositional satisfiability, to name just a few. The methods for solving are normally composed of procedures from three categories: *problem reduction* for breaking a harder problem into a set of hopefully easier ones, *search* for finding one or all solution tuples of each of the subproblems and *solution synthesis* for combining the partial tuples to overall solution tuples. For a detailed yet concise introduction, see [70], from which also the presented categorisation has been taken.

On the other hand, constraint solving addresses problems that are naturally modelled using variables that take values in infinite domains, as e.g. \mathbb{N} , \mathbb{Z} , or \mathbb{R} . Constraint solving is a declarative paradigm that has been merged with *logic programming* to the field of *constraint logic programming (CLP)*, cf. [45] and [47]. An important goal in this context is to hide the respective solving methods from the user. Most arithmetic constraints in this context are linear equations and inequalities.

Although both domains have been subject to remarkable improvements and of great interest to many researchers, one tends to encounter problems when trying to model *hybrid constraint problems*, i.e. those that involve both finite- and infinite-domain variables. However, most engineering applications, like those arising from specification-driven design and diagnosis, will in general produce such hybrid problems. *We should therefore recognise engineering applications as a promotor for working on efficient hybrid solvers.* Thereby, we shall be able to combine the approved, successful concepts of constraint satisfaction and constraint solving.

It has become clear that a great many of hybrid constraint problems may be reformulated to fit in one of the two frameworks mentioned above. But the modelling effort can be immense and an engineer can, in general, not be expected to be trained for that task. Moreover, reformulations often tend to approximate or disguise the actual constraint problem, making the solving sometimes harder in terms of the complexity of the deployed algorithms. For a detailed discussion of issues concerning modelling vs. solving, see e.g. [20].

Besides the need for efficient hybrid constraint solvers, it is obvious that there are many applications which depend on good non-hybrid solvers, as for example for scheduling or linear optimisation. Furthermore, it is clear that *a good non-hybrid solver should outperform any hybrid solver when applied to some non-hybrid constraint problem.* In that respect, the purpose of this dissertation is not to develop a solver that is better than most existing ones, but to develop a kind of hybrid constraint solver with a greater and - seen from an engineer's perspective - more immediate applicability than most non-hybrid solvers.

2.4.2 Detailed Discussion

The prototypic implementation of RCS has been and is currently applied mainly to problems in which most variables are real-valued. Therefore, this discussion will also focus on existing solvers, frameworks and methods of constraint solving and CLP. Most of the following statements can also be found in [45, Sect. 1.3.] in more detail and with pointers to the respective literature.

CHIP is mentioned to be able to treat linear arithmetic constraints over rational numbers or finite sets of integers, respectively. Those two domain types can however not be mixed. Prolog III is, beside linear arithmetic constraints over rational numbers, also capable of computations over string symbols. Subtle correlations between finite domain variables and real-valued ones, like e.g. in the bulb constraint of Fig. 2.3, are hard to pass through existing CLP language interfaces.

In [38], an example of a hybrid solver is presented, deploying constraint propagation and linear programming in order to tackle combinatorial optimisation. As in the framework of *mixed logical / linear programming (MLLP)* proposed by Hooker *et. al.*, constraint propagation is therein used to determine the feasible region of a search tree, the nodes of which are subject to some optimisation task. Each node comprises a problem instance of linear programming.

A special issue of constraint solving are non-linear arithmetic constraints. Those are mostly not directly supported by CLP solvers: CLP(\mathbb{R}) postpones the treatment of non-linear constraints until they have become linear due to substitutions of grounded variables. This very common technique is presented e.g. in [9].

CAL is reported to make partial use of non-linear constraints which are equations relating polynomials. In RISC-CLP(\mathbb{R}) however, non-linear constraints are fully involved in the computation. [37] points out the necessity to be able to treat non-linear constraints in a complete way. This is proposed based on a simple-minded CLP interpreter, and deploying *partial cylindrical algebraic decomposition (CAD)* and *Gröbner bases*, i.e. *Buchbergers algorithm*. The latter is also used as the core in CAL. Hong's algorithm, as presented in [37], will likewise delay non-linear constraints as long as linear subproblems exist. Gröbner bases are used in a loop to simplify non-linear problems. A final call to CAD produces then the answer to the query, e.g. consistency check.

[36] is another overview paper dealing with the three important implementations of CLP; Prolog III, CAL, and CLP(\mathbb{R}). There, a problem with CAL is emphasised: Although being the only of the three implementations to deal with non-linear constraints (by means of CAD), satisfiability of a constraint problem is reported only with respect to the set \mathbb{C} of all complex numbers. So even when CAL announces consistency, real solutions may still be absent.

Filtering algorithms have been considered to handle systems of non-linear arithmetic constraints. In [53] ideas of and links to filtering algorithms can be found. The method presented in this work attempts to replace the quadratic terms x^2 and $x \cdot y$, by linear narrowing constraints. This procedure, called *reformulation linearisation technique*, produces a linear approximation of the non-linear problem which

can then be solved by known techniques like the simplex algorithm.

Apart from non-linear constraints, expressing disjunctive constraint problems is in general not supported by CLP languages. Alternative subproblems need to be stated and handled individually. This requires the user to provide the relevant sequence of conjunctive problems manually, resulting in an interactive process.

Similar problems arise when addressing context spaces, and connected to that, reuse of previous computations. Sequences of constraint problems cannot be specified in order to automatise the analysis of whole problem spaces; at least not directly in the above systems. Little has been published about multi-context layers on top of CLP solvers. Consequently, reuse issues have only been addressed in the context of special-purpose applications.

Considerable work has been done on mutable parameters, mainly in the context of interval arithmetic; cf. requirement "uncertain knowledge". Like many other publications, [71] defines the interval extension to the arithmetic operators $+$, $-$, \cdot , and \div , and stresses the "dependency problem of interval arithmetics". This is present as soon as an interval has multiple occurrences in an arithmetic term, and is responsible for the incompleteness of naive interval calculus (see Chap. 5). [71] addresses polynomial systems and links thus non-linear constraints and interval arithmetic. There are of course embedded systems that deploy interval arithmetic, and that support uncertain parameters. *Interval constraint satisfaction* was concurrently invented by a few research groups in the 1980's, see for example [40]. In summer 1997, the first commercial version of *Interval Solver* for Microsoft Excel was released. Interval Solver is the first implementation of interval constraint satisfaction on a commercial spreadsheet platform. It is capable of evaluating Excel formulas with interval-valued arguments. By using global interval optimisation algorithms and cascaded function globalisation (see [39]), the actual value ranges are obtained without overestimation typical to classical interval arithmetic. Unlike in ordinary Excel, formula values can be bound with intervals.

The remainder of this subsection takes a closer look at a selection of commercially available constraint solver systems.

- **Optimisation Programming Language (OPL Studio):** This leading-edge modelling tool from ILOG is based in the CPLEX solver, see [41].
- **A Modelling Language for Algebraic Programming (AMPL):** The well-known modelling language is able to interact with numerous solvers. For the investigation undertaken here, a demonstration version was used. The underlying solver was MINOS.
- **General Algebraic Modelling System (GAMS):** Here, the modelling language GAMS was used to deploy once again the CPLEX solver; also in a demonstration version.

- **Mathematica:** Mathematica is a famous computer algebra system from Wolfram Research.
- **UniCalc:** This is a system for mathematical programming developed by the Russian institute for artificial intelligence.

Since modelling hybrid constraint problems in such a way that they can be fed into a non-hybrid solver turns out difficult in the first place, the listed systems were applied to some rather simple constraint problems arising from two families of electric circuits. Those are shown in Fig. 2.7.

Although the presented families of constraint problems mention only real-valued variables, and are thus in the above sense non-hybrid, they still serve to point out shortcomings of the applied solvers.

Both families of electric circuits have $k \in \mathbb{N}, k > 0$, repetitions of a special *box*, in Fig. 2.7 named $B_i, i > 0$. This box is in the case of the family $\{R_k\}_{k>0}$ an aggregate of five Ohmic resistors and two Kirchhoff nodes which is not serial-parallel decomposable. “ R ” stands for “resistors”. For the family $\{D_k\}_{k>0}$, two resistors are replaced by diodes; therefore this time the identifier “ D ” is used.

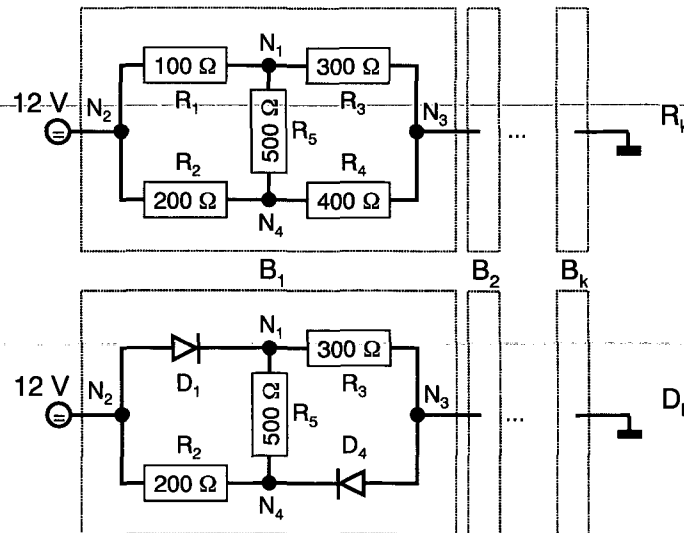


Figure 2.7: Two Families of Electric Circuits

Let us take a closer look at the variables and constraints of the two families of constraint problems. Assigned to each resistor and diode are one current and one voltage variable per port. Similarly, each Kirchhoff node has three ports yielding six variables. Ideal voltage source and ground have one unindexed current and voltage variable each.

Figure 2.8 presents the constraints for $D_k, k > 0$. From those the constraint model for $R_k, k > 0$ is easily obtained, by simply removing the diode block and considering all five resistors instead of just three. Both R_k and D_k involve $44 \cdot k + 4$ real-valued variables and just as many constraints, counting the disjunction inside a diode model

as a single constraint.

The modelling may seem somewhat awkward, in that it involves more variables than actually necessary. But assuming that we instantiate all constraints from prototypes in an electric component library, this is similar to what one should eventually get. Note also that for D_k , there will in practise only be one path with a non-zero current: This is the one that passes, in each box, through the components N_2, D_1, N_1, R_3 and N_3 . However, for any solver, the constraint model of D_k constitutes basically $2 \cdot 2 = 4$ cases of current flow per box, yielding 4^k alternative cases in total that need to be considered initially.

```

for each box  $B_i, 1 \leq i \leq k$  {
  for each resistor  $R_j, j \in \{2, 3, 5\}$  {
     $i_1 R_j B_i + i_2 R_j B_i = 0,$ 
     $u_1 R_j B_i - u_2 R_j B_i = 100 \cdot j \cdot i_1 R_j B_i$  }
  for each diode  $D_j, j \in \{1, 4\}$  {
     $i_1 D_j B_i + i_2 D_j B_i = 0,$ 
     $((u_1 D_j B_i = u_2 D_j B_i \wedge i_1 D_j B_i \geq 0) \vee$ 
     $(u_1 D_j B_i < u_2 D_j B_i \wedge i_1 D_j B_i = 0))$  }
  for each node  $N_j, 1 \leq j \leq 4$  {
     $i_1 N_j B_i + i_2 N_j B_i + i_3 N_j B_i = 0,$ 
     $u_1 N_j B_i = u_2 N_j B_i,$ 
     $u_1 N_j B_i = u_3 N_j B_i$  }
  wires inside  $B_i$  {
     $i_1 N_1 B_i + i_2 R_5 B_i = 0, \quad u_1 N_1 B_i = u_2 R_5 B_i,$ 
     $i_2 N_1 B_i + i_2 R_5 B_i = 0, \quad u_2 N_1 B_i = u_2 R_5 B_i,$ 
     $i_3 N_1 B_i + i_2 R_5 B_i = 0, \quad u_3 N_1 B_i = u_2 R_5 B_i,$ 
     $i_2 N_2 B_i + i_1 R_1 B_i = 0, \quad u_2 N_2 B_i = u_1 R_1 B_i,$ 
     $i_3 N_2 B_i + i_1 R_2 B_i = 0, \quad u_3 N_2 B_i = u_1 R_2 B_i,$ 
     $i_2 N_3 B_i + i_2 R_3 B_i = 0, \quad u_2 N_3 B_i = u_2 R_3 B_i,$ 
     $i_3 N_3 B_i + i_1 R_4 B_i = 0, \quad u_3 N_3 B_i = u_1 R_4 B_i,$ 
     $i_1 N_4 B_i + i_1 R_5 B_i = 0, \quad u_1 N_4 B_i = u_1 R_5 B_i,$ 
     $i_2 N_4 B_i + i_2 R_2 B_i = 0, \quad u_2 N_4 B_i = u_2 R_2 B_i,$ 
     $i_3 N_4 B_i + i_2 R_4 B_i = 0, \quad u_3 N_4 B_i = u_2 R_4 B_i$  } }
  source & ground with wires {
     $u_{SRC} = 12, \quad i_{SRC} + i_1 N_2 B_1 = 0, \quad u_{SRC} = u_1 N_2 B_1,$ 
     $u_{GND} = 0, \quad i_{GND} + i_1 N_3 B_k = 0, \quad u_{GND} = u_1 N_3 B_k$  }
  for each pair of boxes  $(B_{i-1}, B_i), 2 \leq i \leq k$  {
     $i_1 N_3 B_{i-1} + i_1 N_2 B_i = 0, \quad u_1 N_3 B_{i-1} = u_1 N_2 B_i$  } }

```

Figure 2.8: Constraints for the Problem $D_k, k > 0$

UniCalc failed to solve any of the two problems. There is evidence that UniCalc has been designed to solve systems of equations in which each equation is interpreted

as a rule for determining a variable, provided that values are known for all other appearing variables. UniCalc can therefore not be expected to undertake symbolic manipulations, resolve cyclic dependencies or to solve problems by means other than propagation. It will hence be workable only for a small domain of relevant problems.

Secondly, none of the systems was able to solve any of $\{D_k\}_{k>0}$. This is due to the inability to express the disjunctive constraint in the diode model. Obviously, when trying to model alternative physical behaviours of a given component, imposing a disjunctive constraint is the appropriate and rather natural measure.

Although Mathematica computes all variables in the family $\{R_k\}_{k>0}$ for small k , the response times become impracticable for $k > 5$.¹⁰ This is a consequence of the time-consuming symbolic manipulations facilitated by Mathematica. Also, Mathematica assumes that all variables be non-negative, and the constraints be linear equations. Besides the runtimes, this is in general a very strong limitation.

OPL, AMPL, and GAMS are all modelling languages, and one should distinguish between the languages themselves and the power of the underlying solver.

GAMS is the weakest of the three languages, for it just facilitates optimisation and not constraint solving. Its syntax is rather complicated. Moreover, programs are not parametrisable, and the error messages are ambiguous.

OPL has a simple syntax. It can handle constraint solving, and proved to be the fastest among the systems. Unfortunately, OPL will fail when presented non-linear constraints. As an example, suppose the problem R_k would be altered by replacing Ohm's law in the resistor model (see Fig. 2.8) by the two constraints

$$u_1 R_j B_i - u_2 R_j B_i = r R_j B_i \cdot c_1 R_j B_i, \quad r R_j B_i = 100 \cdot j,$$

i.e. introducing a new resistance variable $r R_j B_i$. The new problem R'_k is no longer linear, and OPL will fail to solve it, even though some simple substitutions would turn R'_k into the linear problem R_k .

AMPL is very similar to OPL, and in fact the syntax of OPL was inspired from AMPL. It has the advantage that it can be used with many external solvers. Before solving a system, it applies a *presolve* stage that deploys some simple transformations and makes thereby the initial problem easier. This presolve stage also subsumes the simple substitutions that will turn R'_k into R_k , and hence enable AMPL to also solve the family $\{R'_k\}_{k>0}$.

Further remarks can be made concerning the requirement of processing uncertain parameters. In OPL, AMPL, and GAMS, it is possible to set a user-defined accuracy of all findings. Due to its symbolic reformulations, Mathematica will never reduce the accuracy of its computations, and it is in this respect not scalable.

However, this kind of accuracy does not address the uncertainty or mutability of initial parameters. Both problem families could, for instance, be altered by restricting the resistance of $R_j B_i$ to lie within the interval $[100 \cdot j - \Delta, 100 \cdot j + \Delta]$, where Δ is a small positive number, instead of being exactly $100 \cdot j$. The arising families of constraint problems are much harder to solve, and the above solver systems all fail

¹⁰All investigations have been undertaken on an Intel Celeron 550 MHz, running under Microsoft Windows 98.

for those. In the case of R_k , modelling this kind of uncertain resistance gives R'_k , where there is a extra interval domain constraint for each resistance variable rR_jB_i .

For the above discussion, we have focussed on the electric circuits depicted in Fig. 2.7. We made clear that the modelling in terms of constraints consists of disjunctions and conjunctions of atomic arithmetic constraints; see the diode constraint in Fig. 2.8. Already for that class of constraints, we were able to point out shortcomings of the studied solver systems.

Let us conclude this subsection by taking again a look at the bulb constraint in Fig. 2.3: Clearly, any constraint problem that mentions the presented or similar heterogeneous constraints, will become even harder to solve than the above discussed families of circuits. This is due to the mixture of constraints involving finite domain variables and real-valued ones. Indeed, with this extra difficulty, the application of the above solvers will be possible only after considerable reformulations. However, most engineering tasks simply do not allow for the implied additional effort.

2.4.3 Summary & Outlook on RCS

None of the examined constraint solver systems was able to solve the given families $\{R_k\}_{k>0}$, $\{R'_k\}_{k>0}$, $\{D_k\}_{k>0}$ of constraint problems. As to the latter family, the reason for that was the inability to express and impose disjunctive constraints. Most prominent solvers for constraints over real-valued variables are limited to linear constraints. AMPL deploys a presolve stage to simplify non-linear constraints by substituting some occurring variables with known values. As already mentioned, there are other concepts for dealing with non-linear constraints.

Furthermore, none of the systems is prepared to deal with uncertain parameters, in the sense that imposing interval bounds on some of the variables will basically increase the algebraic degree of the problem and thus make it non-linear and too hard to solve.

Another point is that neither of the presented solvers allows for an almost straightforward input of heterogeneous constraints, that is, so that the modelling effort for engineers remains acceptable.

Although not examined above, most solvers do not support multi-contextuality directly. Still, some systems may reuse previous computations after minor alterations in the sense of *dynamic constraints*. The user can however not state a set of contexts of interest in the first place. In order to investigate a sequence of similar constraint problems he will have to switch from context to context manually, and leave it to the solver to maximise computational reuse in each context switch, if supported at all.

In providing minimal proofs and conflicts respectively, none of the existing systems meets the challenging demands set by engineering applications. Deriving minimal explanations is so far commonly considered a service that is to be provided by additional implementational layers beyond and based upon the actual solver module.

Summarising, the presented requirements catalogue for specification-driven design

and model-based system analysis is only partly covered by existing solver systems.

As will turn out, the relational constraint solver, RCS, is a *concise framework that allows for the satisfaction of all those requirements*. Before we go in for the formal development of that framework, we shall give a brief description of it. This is to give a first idea on how RCS works:

- In order to solve a given constraint problem, RCS will build a so-called *aggregation forest*, that is, a set of binary *aggregation trees*. Their leaf nodes are given by the initial constraints. Any non-leaf node is the result of symbolic manipulations involving the two successor nodes. Those manipulations also take care of possible heterogeneous constraints.
- After having built the trees, RCS can immediately decide the consistency of the initial constraint problem.
- By a subsequent top-down traversal of all aggregation trees, RCS finds all solutions for all variables. This makes sense only for consistent constraint problems; otherwise this traversal is going to be omitted.
- Also in case of consistency, shortest proofs for all solutions can be retrieved, again by directly utilising the generated trees. Note that, with such proofs, the requirement to provide explanations can be satisfied.
- Contrariwise, for inconsistent problem instances, the trees may be used to provide minimal conflicts to the user.
- A diagnosis client may define a whole sequence of contexts to be analysed. RCS will then automatically optimise computational reuse, when switching from one, already analysed context, to the next one. Hereby, reuse can be implemented by identifying reusable subtrees in the aggregation forest that had been generated for the previous context. Consequently, RCS is able to guarantee efficient reuse facilities without the use of TMS's, as mentioned in Subsect. 2.2.2.
- RCS is furthermore able to handle uncertain coefficients in linear constraints and uncertain parameters in non-linear constraints. As an example, solving the above problems R_k and R'_k in RCS is basically done by the same algorithms, and even with very similar traces: Common interval extensions to all basic arithmetic operators have been implemented in our prototype of RCS, making it treat R'_k just as the linear problem R_k . The escrow issue of handling R'_k is instead shifted to the task of producing tightest evaluations of arithmetic terms with mutable parameters. As was argued earlier, this is in general a hard optimisation problem, and is going to be discussed in Chap. 5.
- Finally, Chap. 5 will explain how a concrete implementation of RCS can be designed to support procedural constraints. Basically, those can be seen as a special embodiment of a *term*. Such a term can only be evaluated for a

given vector of input values, by invoking a piece of (unknown) code. Therefore, we cannot manipulate it, and it manifests a *directed dependency* in the computation scheme of RCS.

The two-layer architecture of RCS allows for an almost fixed implementation at top level, the so-called *relational engine*. Improvements in the bottom layer, the *relational processor*, may on the other hand increase the power of the entire solver; cf. Fig. 1.1 for the two-layer architecture.

Those improvements can also lead to the ability to process new classes of constraints and to support new features of the constraint language understood by RCS. Also for those reasons, RCS is a promising alternative for model-based engineering applications, since software developers may initiate desired improvements of RCS according to ongoing work and driven by the needs of new applications, without affecting the existing functionality of the system.

For an example of such a gradual, application-driven extension of the supported constraint language, see App. C.

Chapter 3

A Formal Framework for Relational Aggregation

This chapter introduces the concept of relational aggregation. Starting with basic entities and operations, the more complex structures aggregation tree, aggregation forest, and context space will be defined and related to each other by some lemmas and the main theorems of this dissertation. These facilitate the core algorithms for an implementation of RCS, as presented in subsequent chapters.

3.1 Conventions & Notation

Throughout this document, the following conventions and notational agreements shall be observed.

\mathbb{N}	$\stackrel{def}{=} \{0, 1, 2, 3, \dots\}$	denotes the set of all natural numbers, including 0.
\mathbb{N}_+	$\stackrel{def}{=} \mathbb{N} \setminus \{0\}$	is the set of all positive natural numbers.
$f _{\mathcal{D}}$		stands for the restriction of a function $f : \mathcal{M} \rightarrow \mathcal{N}$ to the set $\mathcal{D} \subseteq \mathcal{M}$.
$\odot_{i \in I} T(i)$		will be used, for any associative operator \odot and any non-empty ordered set of indices $I = \{i_1, i_2, \dots\}$ to abbreviate the term $T(i_1) \odot T(i_2) \odot \dots$. If $ I = 1$, the term shall be evaluated to $T(i_1)$.
$\odot \mathcal{M}$	$\stackrel{def}{=} \odot_{m \in \mathcal{M}} m$	is a shorthand for the case that the operator is also commutative, and index set and set of operands coincide. ¹

Sets will be denoted by capital letters, sets of assignments (see Def. 1) in calligraphic style, e.g. $\mathcal{A}, \mathcal{B}, \mathcal{C}$; lower-case letters stand for the elements of sets. Usually X, Y, Z and x, y, z will be used for sets of variables and individual variables, respectively.

¹To be precise, $\odot_{i \in I} T(i)$ has only been defined for ordered index sets. But due to commutativity, an arbitrary ordering can be chosen, when iterating over $m \in \mathcal{M}$.

In order to define the central objects of this dissertation, *aggregation trees*, some commonly used terms for *trees* need to be fixed. Normally, a tree $\Delta = (V, E)$ is a special graph, i.e. a pair of finitely many *vertices* - also called *nodes* - and *edges* $E \subseteq V \times V$. V may be retrieved by writing $V(\Delta)$. Let furthermore in this dissertation any edge in a tree be directed. The edge from $n_1 \in V(\Delta)$ to $n_2 \in V(\Delta)$ may be denoted by $n_1 \rightarrow n_2$; n_1 being a *predecessor* of n_2 and n_2 a *successor* of n_1 . A *path* from node n to node $m \neq n$ exists if and only if there is a non-empty sequence of nodes $n_0 = n, n_1, \dots, n_k = m$, $k \in \mathbb{N}_+$, such that $n_i \rightarrow n_{i+1}$ is an edge for each $i \in \{0, 1, \dots, k-1\}$. This path may be written as $n \xrightarrow{*} m$.

The characteristic of a tree is that there exists a single node without predecessor called *root*, $\rho(\Delta)$, and that all other nodes have exactly one predecessor. The nodes without successors, are called *leaf nodes* or *leaves*. Any node that is not a leaf is named *non-leaf node*. Let, for any node n in a given tree, $\Delta(n)$ denote the subtree rooted at n , which may consist just of the single node n , in case n is a leaf. Also, let $\Lambda(n)$ be the set of all leaf nodes of $\Delta(n)$. A tree is termed *binary* if any non-leaf node has exactly two successors.²

In order to support the notation of more involved statements concerning trees, let us define *leaf replacements* in the following way. Let $N \subseteq V(\Delta)$ be a non-empty set of nodes such that

$$\forall n_1, n_2 \in N \quad n_1 \neq n_2 \implies \Lambda(n_1) \cap \Lambda(n_2) = \emptyset,$$

that is, no two subtrees rooted at distinct elements of N share a leaf. Then any element of

$$\tau(N) \stackrel{\text{def}}{=} \left\{ \bigcup_{n \in N} \Lambda_n \mid \Lambda_n \subseteq \Lambda(n) \wedge \Lambda_n \neq \emptyset \right\} \quad (3.1)$$

is called a *leaf replacement* of N . By definition, any leaf replacement replaces each non-leaf node n of N by a non-empty set of leaves (namely Λ_n) of the subtree rooted at n . This implies that $|N'| \geq |N|$ for all $N' \in \tau(N)$.

3.2 Preliminaries

3.2.1 Basic Entities

In order to define constraints and all operators for constraints that are crucial for relational aggregation, some basic technical terms shall be made clear and fixed. The definitions include also some phrases that are common in the constraint research community.

Definition 1 (Variable, Domain, Value, Assignment)

A *variable* x is a place holder object. Each variable x has a *domain*, retrieved by $\text{dom}(x)$. Any variable's domain is a non-empty, possibly infinite set of objects called *values*.

²I use Δ because of its resemblance to a depicted tree with its root at the top. Λ , i.e. $\underline{\text{L}}\text{ambda}$, denotes the leaves.

x is said to take or attain its value in $\text{dom}(x)$. An **assignment** to a non-empty finite set X of variables is a function α , that assigns to each variable $x \in X$ a value in $\text{dom}(x)$,

$$\begin{aligned} \alpha: X &\longrightarrow \bigcup_{x \in X} \text{dom}(x) \\ x &\longmapsto v \in \text{dom}(x). \end{aligned}$$

Thinking of assignments as of functions has a rather mathematical flavour. But it has advantages when defining prominent relational operators, as will become clear in what follows.

In the framework of constraint satisfaction, assignments as defined above are called *labels*, and assigning v_1, v_2, \dots, v_n to x_1, x_2, \dots, x_n , $n \in \mathbb{N}_+$, respectively, is there denoted as the label $\langle x_1, v_1 \rangle \langle x_2, v_2 \rangle \dots \langle x_n, v_n \rangle$, see e.g. [70].

Classically, a constraint is defined to be a pair of two sets; the variables related by the constraint, and the set of value tuples allowed by the constraint. The below definition is a variant of that, facilitating subsequent definitions for relational operators. However, due to practical issues that arise when implementing RCS, there is a need for two trivial constraints that do not mention any variable; the empty and the full constraint, see Def. 3. Any other constraint is therefore non-trivial:

Definition 2 (Non-trivial Constraint)

A **non-trivial constraint** c is a pair (X, \mathcal{A}) , where $X = \{x_1, x_2, \dots, x_n\}$, $n \in \mathbb{N}_+$, denotes a non-empty, finite set of variables, and \mathcal{A} is a set of assignments to X . The variables of c can be retrieved by $\text{vars}(c)$, i.e. $\text{vars}(c) = X$.

c is said to be **satisfiable** if and only if $\mathcal{A} \neq \emptyset$. Otherwise c is **unsatisfiable**.

A variable x_i , $i \in \{1, 2, \dots, n\}$, is called a **free variable** of c , if $\mathcal{A} \neq \emptyset$, and

$$\forall \alpha \in \mathcal{A} \forall v_i \in \text{dom}(x_i) \exists \alpha^{(i)} \in \mathcal{A}: \alpha^{(i)}|_{X \setminus \{x_i\}} \equiv \alpha|_{X \setminus \{x_i\}} \wedge \alpha^{(i)}(x_i) = v_i.$$

Let $\text{free}(c)$ denote the set of all free variables of c . Any variable in $\text{vars}(c) \setminus \text{free}(c)$ is termed **restricted** by c . c is **nonrestrictive** if $\text{vars}(c) = \text{free}(c)$, otherwise c is called **restrictive**.

By definition, no unsatisfiable constraint has a free variable; thus any unsatisfiable constraint is also restrictive.

In the case of a free variable x of some non-trivial constraint c , we may in any tuple³ belonging to c , replace the value assigned to x by any element of $\text{dom}(x)$, and will still obtain a tuple that belongs to c . Consequently, if c is nonrestrictive, we may replace any value by any other of the corresponding domain. This yields the following lemma.

Lemma 1

A non-trivial constraint $c = (\{x_1, x_2, \dots, x_n\}, \mathcal{A})$, $n \in \mathbb{N}_+$ is nonrestrictive if and only if

³The term *tuple* may sometimes be used as a synonym for *assignment*, whenever the constraint's set of variables has been enumerated, and the mapping is thus according to index positions.

$$\begin{aligned} &\forall v_1 \in \text{dom}(x_1) \ \forall v_2 \in \text{dom}(x_2) \ \dots \ \forall v_n \in \text{dom}(x_n) \\ &\exists \alpha \in \mathcal{A} \ \forall i \in \{1, 2, \dots, n\} \ \alpha(x_i) = v_i. \end{aligned}$$

The proof of this equivalence, as well as all other proofs omitted in the text, can be found in the appendix. The lemma states that any non-trivial, nonrestrictive constraint contains all possible assignments, i.e. that the set of assignments relates to the cartesian product of the variables' domains.

Example 1: Suppose the variables x, y, z take their values in \mathbb{R} . Then writing $x + y = z \wedge z = 1$ is a shorthand for the non-trivial constraint

$$c = (\{x, y, z\}, \{\alpha : \{x, y, z\} \longrightarrow \mathbb{R}^3 \mid \alpha(x) + \alpha(y) = \alpha(z), \alpha(z) = 1\}).$$

The set of all tuples of c form the intersecting line of two non-parallel planes in three-dimensional space; thus c is satisfiable. Clearly, for any point on the line, leaving two coordinates fixed and altering just one, will not yield a point on that line. Therefore, none of x, y, z is a free variable of c .

$$d = (\{x, y, z\}, \{\alpha : \{x, y, z\} \longrightarrow \mathbb{R}^3 \mid \alpha(x) + \alpha(y) = 1\})$$

looks similar, but the encoded condition $x + y = 1$ does not mention z . Indeed, z is a free variable of d which forms this time a plane, perpendicular to the $x - y$ -plane. Again, neither x nor y is a free variable. Any tuple of c is a tuple of d , since the line encoded by c is contained in the plane captured by d .

Shifting from c to d eliminates z and abandons the condition $z = 1$. It adds a degree of freedom in the z -direction. The next subsection deals, among other issues, with that loss of information implied by projection, i.e. variable elimination.

Definition 3 (Trivial Constraints \circledast and \square , Constraints)

\circledast and \square are defined to be **trivial constraints**. Those have no variable, i.e. $\text{vars}(\circledast) = \text{vars}(\square) = \emptyset$.

Extending definitions for constraint attributes, as declared in Def. 2, \circledast is unsatisfiable and restrictive. Any given variable is restricted by \circledast . Contrariwise, \square is satisfiable and nonrestrictive, and no variable is said to be restricted by \square .

All non-trivial constraints and the two trivial ones form the set of all **constraints**.

Summarising by combining Defs. 2 and 3, one now gets for any constraint c , that

- c is either satisfiable or unsatisfiable;
- c is either restrictive or nonrestrictive;
- any given variable is either restricted by c , or a free variable of c ; and
- the set of variables of c can be retrieved by $\text{vars}(c)$.

Moreover, $\text{vars}(c) = \emptyset$ if and only if c is trivial. Note that the term *free variable* as well as the accessor $\text{free}(\cdot)$ have been defined only for non-trivial constraints.

Since constraints are basically sets of tuples, or more precisely assignments, and the following operators are essential in the theory of *relational databases*, constraints will in this work also be called **relations**. Likewise, for the set of variables $\text{vars}(c)$ of a constraint c , the term **scope** may be used, see e.g. [31].

3.2.2 Operations on Relations

For a more original view on the *join* of two given relations, and *projections* of an arbitrary relation, see e.g. [55] or [2]. Both operators produce new constraints. The join basically collects all assignments that belong to both operands. Projection of a non-trivial constraint (X, \mathcal{A}) onto a subset Y of X , removes from each tuple the entries that correspond to the variables in $X \setminus Y$.

Definition 4 (Join)

Let $c = (X, \mathcal{A})$ and $d = (Y, \mathcal{B})$ be two non-trivial constraints. The **join** of c and d is defined as

$$c \bowtie d \stackrel{\text{def}}{=} \left(X \cup Y, \left\{ \alpha : X \cup Y \rightarrow \bigcup_{z \in X \cup Y} \text{dom}(z) : \alpha|_X \in \mathcal{A} \wedge \alpha|_Y \in \mathcal{B} \right\} \right).$$

Set furthermore for any constraint c ,

$$\begin{aligned} c \bowtie \emptyset &= \emptyset \bowtie c \stackrel{\text{def}}{=} \emptyset \\ c \bowtie \square &= \square \bowtie c \stackrel{\text{def}}{=} c. \end{aligned}$$

Note that both last defining lines yield identical results for $\emptyset \bowtie \square = \square \bowtie \emptyset = \emptyset$.

The join operator is well-studied and known to be both *commutative* and *associative*:

Lemma 2

\bowtie is commutative and associative, i.e. for arbitrary constraints a, b , and c , the following conditions hold

$$\begin{aligned} a \bowtie b &= b \bowtie a, \text{ and} \\ a \bowtie (b \bowtie c) &= (a \bowtie b) \bowtie c. \end{aligned}$$

Definition 5 (Projection)

Let c be any constraint, and $Y \subseteq \text{vars}(c)$ be any subset of variables of c .

In the case that $c = (X, \mathcal{A})$ is non-trivial, the **projection** of c onto Y is defined according to

$$\pi_Y(c) \stackrel{\text{def}}{=} \begin{cases} \square, & Y = \emptyset \wedge c \text{ satisfiable} \\ \emptyset, & Y = \emptyset \wedge c \text{ unsatisfiable} \\ (Y, \{ \beta : Y \rightarrow \bigcup_{y \in Y} \text{dom}(y) \mid \\ \exists \alpha \in \mathcal{A} \alpha|_Y \equiv \beta \}), & Y \neq \emptyset. \end{cases}$$

For a trivial constraint c , $\text{vars}(c) = \emptyset$, and so $Y = \emptyset$ must hold. Define

$$\begin{aligned} \pi_{\emptyset}(\emptyset) &\stackrel{\text{def}}{=} \emptyset, \text{ and} \\ \pi_{\emptyset}(\square) &\stackrel{\text{def}}{=} \square. \end{aligned}$$

An immediate and simple consequence of Def. 5 is

$$\pi_{vars(c)}(c) \equiv c, \quad (3.2)$$

for any constraint c .

Example 1 has already shown that sometimes, due to free variables, a constraint may actually be simplified; regard the constraint d . Any efficient implementation of RCS should prefer such sparse representations, i.e. omitting the variable z in d . Thereby, any instantiated constraint is going to represent a whole class of equivalent constraints in the following sense.

Definition 6 (Equivalence of Constraints)

Let c and d be two non-trivial constraints. Those are defined to be equivalent if

$$c \sim d \stackrel{\text{def}}{\iff} \pi_{vars(c) \setminus free(c)}(c) \equiv \pi_{vars(d) \setminus free(d)}(d). \quad (3.3)$$

Set furthermore, for an arbitrary constraint e ,

$$\begin{aligned} e \sim \emptyset &\iff \emptyset \sim e \stackrel{\text{def}}{\iff} e \text{ unsatisfiable, and} \\ e \sim \square &\iff \square \sim e \stackrel{\text{def}}{\iff} e \text{ nonrestrictive.} \end{aligned}$$

In the above definition, \equiv has been used to relate two constraints. This will mean, throughout the entire work, that either both constraints are the same trivial constraint, or that they have identical scopes *and* identical sets of assignments.

The appendix shows that \sim is well-defined, and that it is indeed an equivalence relation.

For each implementation of constraint solving and constraint programming, the most important tasks are to determine *consistency*, and - in case of consistency - to find all *solutions*.

The next definition fixes both terms:

Definition 7 (Constraint Problem, Solution, (In-)Consistency)

A non-empty set of non-trivial constraints C is called **constraint problem**. Writing $\bowtie C$ as $\bowtie C = (X, \mathcal{A})$, we define

$$\Sigma(C) \stackrel{\text{def}}{=} \mathcal{A} \quad (3.4)$$

to be the set of all **solutions** to C . C is termed **consistent** if and only if $\Sigma(C) \neq \emptyset$, otherwise **inconsistent**.

Lemma 2 guarantees that $\bowtie C$ is well-defined, since the computation does not depend on the order of all applied binary joins. An immediate consequence of Defs. 2, 6, and 7 is

$$\bowtie C \sim \emptyset \iff \Sigma(C) = \emptyset, \quad (3.5)$$

because $\bowtie C$ is unsatisfiable if and only if $\mathcal{A} = \emptyset$. Hence, a means to prove inconsistency of any given constraint problem C is to show that its combined join $\bowtie C$ may

be simplified to the empty constraint \emptyset . Finding efficient schemes for computing $\bowtie C$ has been - and still is in practical applications - a main issue in the theory of relational databases (see [55], [2]) and of the work at hand.

The next lemma provides another intuition to a solution of a constraint problem C : An assignment α is a solution if and only if, for each $c \in C$ there is an appropriate restriction of α belonging to c :

Lemma 3

Let C be a constraint problem, and let α be an assignment to $\bigcup_{c \in C} \text{vars}(c)$. Then

$$\alpha \in \Sigma(C) \iff \forall (X, A) \in C \quad \alpha|_X \in A. \quad (3.6)$$

Example 2: Figure 3.1 depicts the famous graph colouring problem. A map with the four countries Germany, Poland, Austria and the Czech Republic is to be coloured using the three colours *red*, *blue* and *green* such that each country is assigned exactly one colour and no two neighbouring countries are assigned the same colour. In Fig. 3.1, neighbouring countries are given as connected nodes. Each edge stands for an instance of the constraint

$$c(x, y) \stackrel{\text{def}}{=} (\{x, y\}, \{\alpha : \{x, y\} \longrightarrow \{\text{red}, \text{blue}, \text{green}\}^2 : \alpha(x) \neq \alpha(y)\}),$$

with $x, y \in \{G, P, A, C\}$. Typically, the intensional condition $x \neq y$ can be equivalently represented by an extensional enumeration of all allowed pairs of values. The map is colourable in the given sense, if and only if the constraint problem

$$C \stackrel{\text{def}}{=} \{c(G, P), c(G, C), c(G, A), c(A, C), c(P, C)\}$$

has a solution in the sense of Def. 7. According to (3.5), this is the case, since based on the variable ordering (G, P, A, C) (*blue*, *red*, *red*, *green*) is a tuple in $\bowtie C$. The problem could for example be solved using the *bucket elimination scheme*, as

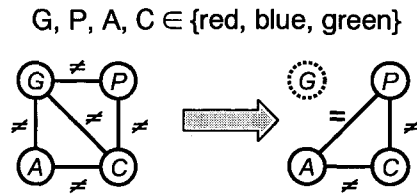


Figure 3.1: A Graph Colouring Problem with Four Countries

presented in [18]. Based on the given variable ordering, this scheme would attempt to eliminate the variable G , and while doing that deduce that A and P must be assigned the same colour, cf. the right-hand side of Fig. 3.1. This is due to the fact that G, P, C as well as G, A, C must be assigned mutually distinct colours, and that there are only three available colours. Bucket elimination guarantees that each

solution to the reduced problem (excluding Germany) can be extended to a solution of the initial problem. In other words,

$$\pi_{\{P,A,C\}}(\bowtie C) = c(P,C) \bowtie c(A,C) \bowtie \hat{c}(A,P),$$

where $\hat{c}(x,y)$ captures equality, i.e. all those assignments to $\{x,y\}$ that do not belong to $c(x,y)$.

Inserting an additional edge in the original graph between A and P turns the graph into a clique. Then the colouring problem is inconsistent for three colours. However, using four colours will, in this case, obviously re-establish consistency since there are only four countries to be coloured.

For large problems, as for example arising from engineering applications, deciding the predicate in (3.5) turns out not practicable, since subsequent joins will make the resulting relations large in terms of scopes. However, one may expect for most hierarchical system models, that their mathematical descriptions - in terms of constraints, see Subsect. 2.1.3 - are going to satisfy a *low density assumption*. That is, each variable mentioned in an arising constraint problem C appears in only few $c \in C$.

This dissertation aims thus at replacing the binary operator \bowtie in (3.5) by some other, more convenient one. This *aggregation* operator, as defined below, will apply after each join some appropriate projection, to reduce scopes again. In order to facilitate aggregation, some useful properties relating join and projection are stated:

Lemma 4

Let c, c_1 , and c_2 be arbitrary constraints, and $X_i = \text{vars}(c_i)$, for $i \in \{1, 2\}$. Then

$$\pi_X(c) \sim \emptyset \iff c \sim \emptyset, \text{ for all } X \subseteq \text{vars}(c), \quad (3.7)$$

$$X_1 \cap X_2 \subseteq X \subseteq X_2 \implies$$

$$(c_1 \bowtie c_2 \sim \emptyset \iff c_1 \bowtie \pi_X(c_2) \sim \emptyset), \quad (3.8)$$

$$X_1 \cap X_2 \subseteq Y_1 \subseteq Y_2 \subseteq X_2 \implies$$

$$\pi_{Y_2}(c_1 \bowtie c_2) \equiv \pi_{Y_1}(c_1 \bowtie c_2) \bowtie \pi_{Y_2}(c_2). \quad (3.9)$$

The left-hand side term of (3.9) becomes, for $Y_2 = X_2$, the portion of c_2 that joins with c_1 , and is commonly referred to as the *semi-join*, $c_2 \bowtie c_1$, see e.g. [55, p. 355].

Definition 8 (Aggregation)

For any two constraints c, d and any set of variables $X \subseteq \text{vars}(c) \cup \text{vars}(d)$, the *aggregation* of c and d with respect to (w.r.t.) X is defined as

$$\text{agg}_X(c, d) \stackrel{\text{def}}{=} \pi_X(c \bowtie d).$$

Example 3: A simple example of an aggregation is presented in Fig. 3.2, which also makes clear that mathematical aggregation, as defined by Def. 8, is the pendant to aggregation in the sense of engineering: Suppose, we have a mathematical

system description of some electric circuit involving the depicted series of two Ohmic resistors. The upper row contains the initial constraints, that is, two instantiations of Ohm's law. A trivial, local simplification can be made by replacing both resistors by a single one. This one is going to have the forward slope resistance $R + S$.

Mathematical aggregation does the same thing on the level of constraints: Joining $v_1 - v = R \cdot i_1 \wedge i_1 = i$ with $v - v_2 = S \cdot i \wedge i = i_2$, and projecting the result onto $\{v_1, v_2, i_1, i_2\}$ produces the constraint shown in the lower row.⁴ No other constraint

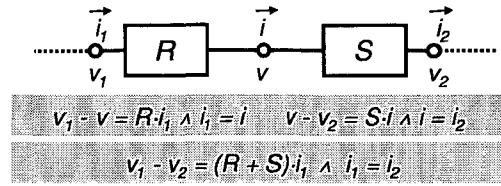


Figure 3.2: Aggregation of Two Ohmic Resistors

will be affected by this local change. More generally, aggregations typically intend to eliminate all those variables that are no longer part of the interface of some component or already aggregated subsystem. In the example, those variables are v and i .

The previous example motivates the following definition.

3.3 Solving Single Constraint Problems

3.3.1 Aggregation Trees

Definition 9 (Interface Variables)

Let C be a set of arbitrary constraints, and d be any constraint not belonging to C , $d \notin C$. The set of variables

$$\text{int}(d, C) \stackrel{\text{def}}{=} \left(\bigcup_{c \in C} \text{vars}(c) \right) \cap \text{vars}(d)$$

is called the interface of d with respect to (w.r.t.) C .

For $C = \emptyset$, the above union is defined to be \emptyset ; and thus $\text{int}(d, \emptyset) \stackrel{\text{def}}{=} \emptyset$.

Obviously, $\text{int}(d, C) \subseteq \text{vars}(d)$, and for $|C| \geq 1$, $\text{int}(d, C) = \emptyset$ holds if and only if no variable of d is mentioned in any constraint of C .

We are now prepared to define *aggregation trees*:

⁴In this example, R and S are assumed to be numerical parameters. The join will just union the two equation sets; elimination of the intermediate variable v is accomplished by adding both Ohmic laws. i can be replaced by either of i_1 and i_2 .

Definition 10 (Connectedness, Aggregation Tree & Forest)

Two non-empty sets M_1, M_2 of arbitrary constraints are **disconnected** if they do not share a variable, i.e. $\text{vars}(c_1) \cap \text{vars}(c_2) = \emptyset$, for any two constraints $c_1 \in M_1$, $c_2 \in M_2$. A non-empty set of constraints M is called **connected** if it cannot be split into a pair of disconnected sets, i.e. there exist no two disconnected subsets $\emptyset \neq M_1 \subset M$ and $\emptyset \neq M_2 \subset M$ such that $M_1 \cap M_2 = \emptyset$ and $M_1 \cup M_2 = M$.

Let now C be an arbitrary constraint problem.

An **aggregation tree** for any connected portion $D \subseteq C$ is a binary tree, the nodes of which are constraints, such that the following conditions hold:

1. The set of all leaf nodes equals D .
2. Any non-leaf node d is the predecessor of two constraints d_1, d_2 with the property $d = \text{agg}_X(d_1, d_2)$, where $X = \text{int}(d_1, C \setminus \Lambda(d)) \cup \text{int}(d_2, C \setminus \Lambda(d))$.

An **aggregation forest** for C is a non-empty set of aggregation trees, the sets of leaves of which union to C .

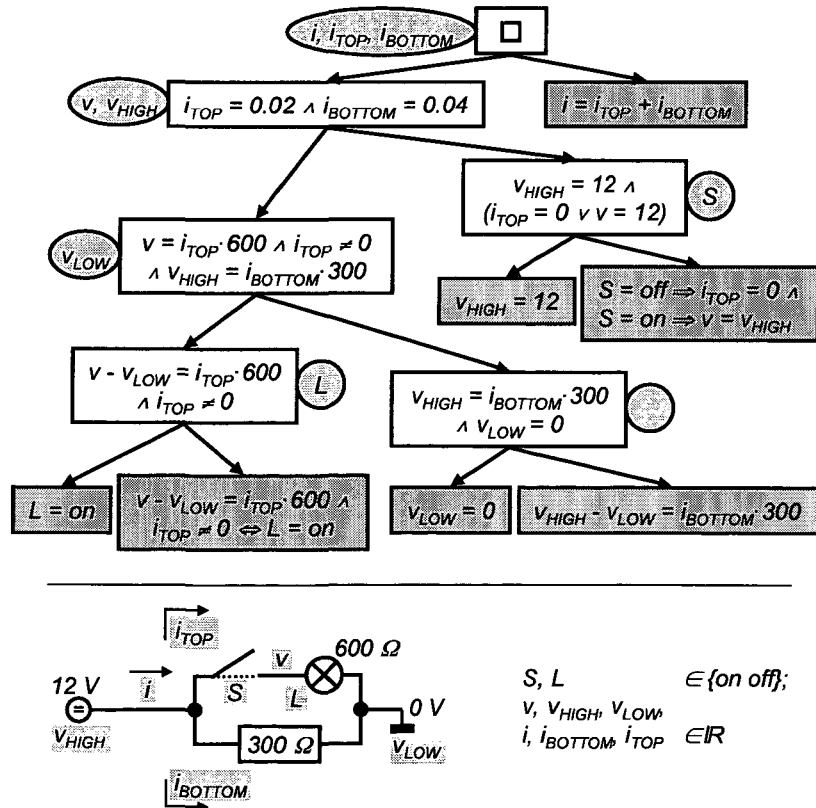


Figure 3.3: An Aggregation Tree for an Electric Circuit

By Def. 10, any non-leaf node d computes a certain projection of the join of its successors d_1 and d_2 . That projection is onto the interface of $d_1 \bowtie d_2$ w.r.t. the set

of constraints in C that do not appear as a leaf in $\Delta(d)$.⁵

Note that any given constraint problem C induces a trivial aggregation forest for C : Just take each $c \in C$ to be a trivial aggregation tree with the single node c . The above definition is clearly satisfied since any singleton set of constraints is connected, and condition 2. for an aggregation tree, is only triggered when non-leaf nodes exist.

Example 4: The bottom part of Fig. 3.3 depicts a small electric circuit. The seven leaves of the shown tree, coloured dark grey, are the constraints that correspond to a mathematical model of the circuit together with the observation that the bulb is lit, $L = on$. For the sake of simplicity, connectors are omitted, and the model is thus a condensed version of what one would obtain when following the modelling procedure given in Subsect. 2.1.3. Also, constraints are written in an obvious intensional shorthand, abbreviating the more elaborate pair notation, as introduced by Def. 2. The presented tree is an aggregation tree in the sense of Def. 10. The bubbles attached to each non-leaf constraint contain the variables that have been eliminated during the respective aggregation. For example, L may be eliminated since it does not lie in the interface to the remaining portion of leaves of the join of $L = on$ with the neighbouring bulb constraint. Note that the root is the nonrestrictive trivial constraint \square . The below Th. 1 proves that this guarantees consistency of the initial constraint problem.

Before stating the first theorem, some properties of aggregation trees shall be listed.

Lemma 5

Let $C, |C| \geq 2$, be a connected constraint problem with at least two constraints; and let Δ be an aggregation tree for C . Then

1. For each variable x mentioned in C , i.e. $x \in \bigcup_{c \in C} \text{vars}(c)$, there exists a unique non-leaf node $\epsilon(x)$ in Δ which eliminates x : $x \in (\text{vars}(r_1) \cup \text{vars}(r_2)) \setminus \text{vars}(r)$, where r_1, r_2 are the successors of $r = \epsilon(x)$.
2. $\forall c \in C \ (x \in \text{vars}(c) \implies c \in \Lambda(\epsilon(x)))$.
3. Δ satisfies a **connectedness condition**⁶: For each variable x mentioned in C , the set of nodes r of Δ with $x \in \text{vars}(r)$ form a subtree rooted at $\epsilon(x)$, with the exception that that root does not mention x .
4. If r_1, r_2 denote again the successors of any node r in Δ , then $\left(\bigcup_{c \in \Lambda(r_1)} \text{vars}(c)\right) \cap \left(\bigcup_{c \in \Lambda(r_2)} \text{vars}(c)\right) \subseteq \text{vars}(r_1) \cap \text{vars}(r_2)$.
5. For any node r in Δ ,

$$(a) \ r \equiv \pi_{\text{vars}(r)}(\bowtie \Lambda(r)),$$

⁵The above definition avoids the introduction of *constraint hypergraphs*, see for instance [31] or [55, p. 448]. Basically, that is a pair (V, E) of vertices and hyperedges such that there is a vertex for each variable in $\bigcup_{c \in C} \text{vars}(c)$, and a hyperedge $e = \text{vars}(c) \in E$ for each $c \in C$. Connected subsets of C , as defined in Def. 10, are then just path-connected sets of hyperedges in the constraint hypergraph associated with C .

⁶Cf. the "connectedness condition" for *join trees*, as e.g. defined in [31].

- (b) If r is a non-leaf node, with successors r_1, r_2 , then
 $r_1 \bowtie r_2 \equiv \pi_{vars(r_1) \cup vars(r_2)}(\bowtie \Lambda(r))$.
6. For any node r in Δ with successors r_1, r_2 , and any set $M \subseteq C \setminus \Lambda(r)$,
 $\bowtie (M \cup \{r_1, r_2\}) \sim \emptyset \iff \bowtie (M \cup \{r\}) \sim \emptyset$.
7. Let M be a set of nodes of Δ such that $\tau(M)$ is well-defined (see (3.1)). Then
 $\forall M' \in \tau(M) \quad (\bowtie M' \sim \emptyset \implies \bowtie M \sim \emptyset)$.

Example 4 may be used again, to verify some of the statements made in the above Lem. 5: There is exactly one bubble, i.e. one eliminating node, for each of the eight variables in the constraint problem, which establishes statement 1. Regarding, for instance, all nodes that mention v_{LOW} , one easily checks the connectedness condition.

3.3.2 Deciding Consistency

As indicated by the above Ex. 4, arriving at \square , as the root of some aggregation tree, proves consistency of the constraint problem defined by the set of all leaves. Contrariwise, the root relation \emptyset signals inconsistency:

Theorem 1 (Deciding Consistency)

Let $C, |C| \geq 2$, be any connected constraint problem with at least two constraints; and let r be the root of an aggregation tree for C . Then

$$r \in \{\emptyset, \square\}; \quad (3.10)$$

$$r \equiv \emptyset \iff C \text{ inconsistent; and} \quad (3.11)$$

$$r \equiv \square \iff C \text{ consistent.} \quad (3.12)$$

The theorem does not elucidate how to actually derive an aggregation tree for any given constraint problem. That is going to be the subject of Sect. 3.4. In this section, the existence of an aggregation tree or forest will tacitly be assumed.

Proof:

(3.10)

Consider the final aggregation that produces the root r of the aggregation tree. Condition 2. of Def. 10 and Def. 8 say that r is the projection onto some set of variables X . However, $C = \Lambda(r)$ by condition 1. of Def. 10, and so X involves two interfaces w.r.t. an empty set of constraints. Thus $X = \emptyset$, by Def. 9. According to Def. 5, r is then restricted to be a trivial constraint.

(3.11)

Two proofs can be given for (3.11); a short one that directly applies one of the properties of aggregation trees as listed in Lem. 5, and an inductive one that emphasises the fact that an aggregation tree is actually a problem simplification scheme.

The first proof applies statement 5.(a) of Lem. 5 to the root node r of the given

aggregation tree. Due to $\text{vars}(r) = \emptyset$ and $\Lambda(r) = C$, this provides the condition $r \equiv \pi_{\emptyset}(\bowtie C)$. Therefore

$$\begin{aligned}
 & r \equiv \emptyset \\
 \iff & r \sim \emptyset, && \text{by (3.10),} \\
 \iff & \pi_{\emptyset}(\bowtie C) \sim \emptyset \\
 \iff & \bowtie C \sim \emptyset, && \text{by (3.7),} \\
 \iff & \Sigma(C) = \emptyset, && \text{according to (3.5),} \\
 \iff & C \text{ inconsistent,} && \text{by Def. 7.}
 \end{aligned}$$

For the inductive proof, pick any two leaves d_1, d_2 of the given aggregation tree, that are successors of the same non-leaf constraint d . Then, by Lem. 5, part 6.,

$$C \text{ inconsistent} \iff (C \setminus \{d_1, d_2\}) \cup \{d\} \text{ inconsistent.}$$

The argument shows that the initial problem can be reduced to deciding consistency for a constraint problem that has one less constraint. Visually, this means to chop off the leaves d_1 and d_2 of the aggregation tree, and make thus d a new leaf node. The new tree has one less leaf. Repeated application of that argument will finally reduce the tree to its root node r which is known to be either \emptyset or \square . Therefore, C is inconsistent if and only if $r \equiv \emptyset$.

(3.12)

Taking (3.10) into account, this is just the negation of (3.11).

q.e.d.

3.3.3 Finding all Solutions

This subsection will assume the existence of an aggregation tree for some connected constraint problem C with at least two constraints. According to Th. 1, consistency may have already been affirmed.

In this setup, a common query is to derive the tightest restriction for a given variable x mentioned in C . In other words, this captures the requirement to return all values in $\text{dom}(x)$ that are assigned by at least one of the solutions $\Sigma(C)$. Considering again the circuit description given in Ex. 4, and supposing that the observation $L = on$ is missing, deriving the tightest restriction for i_{TOP} should yield just the two values 0 and $12 \div 600 = 0.02$, according to the unknown switch position. Similarly, i is going to be either $12 \div 300 = 0.04$ or $12 \div ((600 * 300) / (600 + 300)) = 12 \div 200 = 0.06$.

Apart from the plain information as to what the possible values for a variable x are, there are other motivations for serving the outlined requirement. For example in the context of specification-driven design, narrowing the domain of x will help the user to focus on the relevant system setups. He may then choose a certain subset of feasible values for x , according to some preferences. That subset can be fed back as an additional unary constraint in x . Thereby, constraint processing will guide the designer through the space of all system assemblies.

The next definition captures the notion of *tightest restriction* for some variable:

Definition 11 (Tightest Restriction)

Let $C, |C| \geq 2$, be a constraint problem with at least two constraints. The constraint

$$t_x \stackrel{\text{def}}{=} (\{x\}, \{\alpha|_{\{x\}} : \alpha \in \Sigma(C)\}) = \pi_{\{x\}}(\bowtie C)$$

is called the **tightest restriction** for $x \in \bigcup_{c \in C} \text{vars}(c)$, imposed by C .

Obviously, any assignment in t_x assigns a value just to x , and can be extended to a solution of C . Moreover,

$$\Sigma(C) \subseteq \Sigma \left(\left\{ t_x : x \in \bigcup_{c \in C} \text{vars}(c) \right\} \right). \quad (3.13)$$

So, joining all tightest restrictions will give, in general, more solutions than the original constraint problem has. The intuitive reason for that is that the projections accomplished when deriving tightest restrictions, “destroy” correlations between the variables in C . Computing the right-hand side set of assignments in (3.13) involves joining all tightest restrictions. During this combined join, no partial assignment is going to be excluded since no two tightest restrictions share a variable.

Example 5: Returning to the graph colouring example in Ex. 2 and Fig. 3.1, one observes that $t_x = \{x \mapsto \text{red}, x \mapsto \text{blue}, x \mapsto \text{green}\}^7$, for all $x \in \{G, P, A, C\}$. The reason for that is that there exists a solution in the first place, and that, for any solution, the colours can be permuted in order to produce a new solution. In this example, the right-hand side set of assignments in (3.13) becomes obviously the set of all possible assignments to $\{G, P, A, C\}$, which is a proper superset of $\Sigma(C)$.

To present an example where the set inclusion (3.13) is also proper but the larger set does not contain all possible assignments, regard again the graph colouring example with the additional constraint $P = \text{red}$.

Example 2 has already elaborated $P = A$. And so, assuming the variable ordering (G, P, A, C) , the set of all solutions can be abbreviated by listing the tuples $\{(\text{blue}, \text{red}, \text{red}, \text{green}), (\text{green}, \text{red}, \text{red}, \text{blue})\}$. The left- and right-hand side sets of assignments in (3.13) contain thus 2 and $2 \cdot 1 \cdot 1 \cdot 2 = 4$ assignments, respectively.

After having given some motivation as well as illustrating examples, the question is now how to actually compute tightest restrictions.

Clearly, a solution by means of aggregation trees shall be favoured. The next definition assumes an existing aggregation tree, and introduces an additional constraint, the *backward relation*, for each non-leaf node, which will help to compute tightest restrictions:

Definition 12 (Forward & Backward Relation)

Let C be as in Def. 11, and Δ be an aggregation tree for C . Any non-leaf node of Δ will also be called **forward relation**.

For the root $\rho = \rho(\Delta)$ with successors r_1, r_2 , $\text{bw}(\rho) \stackrel{\text{def}}{=} r_1 \bowtie r_2$ denotes its **backward**

⁷The string “ $x \mapsto v$ ” is a lax way of denoting the assignment that maps x to v .

relation.

For any other non-leaf node r with predecessor R and successors r_1, r_2 , its **backward relation** is defined by

$$bw(r) \stackrel{\text{def}}{=} \pi_{vars(r)}(bw(R)) \bowtie r_1 \bowtie r_2.$$

The reader should note that Def. 12 is a recursive definition. The defining scheme is top-down, starting at the root $\rho(\Delta)$ and descending to the lowest non-leaf nodes. The next chapter presents a bottom-up algorithm for building an aggregation tree Δ , given the leaves, i.e. a constraint problem C . The second phase of assigning backward relations to existing non-leaf nodes is then to take place clearly only afterwards. Therefore, building an aggregation tree will also be referred to as the *forward phase*, whereas the second phase is named *backward phase*. This gives a clue to the names *forward* and *backward relation*.

Theorem 2 (Deriving Tightest Restrictions)

Let C be a consistent constraint problem with $|C| \geq 2$, and let Δ be an aggregation tree for C . Then the following condition holds for any non-leaf node r in Δ with the successors r_1, r_2 .

$$bw(r) \equiv \pi_{vars(r_1) \cup vars(r_2)}(\bowtie C). \quad (3.14)$$

Furthermore, for each $x \in \bigcup_{c \in C} vars(c)$,

$$t_x \equiv \pi_{\{x\}}(bw(\epsilon(x))), \quad (3.15)$$

where $\epsilon(x)$ is the unique node as defined by statement 1. of Lem. 5.

The theorem provides a plan for deriving the tightest restriction for x : After top-down-assigning all backward relations from the root $\rho(\Delta)$ to $\epsilon(x)$, the projection onto $\{x\}$ of the latter will provide t_x .

Proof:

(3.14) (by top-down induction on Δ)

Assume first that $r = \rho(\Delta)$ with successors r_1, r_2 . Then the left-hand side of the assertion becomes $r_1 \bowtie r_2$, by Def. 12. According to statement 5.(b) of Lem. 5, and because of $\Lambda(r) = C$, one obtains the stated equation.

Let now r be any other non-leaf node with predecessor R and successors r_1, r_2 , and let s denote the second successor of R . By induction on Δ , it may be assumed that (3.14) is valid for the node R . Then

$$\begin{aligned} bw(r) &\equiv \pi_{vars(r)}(bw(R)) \bowtie r_1 \bowtie r_2, && \text{by Def. 12,} \\ &\equiv \pi_{vars(r)}(\pi_{vars(r) \cup vars(s)}(\bowtie C)) \bowtie r_1 \bowtie r_2, && \text{due to induction,} \\ &\equiv \pi_{vars(r)}((\bowtie (C \setminus \Lambda(r))) \bowtie (\bowtie \Lambda(r))) \bowtie r_1 \bowtie r_2 \\ &\equiv \pi_{vars(r)}((\bowtie (C \setminus \Lambda(r))) \bowtie (\bowtie \Lambda(r))) \\ &\quad \bowtie \pi_{vars(r_1) \cup vars(r_2)}(\bowtie \Lambda(r)), && \text{by Lem. 5, 5.(b),} \\ &\equiv \pi_{vars(r_1) \cup vars(r_2)}(\bowtie C). \end{aligned}$$

The last equality results from the application of (3.9) with the bindings $c_1 = \bowtie(C \setminus \Lambda(r))$, $c_2 = \bowtie\Lambda(r)$, $Y_1 = \text{vars}(r)$, and $Y_2 = \text{vars}(r_1) \cup \text{vars}(r_2)$. It is easy to verify all pre-conditions of (3.9), apart from $\text{vars}(c_1) \cap \text{vars}(c_2) \subseteq Y_1$. To this end, fix a variable $x \in \text{vars}(c_1) \cap \text{vars}(c_2)$. Then x appears in some leaf of the subtree rooted at r , and in some leaf outside that subtree. But then the connectedness condition, Lem. 5, 3., implies that x must also appear in r , i.e. $x \in \text{vars}(r) = Y_1$.

(3.15)

Choosing $r = \epsilon(x)$ in (3.14), and projecting both sides onto $\{x\}$ immediately yields

$$\pi_{\{x\}}(bw(\epsilon(x))) \equiv \pi_{\{x\}}(\bowtie C) \equiv t_x,$$

by Def. 11.

q.e.d

3.3.4 Providing Minimal Conflicts

Chapter 2 emphasises the requirement of providing minimal subsets of conflicting relations, given an inconsistent constraint problem. This subsection addresses that requirement and presents two theorems which make clear how to derive all minimal conflicts, and one such minimal conflict, respectively. Finding just one minimal conflict is often sufficient for the user, and therefore explicitly addressed. As will turn out later, this problem can be solved more efficiently than deriving all minimal conflicts.

[59] also addresses the task of finding just one minimal conflict, based on aggregation trees. However, it lacks the formal proof of the correctness of the presented algorithm. Neither is the problem of deriving all minimal conflicts addressed.

Definition 13 ((Minimal) Conflict)

Suppose C is an inconsistent constraint problem. Then any inconsistent subset $D \subseteq C$, $D \neq \emptyset$ is called a **conflict** of C .

Moreover, D is named **minimal conflict** of C , if any $E \subset D$ with $E \neq D$, $E \neq \emptyset$ is consistent.

Given an aggregation tree Δ for an inconsistent constraint problem C , the next definition assigns to each node two sets of sets of nodes of Δ . The first, $con_{\downarrow}(\cdot)$, is defined following a top-down scheme, and the second, $con_{\uparrow}(\cdot)$, in a bottom-up manner starting at the leaves. According to the below Th. 3, the set assigned last, $con_{\uparrow}(\rho(\Delta))$, is basically going to be the set of all minimal conflicts of C .

Definition 14 ($con_{\downarrow}(r)$, $con_{\uparrow}(r)$)

Suppose C is an inconsistent connected constraint problem with $|C| \geq 2$, and Δ is an aggregation tree for C . Let $r \in V(\Delta)$ be any node with the successors r_1, r_2 and define

$$\begin{aligned} con_{\downarrow}(\rho(\Delta)) &\stackrel{\text{def}}{=} \{\emptyset\}, \\ con_{\downarrow}(r_i) &\stackrel{\text{def}}{=} \{M \in con_{\downarrow}(r) \mid \bowtie(M \cup \{r_i\}) \sim \emptyset\} \\ &\quad \cup \{M \cup \{r_{3-i}\} \mid M \in con_{\downarrow}(r)\}, \quad i \in \{1, 2\}, \\ con_{\uparrow}(\lambda) &\stackrel{\text{def}}{=} \{M \cup \{\lambda\} \mid M \in con_{\downarrow}(\lambda)\}, \quad \lambda \in \Lambda(\rho(\Delta)), \end{aligned}$$

$$\begin{aligned}
con_{\uparrow}(r) &\stackrel{def}{=} \{M \in con_{\uparrow}(r_1) \mid r_2 \notin M\} \cup \\
&\quad \{M \in con_{\uparrow}(r_2) \mid r_1 \notin M\} \cup \\
&\quad \{M \stackrel{def}{=} (M_1 \setminus \{r_2\}) \cup (M_2 \setminus \{r_1\}) \mid \begin{array}{l} M_1 \in con_{\uparrow}(r_1) \wedge \\ M_2 \in con_{\uparrow}(r_2) \wedge \\ r_1 \in M_2 \wedge \\ r_2 \in M_1 \wedge \\ \bowtie M \sim \emptyset \end{array} \}.
\end{aligned}$$

The next lemma provides some properties of the sets of sets of nodes defined by Def. 14. Those are required to prove the subsequent theorem that relates $con_{\uparrow}(\rho(\Delta))$ to the desired set of all minimal conflicts. Figure 3.4 explains the statements 3. and 4.

Lemma 6

Let C and Δ be as in Def. 14, and write $\rho = \rho(\Delta)$. Then

1. $\forall r \in V(\Delta) \forall M \in con_{\downarrow}(r) \bowtie (M \cup \{r\}) \sim \emptyset$;
2. $\forall r \in V(\Delta) \forall M \in con_{\uparrow}(r) \bowtie M \sim \emptyset$;
3. $\forall r \in V(\Delta) \forall M \in con_{\downarrow}(r) \forall n \in M$ *n is not on $\rho \xrightarrow{*} r$, but n is the successor of some node m on $\rho \xrightarrow{*} r$ with $m \neq r$;*
4. $\forall r \in V(\Delta) \forall M \in con_{\uparrow}(r) \quad M \cap \Lambda(r) \neq \emptyset \wedge$
 $\forall n \in M \quad n \in V(\Delta(r)) \implies n \in \Lambda(r) \wedge$
 $n \notin V(\Delta(r)) \implies$ *n is not on $\rho \xrightarrow{*} r$, but n is the successor of some node m on $\rho \xrightarrow{*} r$ with $m \neq r$;*
5. $\forall M \in con_{\uparrow}(\rho) \quad M \subseteq \Lambda(\rho)$;
6. *If K is a minimal conflict of C , then*
 $\forall r \in V(\Delta) \quad (K \cap \Lambda(r) \neq \emptyset \implies \exists M \in con_{\downarrow}(r) \quad K \in \tau(M \cup \{r\}))^8$;
7. *If K is a minimal conflict of C , then*
 $\forall r \in V(\Delta) \quad (K \cap \Lambda(r) \neq \emptyset \implies \exists M \in con_{\uparrow}(r) \quad K \in \tau(M))$.

Now, by Lem. 6, 5., any element $M \in con_{\uparrow}(\rho)$ contains exclusively leaves, thus $\tau(M) = \{M\}$. And the application of Lem. 6, 7., to ρ yields that $con_{\uparrow}(\rho)$ contains all minimal conflicts of C ; see the proof in the appendix. Moreover, we obtain:

⁸Recall once again the definition in (3.1).

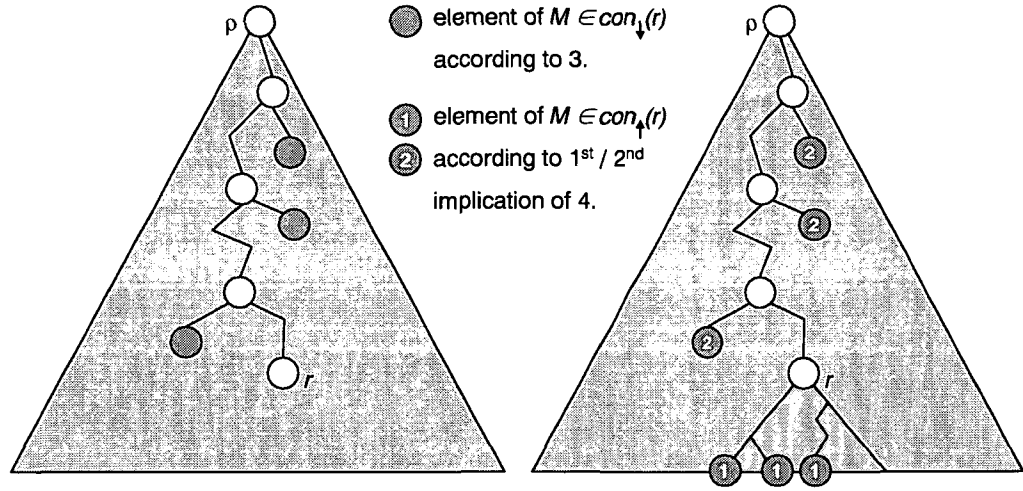


Figure 3.4: Explanation of Statements 3. and 4. of Lem. 6

Theorem 3 (Providing all Minimal Conflicts)

With the above notations, the set of all minimal conflicts of C equals the set

$$\{M \in \text{con}_\uparrow(\rho) \mid \forall N \in \text{con}_\uparrow(\rho) \ N \subseteq M \implies N = M\}.$$

In other words, $\text{con}_\uparrow(\rho)$ contains all minimal conflicts of C and perhaps supersets of those.

Proof:

Suppose K is a minimal conflict of C . Application of Lem. 6, statement 7, to $\rho = \rho(\Delta)$ implies the existence of an $M \in \text{con}_\uparrow(\rho)$ such that $K \in \tau(M)$. Statement 5 of the same lemma ensures that M is a set of leaves, and hence $\tau(M) = \{M\}$. But this means that $K = M$. Therefore, K is an element of $\text{con}_\uparrow(\rho)$.

Lemma 6, 2., ensures furthermore that any element of $\text{con}_\uparrow(\rho)$ is a conflict of C , and is thus the superset of some minimal conflict of C .

Consequently, $\text{con}_\uparrow(\rho)$ contains all minimal conflicts of C and maybe supersets of those, but no other sets of nodes. Since the containment condition of the set provided by Th. 3 excludes all proper supersets, that set must coincide with the set of all minimal conflicts of C . q.e.d

As already mentioned above, very often it suffices to provide just one minimal conflict. This task is addressed by the below Th. 4. Again, we first need to define two auxiliary sets; a top-down set, $\overline{\text{con}}_\downarrow(\cdot)$, and a bottom-up set, $\overline{\text{con}}_\uparrow(\cdot)$, for which the subsequent lemma states some properties. This time, those sets are not going to be sets of sets of nodes, but simply sets of nodes, since we are just interested in deriving one minimal conflict and not all.

It is also noteworthy that this time, the definition of the assigned node sets is not symmetric as in Def. 14. That is due to the fact that, for each non-leaf node, the computations in its two subtrees must be carried out in the same order for both $\widehat{con}_\downarrow(r)$ and $\widehat{con}_\uparrow(r)$.

Definition 15 ($\widehat{con}_\downarrow(r)$, $\widehat{con}_\uparrow(r)$)

Suppose C, Δ, r, r_1 and r_2 are as in Def. 14. Let furthermore \star be a special symbol signaling the undefined status. Suppose that the successors of r are ordered, and that r_1 denote the first successor and r_2 the second. Then define

$$\begin{aligned}
 \widehat{con}_\downarrow(\rho) &\stackrel{\text{def}}{=} \emptyset; \\
 \widehat{con}_\downarrow(r_1) &\stackrel{\text{def}}{=} \begin{cases} \star, & \widehat{con}_\downarrow(r) = \star, \\ \widehat{con}_\downarrow(r), & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_1\}) \sim \emptyset, \\ \star, & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_1\}) \not\sim \emptyset \\ & \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_2\}) \sim \emptyset, \\ \widehat{con}_\downarrow(r) \cup \{r_2\}, & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_1\}) \not\sim \emptyset \\ & \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_2\}) \not\sim \emptyset; \end{cases} \\
 \widehat{con}_\downarrow(r_2) &\stackrel{\text{def}}{=} \begin{cases} \star, & \widehat{con}_\downarrow(r) = \star, \\ \widehat{con}_\downarrow(r), & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_2\}) \sim \emptyset, \\ \star, & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_1\}) \sim \emptyset \\ & \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_2\}) \not\sim \emptyset, \\ \widehat{con}_\uparrow(r_1) \setminus \{r_2\}, & \widehat{con}_\downarrow(r) \neq \star \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_1\}) \not\sim \emptyset \\ & \wedge \Delta(\widehat{con}_\downarrow(r) \cup \{r_2\}) \not\sim \emptyset; \end{cases} \\
 \widehat{con}_\uparrow(\lambda) &\stackrel{\text{def}}{=} \begin{cases} \star, & \widehat{con}_\uparrow(\lambda) = \star \wedge \lambda \in \Lambda(\rho) \\ \widehat{con}_\downarrow(\lambda) \cup \{\lambda\}, & \widehat{con}_\uparrow(\lambda) \neq \star \wedge \lambda \in \Lambda(\rho); \end{cases} \\
 \widehat{con}_\uparrow(r) &\stackrel{\text{def}}{=} \begin{cases} \star, & \widehat{con}_\uparrow(r_1) = \star \wedge \widehat{con}_\uparrow(r_2) = \star, \\ \widehat{con}_\uparrow(r_2), & \widehat{con}_\uparrow(r_1) = \star \wedge \widehat{con}_\uparrow(r_2) \neq \star, \\ \widehat{con}_\uparrow(r_1), & \widehat{con}_\uparrow(r_1) \neq \star \wedge \widehat{con}_\uparrow(r_2) = \star, \\ \widehat{con}_\uparrow(r_2), & \widehat{con}_\uparrow(r_1) \neq \star \wedge \widehat{con}_\uparrow(r_2) \neq \star. \end{cases}
 \end{aligned}$$

Note that the definition of $\widehat{con}_\downarrow(r_2)$ involves $\widehat{con}_\uparrow(r_1)$ under a certain condition $P(r, r_1, r_2)$, and so it is ad hoc not clear whether the given definition scheme is coherent. But a simple inductive argument (over the tree structure) shows that $\widehat{con}_\uparrow(\cdot)$ is well-defined for any first successor r_1 . Also, whenever $P(r, r_1, r_2)$ holds, $\widehat{con}_\uparrow(r_1)$ will not equal \star and thus $\widehat{con}_\uparrow(r_1) \setminus \{r_2\}$ is well-defined, too. Thereby, the coherence of the entire definition can easily be verified.

Lemma 7

With C, Δ and ρ again as in Def. 14, define the logic predicates

$$\begin{aligned}
 E_\downarrow(r) &\stackrel{\text{def}}{=} (\widehat{con}_\downarrow(r) \neq \star \wedge (\widehat{con}_\downarrow(r) \cup \{r\} \text{ is a minimal conflict of } V(\Delta))), \\
 E_\uparrow(r) &\stackrel{\text{def}}{=} (\widehat{con}_\uparrow(r) \neq \star \wedge (\widehat{con}_\uparrow(r) \text{ is a minimal conflict of } V(\Delta))),
 \end{aligned}$$

for any $r \in V(\Delta)$.⁹ Whenever r, r_1, r_2 are not quantified, let r denote any non-leaf node of Δ , and r_1, r_2 its first and second successor, respectively. Then the following conditions hold.

1. $\forall r \in V(\Delta) \quad \widehat{con}_\downarrow(r) = \star \iff \forall s \in V(\Delta(r)) \quad \widehat{con}_\downarrow(s) = \star \iff$
 $\forall s \in \Lambda(r) \quad \widehat{con}_\downarrow(s) = \star \iff \forall s \in \Lambda(r) \quad \widehat{con}_\uparrow(s) = \star \iff$
 $\forall s \in V(\Delta(r)) \quad \widehat{con}_\uparrow(s) = \star \iff \widehat{con}_\uparrow(r) = \star;$
2. $E_\downarrow(\rho);$
3. For all $s \in V(\Delta)$ with $\widehat{con}_\downarrow(s) \neq \star$,
 - (a) $\widehat{con}_\downarrow(s) \setminus V(\Delta(s)) = \widehat{con}_\uparrow(s) \setminus V(\Delta(s)),$
 - (b) $\widehat{con}_\downarrow(s) \cap V(\Delta(s)) = \emptyset,$
 - (c) $\widehat{con}_\uparrow(s) \cap V(\Delta(s)) \subseteq \Lambda(s);$
4. $\widehat{con}_\downarrow(r_1) \neq \star \implies (E_\downarrow(r) \Rightarrow E_\downarrow(r_1));$
5. $(\widehat{con}_\downarrow(r_2) \neq \star \wedge \widehat{con}_\downarrow(r_1) = \star) \implies (E_\downarrow(r) \Rightarrow E_\downarrow(r_2));$
6. $\forall \lambda \in \Lambda(\rho) \quad (E_\downarrow(\lambda) \implies E_\uparrow(\lambda));$
7. $(\widehat{con}_\uparrow(r_1) \neq \star \wedge \widehat{con}_\uparrow(r_2) = \star) \implies (E_\uparrow(r_1) \Rightarrow E_\uparrow(r));$
8. $\widehat{con}_\uparrow(r_2) \neq \star \implies (E_\uparrow(r_2) \Rightarrow E_\uparrow(r));$
9. $(\widehat{con}_\downarrow(r_1) \neq \star \wedge \widehat{con}_\downarrow(r_2) \neq \star) \implies (E_\uparrow(r_1) \Rightarrow E_\downarrow(r_2));$
10. $E_\uparrow(\rho).$

Theorem 4 (Providing one Minimal Conflict)

Let $C, |C| \geq 2$, be an inconsistent connected constraint problem, and Δ be an aggregation tree for C . Then the set of nodes $\widehat{con}_\uparrow(\rho(\Delta))$ is a minimal conflict of C .

Proof:

Statement 10 of Lemma 7 guarantees that $\widehat{con}_\uparrow(\rho)$ is a minimal conflict of $V(\Delta)$. Moreover, due to statement 3(c) of the same lemma, we know that $\widehat{con}_\uparrow(\rho)$ consists exclusively of leaves of Δ , i.e. of constraints in C . Therefore, the assertion follows. q.e.d

Definition 15 entails a plan for computing $\widehat{con}_\uparrow(\rho)$, and the next chapter is going to present the corresponding algorithm.

Before we shall turn to minimal explanations which make up the remainder of this chapter, let us have a closer look at an example. It presents an inconsistent connected constraint problem that has exactly two minimal conflicts. All four sets

⁹Clearly, if the first conjunct turns out *false*, the second shall not be investigated.

$con_{\downarrow}(\cdot)$, $con_{\uparrow}(\cdot)$, $\widehat{con}_{\downarrow}(\cdot)$ and $\widehat{con}_{\uparrow}(\cdot)$ will be presented for each node of a corresponding aggregation tree.

Example 6: A prominent constraint satisfaction problem is the so-called *n-queens problem*. The task is to place $n, n \in \mathbb{N}_+$, queens on an $n \times n$ chessboard such that no two queens attack each other. The chess rules state that a queen attacks any other piece if it is either on the same row, column, or diagonal to that piece. One easily observes that that problem has no solution for $n = 3$. Figure 3.5 depicts an aggregation tree for the 3-queens problem. A popular way of modelling the *n-queens*

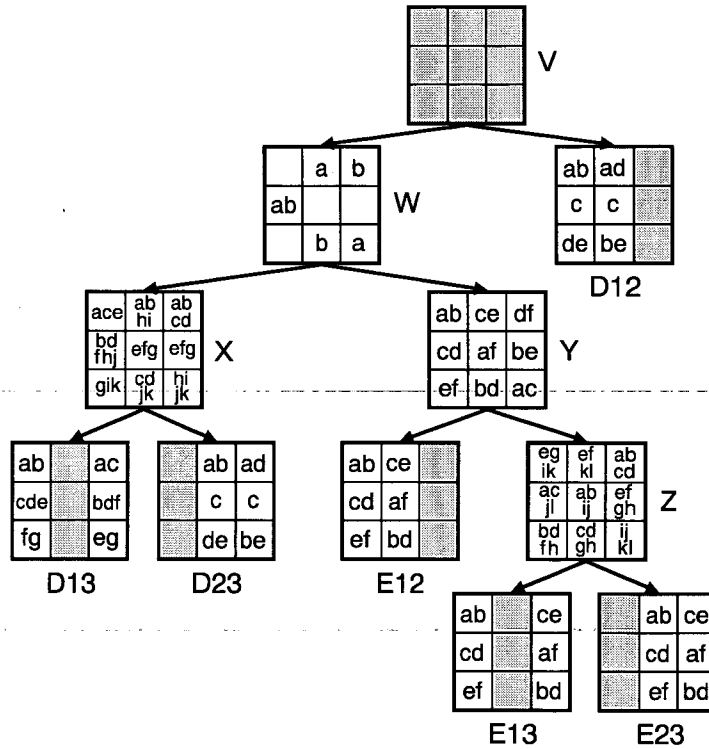


Figure 3.5: Aggregation Tree for the 3-Queens Problem

problem is to introduce n row variables $x_i, i \in \{1, 2, \dots, n\}$, where the i^{th} queen is to be placed on row x_i , column i . That guarantees that no two queens are on the same column. In Fig. 3.5, the leaf node Eij captures the constraint that the queens on the columns i and j must not be on the same row; whereas Dij says that they are not placed diagonally.

Columns are coloured grey if the respective constraint does not constrain the queen on that column. The lower case letters present all possible solution tuples in the corresponding node. E.g. 3 queens may be placed according to the positions of the letter *d* in node *Z*, i.e. at row 1, column 3, and row 3, column 1, and row 3, column 2. This placement satisfies both leaf constraints *E13* and *E23*, but not *E12*. The table presented in Fig. 3.7 lists all four conflict sets, as defined above, for each

node in the aggregation tree. By Th. 3, the entry for $con_{\uparrow}(V)$ contains all minimal conflicts, and maybe supersets of those. Obviously, the first two sets in the given set of sets must be those minimal conflicts. The remaining three sets are proper supersets. We see thus that there are exactly two minimal conflicts for the 3-queens problem.

Definition 15 requires the successors of each non-leaf node to be ordered: Let in our tree in Fig. 3.5, the left successor always come first. Then, the single minimal conflict found by $\widehat{con}_{\uparrow}(V)$ is $\{D23, E12, E13, E23, D12\}$. Note that the sets for finding one minimal conflict are computed in a depth-first manner, i.e. with respect to the scheme given in Fig. 3.6

It is furthermore easy to verify that, if we swapped the ordering of the successors of node W from (X, Y) to (Y, X) , then $\widehat{con}_{\uparrow}(V)$ would result in the other minimal conflict $\{E13, D12, D13, D23\}$.

3.3.5 Providing Minimal Explanations

We have already seen an example of a consistent constraint problem, where, for a certain variable, any solution to the entire problem must assign the same value to that variable: The tree in Figure 3.3 proves consistency, by Th. 1, and i_{BOTTOM} has to be assigned the value 0.04, as becomes clear by taking a look at the left successor of the root. Obviously, that value can already be derived from the three leaves $v_{HIGH} = 12$, $v_{LOW} = 0$, and $v_{HIGH} - v_{LOW} = i_{BOTTOM} \cdot 300$. In other words, the assignment $i_{BOTTOM} = 0.04$ is independent of the observation concerning the light L . Neither does the switch position S play a role.

This is a common pattern in engineering, where subsets of the entire set of constraints often already fix certain variables. However, the user might get lost in too many details, and not recognise those causal dependencies. *Explanations* as defined below, can help resolve that situation by reporting smallest subsets of constraints that already suffice to deduce the tightest restriction for some variable, as imposed by the entire set of constraints.

Definition 16 ((Minimal) Explanation)

Let C be a consistent constraint problem, and x be a variable that appears in at least one $c \in C$. Assume furthermore that there exists at least one exceptional value $v \in \text{dom}(x)$ such that $(x \mapsto v)$ is not an assignment of the tightest restriction t_x for x imposed by C .

Then any non-empty set $E \subseteq C$ with $\pi_{\{x\}}(\bowtie E) \equiv t_x$ is called an **explanation** for x . Moreover, E is called a **minimal explanation** if, for any non-empty proper subset $D \subset E$, $\pi_{\{x\}}(\bowtie D) \not\equiv t_x$.¹⁰

Note that the existence of some exceptional value $v \in \text{dom}(x)$ guarantees that the domain of x is restricted by C at all. For, if any assignment $(v \mapsto x), v \in \text{dom}(x)$, belongs to t_x , then x may, according to C , still assume any value of its domain. But that essentially means that x is not constrained, and hence there is nothing to

¹⁰Explanations as defined here are very often called *minimal supporting sets*. In those contexts, the term *explanation* is usually reserved for tree-like structures, the leaves of which form the minimal supporting set.

explain.

The main result of this subsection is that, given a consistent constraint problem C , minimal explanations for a variable x can basically be derived by computing minimal conflicts for a slightly altered inconsistent constraint problem.

To make this work, we first need an appropriate inconsistent constraint problem C_x . Secondly, in order to run the machinery for computing minimal conflicts, developed in the previous subsection, we must have an aggregation tree Δ_x for C_x . Preferably, Δ_x is an alteration of a given aggregation tree Δ for C .

So, let C denote a consistent connected constraint problem with $|C| \geq 2$. We fix a variable $x \in \bigcup_{c \in C} \text{vars}(c)$ for which C excludes at least one value $v_0 \in \text{dom}(x)$, and write its tightest restriction $t_x = \pi_{\{x\}}(\boxtimes C)$ imposed by C , as $t_x = (\{x\}, \mathcal{A})$. Then we can define the constraint

$$s_x \stackrel{\text{def}}{=} (\{x\}, \{\alpha : \{x\} \longrightarrow \text{dom}(x) \mid \alpha \notin \mathcal{A}\}). \quad (3.16)$$

By assumption, $(x \mapsto v_0)$ is not an assignment of t_x but hence of s_x , and so $s_x \not\sim \emptyset$. Likewise, $s_x \not\sim \square$, since $t_x \not\sim \emptyset$ because C is assumed to be consistent. However, by construction, $t_x \boxtimes s_x \sim \emptyset$, thus consequently

$$\boxtimes (C \cup \{s_x\}) \sim \emptyset. \quad (3.17)$$

Indeed, we now have the following theorem that relates minimal explanations $E \subseteq C$ for x , to minimal conflicts of $C \cup \{s_x\}$:

Theorem 5 (Providing Minimal Explanations)

With C, x and s_x as above, we have

$$\begin{aligned} & E \text{ minimal explanation of } C \text{ for } x \\ \iff & E \cup \{s_x\} \text{ minimal conflict of } C \cup \{s_x\}. \end{aligned}$$

Proof:

Let, for the entire proof, $C_x \stackrel{\text{def}}{=} C \cup \{s_x\}$ and $E_x \stackrel{\text{def}}{=} E \cup \{s_x\}$.

\implies

By Def. 16 and assumption, we know that $\pi_{\{x\}}(\boxtimes E) = t_x$. Since $t_x \boxtimes s_x \sim \emptyset$, this implies

$$\begin{aligned} & \pi_{\{x\}}(\boxtimes E) \boxtimes s_x \sim \emptyset \\ \iff & \boxtimes E \boxtimes s_x \sim \emptyset, & \text{by (3.8) with } X = \{x\}, \\ \iff & \boxtimes E_x \sim \emptyset, \end{aligned}$$

i.e. $E_x \subseteq C_x$ is a conflict.

For the proof of minimality, choose any $c \in E_x$ and try to suspend it. Obviously, $s_x \in E_x$ cannot be suspended, since $E_x \setminus \{s_x\} = E \subseteq C$ is consistent.

So, assume in the following that $c \in E$. Then we have the following equivalences

$$\begin{aligned} & \boxtimes ((E \setminus \{c\}) \cup \{s_x\}) \sim \emptyset, & (*), \text{ marked for later reference,} \\ \iff & \boxtimes (E \setminus \{c\}) \boxtimes s_x \sim \emptyset \\ \iff & \pi_{\{x\}}(\boxtimes (E \setminus \{c\})) \boxtimes s_x \sim \emptyset, & \text{by (3.8) with } X = \{x\}. \end{aligned}$$

Let \mathcal{A} , \mathcal{B} and \mathcal{C} denote the sets of assignments of $\pi_{\{x\}}(\bowtie(E \setminus \{c\}))$, s_x and t_x , respectively. We are trying to show that the first condition (*) in the above chain is false; so let us assume it were true, in order to produce a contradiction.

Thus, the above argument shows that $\mathcal{A} \cap \mathcal{B} = \emptyset$. But then by definition of s_x , $\mathcal{A} \subseteq \mathcal{C}$.

Because of $E \setminus \{c\} \subseteq C$, we know that any assignment of $\bowtie C$ is an assignment of $\bowtie(E \setminus \{c\})$, and so we also derive the converse inclusion $\mathcal{A} \supseteq \mathcal{C}$, i.e. $\mathcal{A} = \mathcal{C}$.

Summarising, this proves that the set of assignments of $\pi_{\{x\}}(\bowtie(E \setminus \{c\}))$ coincides with that of t_x , and therefore, E is not a minimal explanation. That clearly contradicts the assumption of the \Rightarrow -part of the proof, and so (*) must be false, implying the minimality of E_x .

\Leftarrow

The equivalence

$$\pi_{\{x\}}(\bowtie E) \bowtie s_x \sim \emptyset \iff \bowtie E_x \sim \emptyset$$

was already shown in the first part of the proof. Since E_x is now assumed to be a minimal conflict, both conditions must be true. We obtain thus that no assignment of $\pi_{\{x\}}(\bowtie E)$ is an assignment of s_x . In other words, the entire set of assignments of $\pi_{\{x\}}(\bowtie E)$ are subsumed in those of t_x . Let us call those \mathcal{A} and \mathcal{B} , respectively, then this writes as $\mathcal{A} \subseteq \mathcal{B}$.

Again, since $E \subseteq C$, any assignment of $\bowtie C$ is one of $\bowtie E$, too. Therefore, $\mathcal{B} \subseteq \mathcal{A}$; i.e. $\mathcal{A} = \mathcal{B}$, which proves that E is an explanation for x .

It remains to show the minimality of E . Consider again any $c \in E$, and the second chain of equivalences from the first part of the proof:

$$\bowtie((E \setminus \{c\}) \cup \{s_x\}) \sim \emptyset \iff \pi_{\{x\}}(\bowtie(E \setminus \{c\})) \bowtie s_x \sim \emptyset.$$

The first statement is false due to minimality of E_x . Therefore, the right-hand side statement is also false. There must hence exist an assignment of $\pi_{\{x\}}(\bowtie(E \setminus \{c\}))$ that is also an assignment of s_x . Now, the sets of assignments of s_x and t_x are disjoint, and so $E \setminus \{c\}$ can no longer be an explanation for x . This proves the minimality of E , and we are done. q.e.d

Obviously, we would like to apply the above developed conflict-finding procedures, in order to derive one or all minimal explanations. The remaining question is thus, how can we alter an aggregation tree Δ for C , to serve as an aggregation tree Δ_x for $C_x = C \cup \{s_x\}$?

We cannot just introduce a new root that has as successors the old root node and s_x . This is because that new tree would violate the connectedness condition introduced in Lem. 5, 3. But the following procedure generates a new tree Δ_x that is indeed a valid aggregation tree for C_x ; see Fig. 3.8

Let $n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k$ denote the sequence of edges of Δ that make up to the path $\rho(\Delta) \xrightarrow{*} \epsilon(x)$, i.e. $n_1 = \rho$ and $n_k = \epsilon(x)$. Note that $k = 1$ is a possible scenario; the one in which $\epsilon(x) = \rho$.

Alteration of Δ is now done by first altering n_k , then n_{k-1} , n_{k-2} , and so forth up to

n_1 . Assuming that each $n_i, i \in \{k, k-1, \dots, 1\}$, has the (possibly already altered) successors n_i^1 and n_i^2 , those alterations are according to the replacement

$$n_i' \stackrel{\text{def}}{=} \pi_X(n_i^1 \bowtie n_i^2), \quad \text{where } X = \text{vars}(n_i) \cup \{x\}.$$

This means that we re-introduce the variable x on the path from $\rho(\Delta)$ down to $\epsilon(x)$. Finally, Δ_x is obtained as the tree rooted at $\rho^* \stackrel{\text{def}}{=} \pi_\emptyset(\rho' \bowtie s_x)$, with the altered old root $\rho' = n_1'$ and s_x as successors.

It is easy to check that Δ_x is indeed an aggregation tree for C_x in the sense of Def. 10. Therefore, by (3.17) and Th. 1, $\rho^* \equiv \emptyset$.

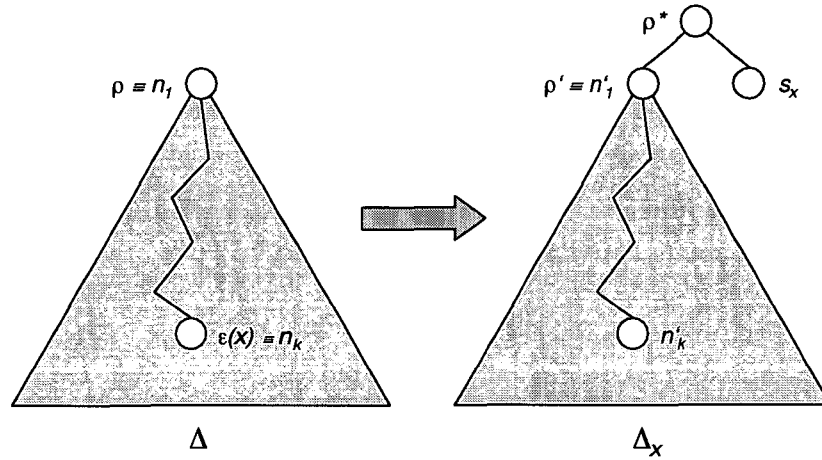


Figure 3.8: Construction of Δ_x from Δ

Example 7: The tree in Fig. 3.3, for the constraint problem C defined by the set of all leaf nodes, can quickly be altered to serve as an aggregation tree for C_x , where $x = i_{\text{BOTTOM}}$ is the variable to be explained. We just need to replace the root by the node $n \equiv (i_{\text{BOTTOM}} = 0.04)$, and introduce a new root \emptyset with the successors n and $s_x \equiv (i_{\text{BOTTOM}} \neq 0.04)$.

That new tree Δ_x can be utilised, according to Defs. 14 and 15, to compute all or just one minimal conflict, respectively. In the example, there is only one minimal conflict. It consists of the four constraints $v_{\text{HIGH}} = 12$, $v_{\text{LOW}} = 0$, $v_{\text{HIGH}} - v_{\text{LOW}} = i_{\text{BOTTOM}} \cdot 300$, and $i_{\text{BOTTOM}} \neq 0.04$. Theorem 5 implies then that there is hence exactly one minimal explanation for i_{BOTTOM} in C . It consists of the previously mentioned constraints, except $i_{\text{BOTTOM}} \neq 0.04$. Indeed, we can deduce $i_{\text{BOTTOM}} = 0.04$ simply by substituting v_{HIGH} and v_{LOW} by 12 and 0, respectively, in the equation $v_{\text{HIGH}} - v_{\text{LOW}} = i_{\text{BOTTOM}} \cdot 300 \iff 12 = i_{\text{BOTTOM}} \cdot 300 \iff i_{\text{BOTTOM}} = 0.04$.

3.4 Solving Sequences of Similar Constraint Problems

In the previous section, our concern was to solve any single given constraint problem, i.e. to decide consistency, derive tightest restrictions for all variables, and find

minimal conflicts and explanations, respectively. Whereas the underlying set of constraints was fixed, we shall now focus on varying sets of constraints.

This means that we will be presented a *context space*, i.e. a set of mutually distinct constraint problems rather than just a single problem, a *context*. However, we can only then hope to apply incremental solving techniques, in order to tackle the context space, if the contexts at hand facilitate a certain degree of similarity.

The focus of this work is mainly on engineering applications and the arising context spaces. Chap. 2 already addressed the issue of multi-contextuality, with the prominent examples *diagnosis* and *specification-driven design*. A common characteristic of those engineering tasks is that they decompose into large spaces of very similar contexts. The size makes clear that incremental solving techniques are inevitable; the similarity guarantees their applicability.

A high degree of similarity facilitates, moreover, means of pruning a given context space: If one context C_0 is found to be inconsistent, then all other contexts that contain a minimal conflict of C_0 must also be inconsistent, and can therefore immediately be disregarded.

3.4.1 Contexts and Context Spaces

Let us start with an example, in order to motivate the definition of a context space.

Example 8: We go back to the electric circuit depicted in Fig. 2.4. There, we have 7 component constraints that are instantiations of the constraints of Fig. 2.3; one instance of *Source*, *Wire* and *Ground*, and two instances of *Bulb* and *Node*. Additionally, there are 7 identifications, due to the 7 pairs of ports. In total, this gives a *system description* of 14 constraints, where we count the respective disjunctions of conjunctions (as in the case of a *Bulb* or *Wire*, and the conjunctions (all other cases) as single constraints.

A diagnosis tool is likely to be supported by additional information that stems from observations, and that can be expressed in terms of assignments to the variables L^{B1} and L^{B2} . It may then try to find assignments to the three behavioural mode variables M^W , M^{B1} and M^{B2} , so that the observations can be justified. This results in the task to determine which mode assumptions are consistent with the system description and the observations.

So, by collecting all relevant constraint problems, one obtains a context space of $2^3 = 8$ contexts, since each of the three mode variables can take two distinct values. Clearly, any two contexts have the system description and two observation assignments in common, i.e. $14 + 2 = 16$ constraints out of $16 + 3 = 19$. That allows us to speak of a high degree of similarity.

Suppose $L^{B1} = on$, $L^{B2} = off$ have been observed, then there is obviously only one consistent mode assumption; $M^W = ok$, $M^{B1} = ok$, $M^{B2} = broken$. Consideration of the mode assumption $(M^W, M^{B1}, M^{B2}) = (broken, ok, ok)$ should yield - among other minimal conflicts - the minimal conflict $\{M^W = broken, Wire^W, L^{B1} = on, Bulb^{B1}, L^{B2} = off, Bulb^{B2}, Node^{N1}\}$.¹¹ That would enable the diagno-

¹¹The first two constraints imply that the current through W is zero. The next two imply that the current through Fig. 2.4's top branch has a modulus of at least C . The following two constraints

sis tool to immediately disregard half of all mode assumptions, namely those that assign *broken* to M^W . M^W could right away be determined to be *ok*.

Definition 17 (Context & Context Space, etc.)

Let F be a possibly empty set of non-trivial constraints, and $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$, $n \in \mathbb{N}_+$, denote a set such that any $G_i \in \mathcal{G}$ is a non-empty set of non-trivial constraints. Then each set of constraints

$$F \cup \{g_1, g_2, \dots, g_n \mid \forall i \in \{1, 2, \dots, n\} \ g_i \in G_i\}$$

is called a **context**, and the set of all those constructions form a **context space**. F is named the **fixed portion** of the context space; each $G \in \mathcal{G}$ is termed **one-of** and $g \in G$ is an **alternative** of the one-of G .

In the above Ex. 8, the fixed portion F consists of 14 constraints that constitute the system description, and 2 observations, i.e. $|F| = 16$. Furthermore,

$$\mathcal{G} = \left\{ \begin{array}{l} \{M^W = ok, M^W = broken\}, \\ \{M^{B1} = ok, M^{B1} = broken\}, \\ \{M^{B1} = ok, M^{B1} = broken\} \end{array} \right\},$$

and \mathcal{G} contains 3 one-ofs. Choosing a certain context is obviously done by picking one alternative of each one-of, which motivates the above naming conventions.

Note that Def. 17 allows for an empty fixed portion, i.e. two contexts of a context space may have no constraint in common. This is however neither intended nor the typical setup in the engineering applications we are addressing here. Also, since the set of one-ofs is not permitted to be empty, nor any of those one-ofs, we will end up with a non-empty context space in which each context is a constraint problem according to Def. 7.

In order to efficiently analyse a context space with contexts of great similarity, we need to be able to identify reusable bits of computation. This is the subject of the next subsection.

3.4.2 Identification of Reusable Aggregation Subtrees

Suppose we are given a context space and have already built an aggregation forest Φ_0 for some initial context C_0 . Switching to a next context, C_1 , means to stick to the fixed portion, but also to replace a certain choice of alternatives by another choice. More generally, we need to remove the subset of constraints $C_0 \setminus C_1$ from C_0 , and afterwards add the set $C_1 \setminus C_0$. In the case of a context space as defined by Def. 17, those two sets will always have the same cardinality. However, the following argumentation also holds for the case of differing cardinalities.

Note that none of the backward relations that have possibly been assigned to nodes in Φ_0 , may be reused since they result from global knowledge about C_0 . So, our concern here is to determine reusable forward relations, only.

concerning $B2$ give a current through the lower branch with modulus less than C . But then, those three currents cannot add up to zero, as the final constraint dictates.

Obviously, any subtree the leaves of which are entirely contained in $C_0 \cap C_1$, embodies a valid computation for the new context C_1 . Fig. 3.9 illustrates how an existing aggregation tree for the problem depicted in Fig. 3.3 can be repaired in order to serve as an aggregation tree for the altered problem in which the observation $L = on$ has been replaced by $L = off$. The subtrees in the grey area may be reused right away; only one path needs to be recomputed. Those are the new nodes, marked by stars, that result from a bottom-up computation.

There is however a second important issue that has to be taken into account: Any

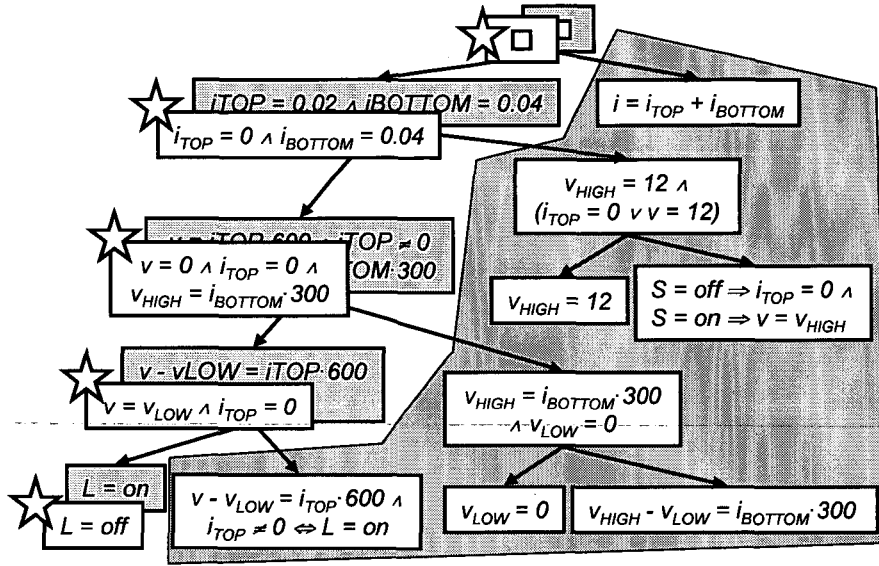


Figure 3.9: Repairing the Aggregation Tree of Fig. 3.3 by Reusing Subtrees

variable x mentioned by any newly-introduced constraint $c_1 \in C_1 \setminus C_0$ may already have been present in some old constraint $c_0 \in C_0$. Consequently, it was eliminated at the unique node $\epsilon(x)$ in Φ_0 . But if $\epsilon(x)$ is not on the path $\rho(\Delta_0) \xrightarrow{*} c_0$, where Δ_0 denotes the aggregation tree in Φ_0 that entails c_0 , then a simple path recomputation as shown in Fig. 3.9 will not produce a valid aggregation tree. That is because, after such a path recomputation, the altered tree will no longer facilitate the connectedness condition for x ; cf. Lem. 5, part 3.

This situation is exemplified in Fig. 3.10, where each variable may take its values in \mathbb{R} , and “ $t \in \mathbb{R}$ ” stands for the non-trivial but non-restrictive constraint $(\{t\}, \{(t \mapsto v) \mid v \in \mathbb{R}\})$. Here the leaf $x = 1$ is being replaced by $x = w + 1$ which suddenly also mentions w .

Still, both the problem of replaced leaves and the problem of altered scopes can be taken care of by a single procedure. That procedure assumes an aggregation forest Φ_0 for C_0 , and produces a valid aggregation forest Φ_1 for C_1 :

Step 1: For each abandoned leaf $c_0 \in C_0 \setminus C_1$ in a tree Δ_0 of Φ_0 , all nodes on the path $\rho(\Delta_0) \xrightarrow{*} c_0$ shall be marked as “to be abandoned”. An example is the left

path in Fig. 3.10.

Step 2: For each newly-introduced leaf $c_1 \in C_1 \setminus C_0$ and each $x \in \text{vars}(c_1)$, every node on the path down to $\epsilon(x)$ from the corresponding tree's root, shall be marked, too. This applies to the right path in the aggregation tree of Fig. 3.10.

Step 3: We collect all maximal subtrees in Φ_0 that have unmarked roots; cf. the grey area in Fig. 3.10. Moreover, the constraints contained in $C_1 \setminus C_0$ (in Fig. 3.10: $x = w + 1$) will be collected, too. The resulting set forms the new aggregation forest Φ_1 .

Note that, in Fig. 3.9, the marking according to the above step 2 will not mark any previously unmarked node, since $\epsilon(L)$ lies on the path that has been marked in step 1.

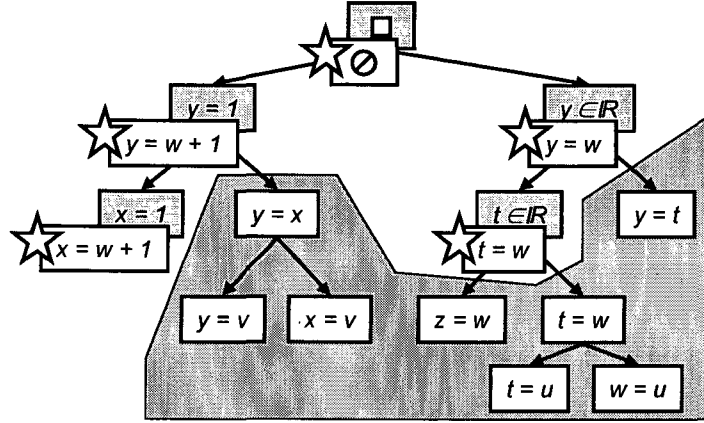


Figure 3.10: Additional Recomputation due to Altered Scopes

Lemma 8

With the above notations and procedure, we obtain an aggregation forest Φ_1 with the following properties.

1. Φ_1 is an aggregation forest for C_1 in the sense of Def. 10.
2. If two trees $\Delta_1, \Delta_2 \in \Phi_1$ share a variable x , i.e. $x \in \text{vars}(c_1) \cap \text{vars}(c_2)$ for some $c_i \in V(\Delta_i)$, $i \in \{1, 2\}$, then their roots also share that variable, i.e. $x \in \text{vars}(\rho(\Delta_1)) \cap \text{vars}(\rho(\Delta_2))$.

The lemma implies that we may decide consistency for the connected subsets of C_1 , according to Th. 1, simply by building aggregation trees from the root nodes in Φ_1 .

The natural language algorithm given above is going to be stated as pseudo-code in the next chapter.

So far, we have always assumed the existence of aggregation trees. For this chapter, the remaining question concerns the task of actually constructing one from a given set of constraints.

3.5 Aggregation Strategies

Figure 3.11 depicts the typical situation during the process of building an aggregation forest for some constraint problem C , $|C| \geq 2$. We are given a non-empty set of derived constraints, which form initially just the set C , and have to find two partial aggregation trees the roots of which are to be aggregated next.

In order to satisfy Def. 10, namely the part that guarantees the set of leaves of any aggregation tree to form a connected set, the relations r_1, r_2 in Fig. 3.11 need to share a variable, i.e. $\text{vars}(r_1) \cap \text{vars}(r_2) \neq \emptyset$; see also Lem. 5, part 4. If no such choice can be made then no two partial trees share a variable. In this case, the partial trees are the aggregation trees of the final aggregation forest. Moreover, each root relation must be a trivial constraint.

Besides this termination criterion which is due to the disconnectedness of C , there

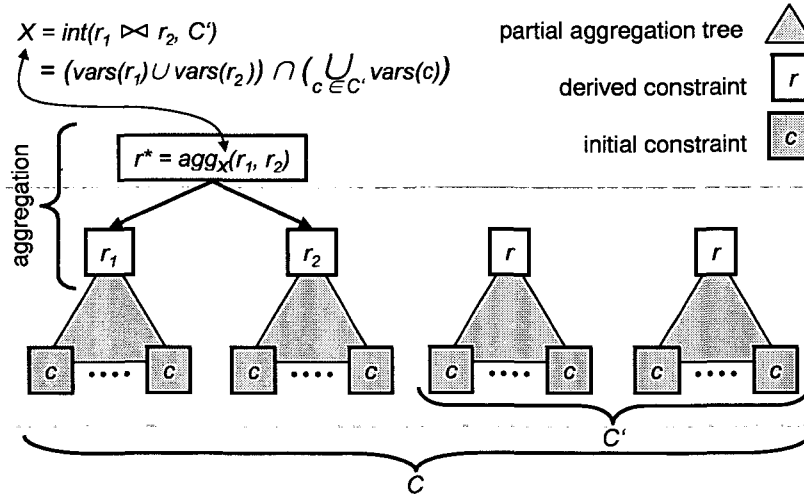


Figure 3.11: One Aggregation Step While Building an Aggregation Forest

are the following two criteria: The newly-created relation r^* may coincide with \emptyset in which case a connected component of C and hence C is proved to be inconsistent, according to Th. 1. Or, there is only one partial aggregation tree left. In that case, the aggregation forest has just one aggregation tree, and C is a connected constraint problem.

Figure 3.11 presents again the formula for computing the scope X of the new constraint r^* . It is clear that different choices of (r_1, r_2) will induce different relations r^* with different scopes X . However, basically any two partial aggregation trees that share at least one variable, may be chosen for the next aggregation. Consequently, any strategy that observes that condition, will produce a valid aggregation forest for C .

3.5.1 On-The-Fly Strategies

Let us call, in the setting of Fig. 3.11, any method that proposes a pair of partial aggregation trees for the next aggregation, an *on-the-fly aggregation strategy*. In order to facilitate subsequent ideas, on-the-fly strategies may, at run-time, not always be aware of the entire set of partial aggregation trees but only of proper subsets. In that sense, Fig. 3.11 presents only those partial trees that can be “seen” by the strategy. More precisely, such a strategy will be

given: a non-empty set of relations that are the roots of partial aggregation trees, and
a set of *protected variables*. These variables are known to appear in other partial aggregation trees which the strategy is not aware of. Observing the connectedness condition in Lem. 5, 3., protected variables must not be eliminated during the next aggregation. This clearly supports the notion of interfaces as defined in Def. 9. The strategy will then

return: a pair of relations of the given set, that share at least one variable; unless it announces an

exception: if the given set has just one element, or one of the given relations equals \emptyset , or no two given relations share a variable.

Since on-the-fly strategies “see” in general only *some* partial aggregation trees, and give a clue only as to what the operands of the imminent aggregation shall be, we may also call them *local aggregation strategies*. Diverse local aggregation strategies have been implemented and tested in our prototypic implementation of RCS in Java. However, the results have been omitted in this work.

The formalism developed in this chapter defines a rather general framework for solving constraint problems, based on the two operators *join* and *project*. So far, no problem-specific assumption has been entered in the elaborated train of thoughts. However, the introduction already emphasised that relational aggregation is likely to be a practicable concept whenever the constraint problems at hand are characterised by a *low density*: If each variable appears only in few initial constraints, then we are likely to eliminate variables more often, while building an aggregation forest. Figure 3.2 also raises the hope that any new relation r^* is of similar scope size as r_1 and r_2 . Consequently, on-the-fly strategies may choose a “best” pair of operands for the next aggregation, by investigating scopes, i.e. sets of variables.

The Minimal Scope and the Maximal Elimination Strategy

The aggregation shown in Fig. 3.12 combines two partial trees, the roots of which have certain scopes that still interact with the root relations of other partial aggregation trees. With the intuition that each root is the combined join of all leaves, projected onto an appropriate interface, cf. Lem. 5, 5.(a), we may view each root relation as a microprocessor with one pin for each of its variables.

Then, Fig. 3.12 illustrates the effect of one aggregation in terms of the involved

scopes. The aggregation eliminates b variables, leaving $a + c$ variables for the resulting new root.

Experiments have been carried out with on-the-fly strategies for choosing the next pair of operands due for aggregation, that always pick the pair

- which minimises $a + c$, i.e. the scope size of r^* , (**minimal scope strategy**); and
- which maximises b , i.e. the number of eliminable variables in accordance with the protected ones, (**maximal elimination strategy**).

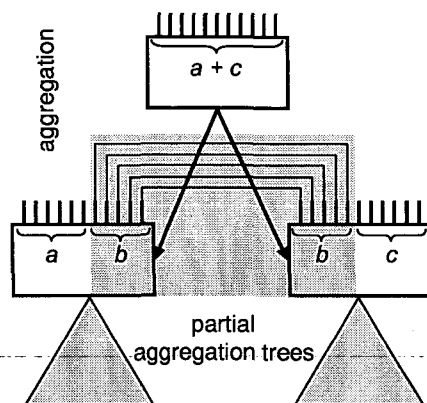


Figure 3.12: An Aggregation Step With Scope Sizes

The latter strategy is inspired by the intention to always simplify the remaining overall constraint problem in terms of involved variables. However, we see that $a + c$ and b are algebraically independent, and thus both strategies may deploy a very different behaviour. On the other hand, both could be combined by considering linear combinations or other functions of $a + c$ and b .

Linearising On-The-Fly Strategies

Clearly, exhaustive local strategies will have to consider and validate any pair of root relations of all provided partial aggregation trees, in order to propose a *best* one. *Best* here means best according to a certain heuristic, of which we have seen two above.

Assuming a set of n partial aggregation trees, this means to consider $\binom{n}{2}$ candidates, i.e. a quadratic amount. The following idea has been introduced in our prototypic implementation of RCS in order to decrease that effort to linear: We maintain a sorted list of n root constraints rather than an arbitrary list. Before the ordering criterion is explained, let us see how a local strategy is deployed utilising the implied ordering.

Starting at the head of the list, we look for a constraint r_1 that shares a

variable with at least one other constraint in the list. Hence, the set $R = R(r_1) = \{ r \mid \text{vars}(r_1) \cap \text{vars}(r) \neq \emptyset \}$ is not empty. Then, $r_2 \in R$ is chosen according to the respective on-the-fly strategy. Obviously, finding r_1 as well as r_2 takes only linear time. After having aggregated r_1 and r_2 , both have to be deleted from the list, and r^* has to be placed in the right position. This again takes only linear time. However, sorting m initial constraints is known to cost $O(m \cdot \log(m))$ but takes place only once, e.g. during the input of all constraints.

By what criterion shall constraints be ordered? In the Java implementation of RCS, each constraint can provide a measure of its *constrainedness*. This non-negative number, which may clearly be used to sort any given set of constraints, is computed by a heuristic that estimates how many assignments belong to the constraint. E.g. \emptyset will return 0, whereas \square is to return the largest number on the corresponding scale. Prominent constraints, like for example singleton assignments as $x = 1$, are considered highly constraining and return therefore a small estimate, that is, one close to zero.

Apart from enabling a linear effort for on-the-fly strategies, the introduction of the given constraint ordering is also a means to incorporate knowledge about the actual data encoded in the constraints; as opposed to purely structural information concerning scopes. So far, we have not paid attention to the actual data encoded in constraints, and shall come back to that issue when discussing the implementation of a *relational processor* in Chap. 5.

3.5.2 Clustered Aggregation Strategies

Aggregation trees encode a specific ordering in which the given constraints are combined: Before the roots of two subtrees are aggregated, the leaves of both subtrees need to be combined separately. The usage of local aggregation strategies will just produce some aggregation forest, and thereby some specific a posteriori ordering of aggregations. However, there are situations in which we might want to influence that ordering a priori.

E.g. when trying to solve a system of linear and non-linear equations in a naive way, see [9]; it seems a good strategy to consider the linear portion of the problem first. The resulting restrictions for some of the variables, will possibly simplify the non-linear portion in a beneficial way. This is an example where additional knowledge about the actual constraints can suggest a certain ordering of aggregations.

Figure 3.13 shows a *clustered aggregation strategy*; also called *global strategy* as opposed to the above local strategies. Any cluster, i.e. (sub-)tree rooted at a grey node, contains $s, s \geq 0$, embedded subclusters that are attached as subtrees; and $t, t \geq 0$, constraints of the initial constraint problem, where $s + t \geq 2$. Moreover, all constraints mentioned in a cluster, i.e. the leaves, must form a connected set. A later subsection will formalise and generalise the concept of clustered aggregation strategies; so here we shall only give some intuition and informal notion.

Each cluster will, at runtime, be interpreted as a bottom-up plan for producing an aggregation tree Δ for the constraint problem given by the set of leaves $\Lambda(\rho(\Delta))$. To this end, we need in general a local aggregation strategy σ .

1. All constraints in a terminal cluster, i.e. in a cluster without subclusters, will be used to build an aggregation subtree according to σ .
2. Furthermore, any non-terminal cluster builds a tree from its immediately succeeding constraints, and the subtrees that result from its subclusters. Again, this is done according to σ .
3. The variable set attached to the root of each cluster lists all protected variables, i.e. those that must not be eliminated while building the respective aggregation subtree utilising σ .¹²

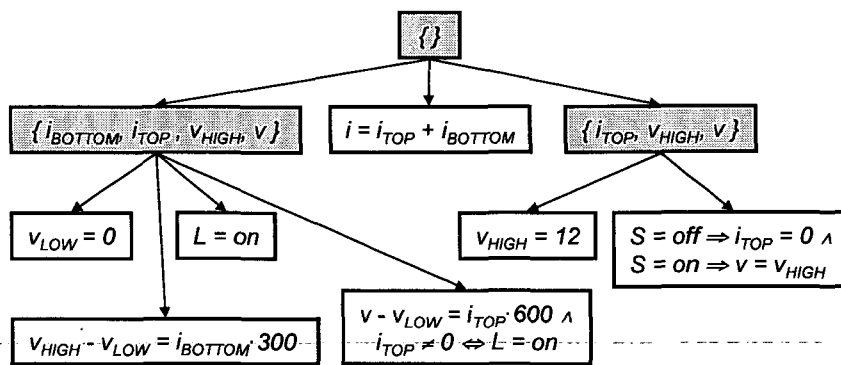


Figure 3.13: A Clustered Strategy for the Tree in Fig. 3.3

As an example, check that Fig. 3.13 shows indeed a clustered strategy that will, with an appropriate σ , generate the aggregation tree shown in Fig. 3.3.

Note that we shall only then need the local strategy σ , if there is a (sub-)cluster for which $s + t \geq 3$, with s and t as above. For otherwise, $s + t = 2$ for all (sub-)clusters, and the local strategy σ would, at each call, have to “choose” a pair of subtrees among a set of cardinality 2, and hence do nothing. As opposed to such a completely specified global strategy, the other extreme is the completely unspecified one. Here, we just have a root with an empty variable set attached, and as successors the set of all initial constraints. According to the above procedure, the entire work will then have to be done by σ .

3.5.3 Aggregation Strategies based on Decomposition Methods

Our prototypic implementation of RCS has been equipped with an interface for exporting constraint problems and importing applicable clustered strategies. This allows us to deploy all sorts of external strategy modules and utilise their outputs.

¹²Note that those variable sets can be omitted if each subcluster has, at runtime, complete information about the entire clustered strategy, and hence about all constraint scopes. However, in an object-oriented implementation this may not be favourable, and we might instead directly provide the set of protected variables for each cluster.

This aspect was also the content of a cooperation between the DaimlerChrysler company and the "Databases and Artificial Intelligence Group" at the Vienna University of Technology. The latter has gathered expert knowledge on *structural CSP*¹³ *decomposition methods*, see e.g. [31] for a detailed overview over the most prominent decomposition methods and their relationships according to the notion of comparability and predominance defined there. In the following, *acyclic* and *cyclic CSPs* will be mentioned as well as other, related terms. Those shall not be thoroughly defined in this work; the reader is asked to consult the above references, below pointers and further links established there.

Decomposition methods first appeared in the context of boolean conjunctive queries (BCQs) in the theory of relational databases, cf. [55] or [2]. Today, the mapping between conjunctive database queries and constraint satisfaction is well-understood, and consequently decomposition techniques have been and are vividly applied to CSPs.

In the case of an acyclic problem and only then, a so-called *join tree* can be found; see [55] or [31] for an exact definition. *Yannakakis algorithm*, originally developed for BCQs, can be adapted for finding all solution tuples of any given acyclic CSP; cf. [75] for the original algorithm. That sequential algorithm takes only polynomial time, whereas solving cyclic CSPs is known to be NP-complete. Moreover, solving algorithms based on join trees have been found to be highly parallelisable, [30].

For all those reasons, the goal is, in the case of a cyclic CSP, to generate a structure that is similar to a join tree. This is the common goal of all decomposition methods, that is, to provide efficient join schemes also for cyclic CSPs.

[31] proves that, according to the comparison criteria introduced there, *hypertree decomposition* turns out superior to all other prominent decomposition methods. Therefore, we shall concentrate here on that method. In what follows, we give an overview, and explain how a *complete hypertree decomposition* may be used to derive a clustered aggregation strategy for a connected constraint problem.

Complete Hypertree Decompositions

In order to give an example of a *hypertree decomposition*, let us recall the two families of electric circuits in Fig. 2.7. The top of Figure 3.14 shows part of the corresponding constraint hypergraph that captures one box B_i . The environment of that box is represented by the two constraints *before* and *after*.

We have already given a loose definition of a constraint hypergraph; see the remark that follows Def. 10. In Fig. 3.14, capital letters represent variables.¹⁴ Grey and white shapes depict the hyperedges which capture the respective constraints, as e.g. given in Fig. 2.8 for the family $\{D_k\}_{k>0}$.

The bottom part of Fig. 3.14 presents a *complete hypertree decomposition* for the above constraint hypergraph, in the so-called *atom representation*. Again, we shall not define those terms here; cf. e.g. [32]. Still, the defining properties of a complete hypertree decomposition shall be given in natural language:

¹³Constraint Satisfaction Problem

¹⁴Actually, each capital letter stands for a vector of one current and one voltage variable associated with the electric ports that have been omitted in the figure.

1. A hypertree decomposition (\mathcal{HD}) is a tree, the vertices of which are sets of hyperedges of the constraint hypergraph (\mathcal{CH}). Moreover, each variable of a hyperedge is either explicitly listed or replaced by an extra symbol. (Typically, the underscore “_” is used.)
2. For each hyperedge in \mathcal{CH} (i.e. for each constraint in the initial constraint problem), there is a vertex in \mathcal{HD} in which it appears with all variables listed. (This subsumes the completeness condition, cf. [32].)
3. For each variable X , the vertices in which X is listed form a subtree of \mathcal{HD} . (*connectedness condition*)
4. If a variable X is listed in some vertex v of \mathcal{HD} , then it must not be replaced by “_” in the predecessor of v .

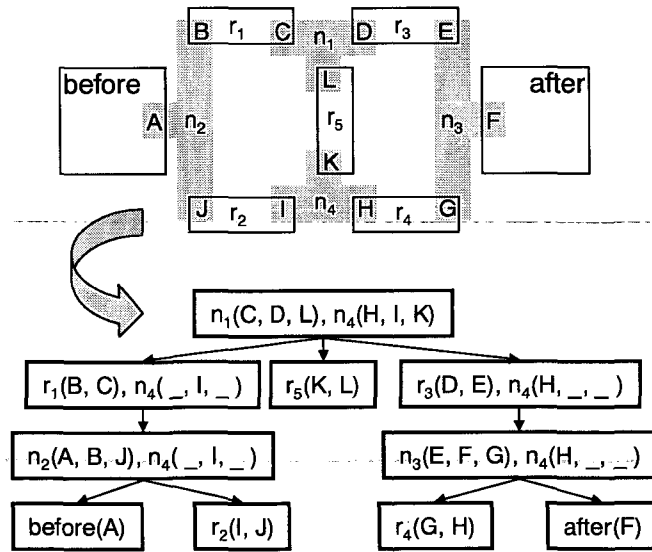


Figure 3.14: Hypergraph and Decomposition for the Families in Fig. 2.7

It is easy to verify the validity of all conditions in the example of Fig. 3.14. Note that the tree has been derived basically by unseaming the constraint hypergraph in the hyperedge n_4 . However, n_4 needs then to be added to numerous vertices of the decomposition, in order to maintain the connectedness condition for the variables I and H . This sort of clustering is the appropriate measure to break cycles in cyclic hypergraphs. The maximal number of hyperedges clustered in a single vertex is called the *hypertree width* of the tree at hand. The minimal number of all those tree-specific numbers is the hypertree width associated with the original constraint problem. [32] shows that CSPs with their hypertree width bounded by a constant, can be solved in polynomial time, re-establishing Yannakakis' result for infinitely many classes of cyclic instances.

Since only acyclic hypergraphs have a width of 1, and since the hypergraph in

Fig. 3.14 is cyclic, the problem's width must be at least 2. The given decomposition shows that 2 can indeed be attained, and thus the width of the problem is deduced to be 2.

Variables that have been replaced by “_”, will be interpreted as being eliminated from the respective constraint. Therefore, hypertree decompositions will typically involve not only all initial constraints but also some of their projections. Still, the above condition 2 ensures that each initial constraint will appear at least once without any variable eliminated.

The next paragraph shows how a hypertree decomposition, once provided by some external module, naturally induces a clustered aggregation strategy that can be used to build an aggregation tree for the initially given constraint problem. The arising aggregation tree needs to facilitate the connectedness condition of Lem. 5, 3., which is guaranteed by the above connectedness condition 3 for hypertree decompositions.

Conversion of a Complete Hypertree Decomposition into a Clustered Aggregation Strategy

Figure 3.15 shows the clustered aggregation strategy that results from the hypertree decomposition (\mathcal{HD}) presented above, in Fig. 3.14. The corresponding algorithm is

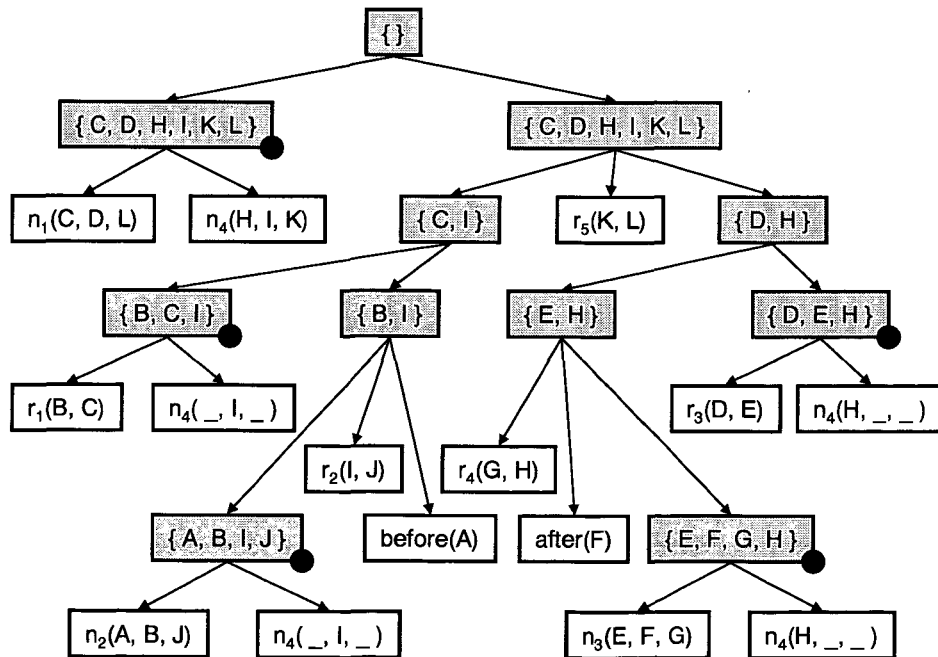


Figure 3.15: Clustered Strategy Derived from the Decomposition in Fig. 3.14

straight-forward and translates \mathcal{HD} in a bottom-up manner. Let $\theta(n)$ denote the translation of a single node $n \in V(\mathcal{HD})$, and $\Theta(n)$ the translation of the (sub-)tree

rooted at n . Then the following building blocks define an algorithm for translating the entire tree \mathcal{HD} .

1. $\theta(n)$, $n \in V(\mathcal{HD})$: If n hosts just one constraint, then $\theta(n)$ is that constraint; see e.g. $r_5(K, L)$ in Fig. 3.15.
Otherwise, if n consists of at least two constraints, then $\theta(n)$ is a tree Δ with the constraints of n as its successors. (In Fig. 3.15, the roots of those subtrees are marked with a black dot. For example, the translation of the node containing $n_3(E, F, G), n_4(H, \neg, -)$ is the bottom right subtree.) The root of Δ has attached a set of variables which is defined according to 4.
2. $\Theta(\lambda)$, $\lambda \in \Lambda(\rho(\mathcal{HD}))$: We define $\Theta(\lambda) \stackrel{\text{def}}{=} \theta(\lambda)$, for each leaf node λ of \mathcal{HD} .
3. $\Theta(n)$, $n \in V(\mathcal{HD}) \setminus \Lambda(\rho(\mathcal{HD}))$: For any non-leaf node n , $\Theta(n)$ is a tree Δ that contains $\Theta(n')$ as a subtree, for each successor n' of n . Additionally, Δ contains the translation of n itself, i.e. $\theta(n)$, as a subtree. Again, $\rho(\Delta)$ hosts a set of variables according to 4.
4. Attached to each non-leaf node n is a set of variables. It contains the interface of the combined join of all leaves in $\Lambda(\Delta(n))$, with respect to the remaining set of leaves. For example, the variable set $\{E, H\}$ in the middle of Fig. 3.15, is the portion of $\{E, F, G, H\}$ that also appears outside the corresponding subtree.

The given translation procedure is relatively simple, still some remarks have to be made.

Starting from a connected constraint problem C , a hypertree decomposition may introduce projections of one or more constraints. Therefore, the initial problem may be replaced by a problem

$$C' \stackrel{\text{def}}{=} C \cup P, \quad |P| \geq 1,$$

where P contains projections of some constraints of C . The question is, how can an aggregation tree built for C' be used to obtain information concerning C ?

First, we have equality of the sets of solution tuples, that is

$$\Sigma(C) = \Sigma(C'). \quad (3.18)$$

This follows easily, since (3.9) applied to $c_1 \equiv \square$ with $Y_1 \subseteq Y_2 = X_2$ shows that $c \equiv \pi_Y(c) \bowtie c$, for any proper subset $Y \subset \text{vars}(c)$. Moreover, that implies that C is consistent if and only if C' is. Therefore, both forward and backward relations computed for C' are valid also for C .

However, minimal conflicts and minimal explanations, respectively, that will be derived for C' may involve elements of P and thus not directly be usable for C . Clearly, if we compute *all* minimal conflicts or explanations for C' , those sets will especially contain all minimal conflicts or explanations, respectively, for C , since $C \subseteq C'$. The following example throws light on the problem of computing *one* minimal conflict for C from one that had been found for C' .

Example 9: Let

$$C = \left\{ \begin{array}{lll} c_1 & \equiv & w = 1 \wedge x = 1 \wedge y = 1, \\ c_2 & \equiv & z = x + y \wedge w = 2, \\ c_3 & \equiv & z = 3 \end{array} \right\}$$

be the initial constraint problem which is clearly inconsistent. Each variable takes its values in \mathbb{R} . Let P contain the projections $\pi_{\{x\}}(c_1), \pi_{\{y\}}(c_1), \pi_{\{x,y,z\}}(c_2)$, i.e.

$$P = \{x = 1, y = 1, z = x + y\}.$$

Obviously, $K' \stackrel{\text{def}}{=} \{c_3\} \cup P$ is a minimal conflict of $C' \stackrel{\text{def}}{=} C \cup P$, but not of C . The natural approach for deriving a minimal conflict K for C from K' should be abstraction: We replace each $\pi_X(c) \in P \cap K'$ by the initial constraint $c \in C$. In our example this yields the constraints c_3, c_1, c_1, c_2 . The replacement procedure will hence introduce duplicates. But just cancelling those duplicates will in general not suffice, as $\{c_3, c_1, c_2\}$ is still not a minimal conflict of C , because $c_1 \bowtie c_2 \sim \emptyset$, and c_3 can thus be suspended.

This example illustrates that we need to deploy firstly abstraction, secondly cancellation of duplicates, and thirdly additional suspension, in order to compute K from K' . Proving that these three steps will indeed produce a minimal conflict for C , is unproblematic and not carried out here.

3.5.4 Generic Aggregation Strategies

The focus of the previous subsections was on local and global strategies for single constraint problems. We shall now take a look at strategies which are valid not only for one problem, but for an entire context space, as defined in Def. 17. Moreover, the previous ideas of clustered strategies and of introducing projections of initial constraints, as e.g. done by many decomposition methods, shall be incorporated. This will guide us to the more general *generic aggregation strategies*. We will give a formal definition, and state a translation function that builds an aggregation tree for any connected constraint problem, that is, for any context of a context space. The following context space, with the notation as in Def. 17, will serve as a running example, in order to explain and illustrate generic strategies:

$$\Gamma = \{\mathcal{F} \cup \{r(x)\}, \mathcal{F} \cup \{s(x, y)\}, \mathcal{F} \cup \{t(x, y)\}, \{\mathcal{F} \cup \{u(y)\}\}, \text{ where } \mathcal{F} = \{a(x, z), b(x, z), c(x, z), d(x, z), e(y, z)\}.$$

The set \mathcal{F} is hence the fixed portion, whereas $\{r(x), s(x, y), t(x, y), u(y)\}$ is the sole one-of of the context space Γ . The functional notation is intended to make the relations' scopes explicit, e.g. $\text{vars}(s(x, y)) = \{x, y\}$.

Figure 3.16 shows a generic strategy for Γ that incorporates, besides the above relations, also a projection of an element of \mathcal{F} , namely of $e(y, z)$, and a projection of the one-of. The graph is a directed, acyclic graph (DAG) with exactly one root, that is, a node without predecessors.¹⁵ The terminal nodes are the elements of \mathcal{F}

¹⁵With what has been defined so far, *acyclicity* means that for any two nodes n, m , at most one of the paths $n \xrightarrow{*} m, m \xrightarrow{*} n$ may exist, but not both.

and the alternatives of the one-of. In Fig. 3.16, those are white as opposed to the partly grey, non-terminal ones. There are *cluster nodes* (labeled “C”), *projection nodes* (labeled “ π ”), and a *one-of node* (with the disjunction operator “ \vee ” as label). As in the above subsection about clustered strategies, the embedding of subclusters

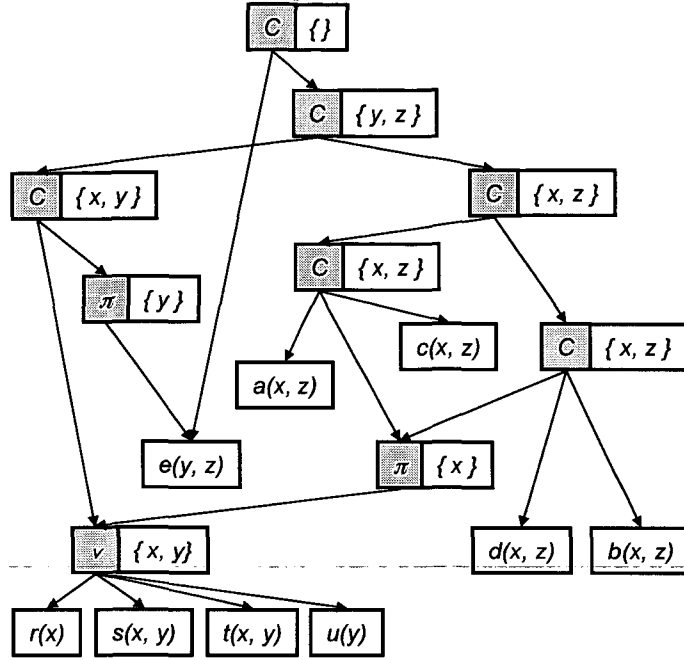


Figure 3.16: A Generic Aggregation Strategy for Γ

is done by attaching succeeding subgraphs. Due to the presence of projections, some constraints participate more than just once, and so we abandon the use of trees in favour of DAGs.

Let us fix an accurate definition:

Definition 18 (Generic Aggregation Strategy)

Let \mathcal{F} and \mathcal{G} with $|\mathcal{F}| + |\mathcal{G}| \geq 2$ be the fixed portion and the set of one-ofs, respectively, of a context space Γ . A **generic aggregation strategy** for Γ is a DAG Υ that may contain four sorts of nodes; cluster nodes, projection nodes, one-of nodes and plain constraints. The former three sorts of nodes host a set of variables, $X(n)$.¹⁶ For the benefit of convenience let us extend $X(\cdot)$ to nodes n in Υ that are plain constraints, by writing $X(n) = \text{vars}(n)$. Then Υ satisfies the following conditions:

1. There is exactly one node $\rho(\Upsilon)$ in Υ that has no predecessor, called the root.
2. The set of nodes without successor, called the leaves, coincide with the plain constraints in Υ . They must equal the set $\mathcal{F} \cup \mathcal{G}$.

¹⁶In Fig. 3.16, those attached sets of variables coincide with the nodes' actual contents. For the sake of understanding, the reader should also verify the listed properties of Υ using that figure.

3. For any one-of node n , the set N' of successors consists of constraints. Moreover, $N' \in \mathcal{G}$, i.e. N' must form a one-of. $X(n) = \bigcup_{n' \in N'} X(n')$ has to hold. Furthermore, for any one-of $G \in \mathcal{G}$ and any alternative $g \in G$, g must appear, that is, must be the successor of some one-of node in Υ .
4. The sole successor n' of a projection node n must be either a constraint, or a one-of node. Moreover, $X(n) \subset X(n')$, explicitly excluding equality.
5. Each cluster node n has at least two successors and at most one predecessor. With N' denoting the set of successors of n , and M the set of leaves Λ of Υ for which no path $n \xrightarrow{*} \lambda$ exists, $X(n) = (\bigcup_{n' \in N'} X(n')) \cap (\bigcup_{m \in M} X(m))$.
6. For each variable x assigned to some node in Υ , a **scope condition** must hold: There is a subgraph of Υ that is a tree Δ for which the following conditions are valid:
 - (a) A constraint n of Υ belongs to Δ if and only if $x \in X(n)$.
 - (b) Any one-of node or cluster node n for which $x \in X(n)$, belongs to Δ .
 - (c) Conversely, if a cluster node n belongs to Δ then either $x \in X(n)$, or $n = \rho(\Delta)$.
 - (d) Δ does not contain a projection node of Υ .

The scope property is similar to connectedness conditions, as previously defined for aggregation trees and mentioned for hypertree decompositions. Figure 3.17 illus-

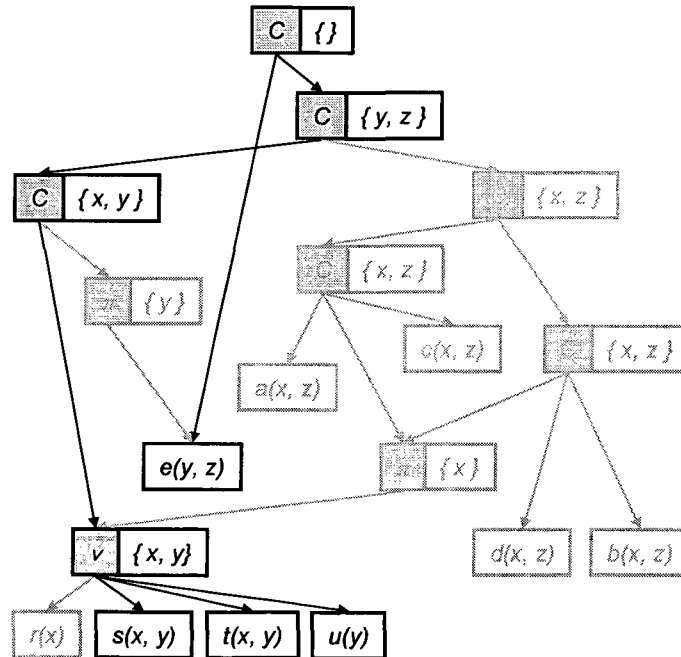


Figure 3.17: Illustration of the Scope Property for y in Fig. 3.16

trates that property for the variable y in the strategy of Fig. 3.16. The part of Υ that does not belong to the tree Δ has been faded out.

The following lemma repeats and extends the already given translation scheme, for obtaining an aggregation tree from a clustered aggregation strategy. Basically, we just need to add the translations of one-of nodes and projections:

Lemma 9

Let Γ be a context space as in Def. 18, Υ a generic aggregation strategy for Γ , and $C \in \Gamma$ be an arbitrary connected context. Suppose furthermore, that σ is a local aggregation strategy.

The following properties define a function Θ_C^c , Θ for short, that returns an aggregation tree for any node in Υ :

1. For any leaf λ of Υ , set $\Theta(\lambda) \stackrel{\text{def}}{=} \lambda$, i.e. the single-node tree that consists only of the constraint λ .
2. For any one-of node n , let $\Theta(n) \stackrel{\text{def}}{=} \Theta(n')$, where n' is that successor of n that corresponds to the appropriate alternative, as chosen in the context C .
3. Suppose n is a projection node in Υ and n' its sole successor node. Then $\Theta(n')$ is, according to 1. and 2. a single-node tree. Let c denote the single constraint, then define $\Theta(n)$ to be the single-node tree with the constraint $\pi_X(c)$, where $X = X(n) \cap \text{vars}(c)$.
4. The remaining case deals with a cluster node n that has the set of successors N' : Let here $\Theta(n)$ be the aggregation tree built from the trees $\{\Theta(n') \mid n' \in N'\}$ according to σ and protecting, i.e. not eliminating, the variables in $X(n)$.

Then, $\Theta(\rho(\Upsilon))$ is an aggregation tree for some constraint problem C' such that $\Sigma(C') = \Sigma(C)$.

A brief remark concerning Lem. 9, 3., shall be made. At first sight, one should expect here the projection of c onto $X(n)$. The intersection of $X(n)$ with $\text{vars}(c)$, as stated above, has a technical reason that becomes clear by taking a look at the one-of node n' of Fig. 3.16, and its preceding projection node n : Suppose the context C would pick the alternative $u(y)$. Then the computation of $\Theta(n)$ involves projecting $u(y)$ onto $X(n) \cap \{y\} = \{x\} \cap \{y\} = \emptyset$. This shows why we cannot just project $u(y)$ onto $X(n) = \{x\}$ which would in this case be undefined; see Def. 5.

However, most one-ofs in practical context spaces will normally consist of alternatives that all have the same scope, being just mutually distinct assignments to discrete variables modelling behavioural modes, switch positions and the like. In that case, the intersection X , given in Lem. 9, 3., will coincide with $X(n)$ since then $X(n) \subseteq \text{vars}(c)$. So, the situation of Fig. 3.16, is rather artificial.

This ends the investigations on local and global aggregation strategies, and closes the formal framework for relational aggregation.

Chapter 4

Architecture and Algorithms of a Relational Engine

The previous chapter has paved the way for a concrete implementation of a relational engine that is based on a generic view on constraints; relations. This chapter is a guide to the implementation of a relational constraint solver that assumes serviceable join and project operators for relations. The provision of those two core methods is going to be the subject of the subsequent chapter.

The following sections will present a class model and pseudo-code for the most important algorithms, as suggested by the theorems of Chap. 3. We shall also take a look at their respective complexities, assuming the previously mentioned low density of constraint problems arising from model-based engineering problems.

The chapter is moreover dealing with issues of soundness and completeness of concrete realisations of RCS and provides the respective formalism.

4.1 Overview

Our prototypic implementation of RCS can be roughly divided into two parts: a master layer called *relational engine* and a slave layer named *relational processor*. Basically, besides other services provided by the processor, the latter hosts classes for representing trivial and non-trivial relations and implements the core operators join and project, as defined in Defs. 4 and 5. In Fig. 4.1, this relational processor is strongly abstracted in that it is represented by just a single class, **Relation**. All other classes shown in the figure belong to the relational engine. Our actual Java implementation uses some additional classes which shall be omitted in the picture, for the sake of simplicity.

Figure 4.1 follows the patterns and syntax defined by the *object modelling language UML*; see [29]: The top of each box shows the name of the class. The middle section lists *attributes* or sometimes called *members* of each of the class's instances. The bottom part contains the most important operations implemented for each instance of the class. A class may be the *generalisation* of other classes, as e.g. **Node** being the *superclass* of its *subclasses* **LeafNode** and **AggNode**. Classes may be linked by so-called *associations* that can mention *roles* and *multiplicities*. For example, any 1 instance of **AggNode** is going to be the *father* of exactly 2 *subtrees* which will be

represented by their root nodes, cf. Fig. 4.1. In our implementation, `Node` is an abstract class which means that there cannot exist instances of it, but only of its two subclasses.

As for the notation of pseudo-code, we shall use a common way of writing that is inspired by object-oriented programming languages, as for instance Java. Hereby, the consecutiveness of method calls is noted by connecting them via a dot. So, e.g. writing `obj.method1(arguments1).method2(arguments2)` means the application of `method2` with the arguments `arguments2` to the result of applying `method1` with arguments `arguments1` to the initially given object `obj`. The prefix `obj.` can be omitted if `obj` and the instance for which the respective pseudo-code procedure is being called are identical. Still, it may ease understanding to explicitly mention that instance by writing `this`.

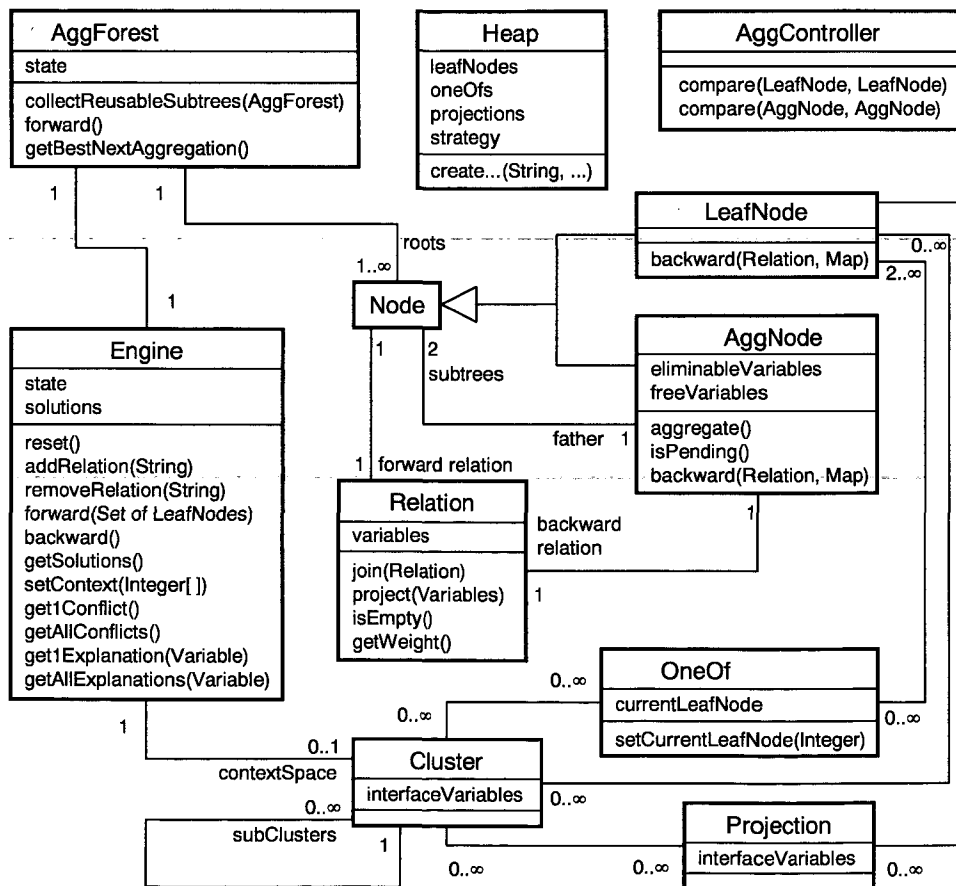


Figure 4.1: UML Diagram of a Relational Engine

Before we take a more thorough look at the classes in Fig. 4.1 and their relationships, it is important to make some remarks on *soundness* and *completeness*:

4.1.1 Approximation of the Formal Framework

The formal framework developed in Chap. 3 gives rise to statements that support the decision of consistency, see Th. 1, and the derivation of tightest bounds, see Th. 2.

However, no concrete implementation is going to realise those theoretical results. Implementational shortcomings, like e.g. the inability to eliminate a certain variable in a non-linear system of arithmetic equations, may result in the solver's inability to actually determine consistency for a given constraint problem. We shall come back to this point in the next chapter.

First, we shall point out that our prototype is not allowed to leave the problem of consistency unanswered. Consequently, it has to answer that question either positively or negatively, even when it does not know for sure. In our Java implementation, **any problem that is too hard to be determined, will be judged consistent**. Hereby, the intention is the following: The *default* for any analysed constraint problem is its consistency, unless the solver manages to prove the contrary. As long as the implementation fails to prove inconsistency or has not yet gathered sufficient evidence for inconsistency, the given problem will be assumed consistent. This can be seen as inspired by the engineering task of diagnosis; see Chap. 2: When there is no reason to suspect one or more components to be broken, the entire diagnosed system will be considered to function.

There is another issue concerning consistency. It concerns the operations implemented at **Relation** that form the basis for all high-level algorithms of a relational engine.

Theorem 1 decides consistency purely by syntactical means of distinguishing the two trivial constraints \emptyset and \square . However, a concrete implementation can be made more efficient by providing a predicate **isEmpty** for any **Relation**; see Fig. 4.1.

A poor implementation of **isEmpty** will answer **true** only in the case that the instance of **Relation** at hand coincides with a representation of \emptyset , re-establishing the situation in Th. 1. A more advanced version may perform some simple quick checks, and, e.g., discover that the conjunction $t_1 = t_2 \wedge t_1 \neq t_2$ cannot have a solution, no matter what the terms t_1, t_2 are. Hence, **r.isEmpty()** is to answer **true**, whenever, for the constraint c represented by **r**, $c \sim \emptyset$ can be proved. Otherwise, following a defensive strategy, **false** must be returned. Note that this fits nicely in the above characterisation of our prototype, with consistency as default.

The advantage of a more advanced version of **isEmpty** is that, during the process of building an aggregation tree, a conflict may be discovered earlier. As a consequence, finding one or all minimal conflicts becomes cheaper due to a "smaller" tree upon which the respective algorithms are based.

However, a more advanced implementation will, at some point, also consume a significantly higher amount of runtime; and we end up with a typical trade-off problem. Interestingly, the purpose of **isEmpty** is the same as that of the entire solver; to decide consistency, even though on a lower and more immediate level. Therefore, we should stick to cheap syntactical checks in the implementation of **isEmpty**.

Besides being conservative in announcing inconsistency, our implementation of the basic operations **join** and **project** compute *overestimations* of the actual relations. The term *overestimation* is going to be used more often in what follows. Thus, let us capture this notion in a formal definition.

Definition 19 (Implication of Constraints)

Let $c = (X, \mathcal{A})$ and $d = (Y, \mathcal{B})$ be two non-trivial constraints. We say that c *implies* d , denoted by $c \rightarrow d$, if and only if the following two conditions hold.

1. $X \cap Y \neq \emptyset \implies \{\alpha|_{X \cap Y} : \alpha \in \mathcal{A}\} \subseteq \{\beta|_{X \cap Y} : \beta \in \mathcal{B}\}$
2. $Y \setminus X \subseteq \text{free}(d)$

For any constraints c, d , the formulae $\emptyset \rightarrow d$ and $c \rightarrow \square$ are defined to be true. Furthermore,

$$\begin{aligned} c \rightarrow \emptyset &\stackrel{\text{def}}{\iff} c \sim \emptyset, \text{ and} \\ \square \rightarrow d &\stackrel{\text{def}}{\iff} d \sim \square. \end{aligned}$$

If c implies d , we also say that c is *more restrictive* than d , and d is *weaker* or *less restrictive* than c . Moreover, d is said to be an *overestimation* of c .

In order to prepare ourselves to prove a further main result of this work, we need to verify some calculation rules for implications. The next lemma states that the validity of an implication is independent from the representatives of the respective equivalence classes, (4.1). Furthermore, implication is transitive, (4.3). And, finally, join and project turn out *monotonous* with respect to implication: (4.4) and (4.5) state that both operators yield, for weaker arguments, weaker results.

Note that all formalisations given in this section could have been included in Chap. 3, where we elaborated the formal framework of RCS. However, Def. 19 and the consequences listed here address aspects of concrete implementations of RCS that must handle phenomena as approximated constraints in the sense of overestimations. Therefore, all those results have been placed here. The connection between the theoretical conception of RCS and their practical realisations by means of concrete implementations constitute also the core of Th. 6, below.

Lemma 10

In what follows, all placeholders denote constraints, unless stated otherwise. Then,

$$c \sim c' \wedge d \sim d' \wedge c \rightarrow d \implies c' \rightarrow d', \quad (4.1)$$

$$c \rightarrow d \wedge d \rightarrow c \iff c \sim d, \quad (4.2)$$

$$c \rightarrow d \wedge d \rightarrow e \implies c \rightarrow e, \quad (4.3)$$

$$c_1 \rightarrow c_2 \wedge d_1 \rightarrow d_2 \implies c_1 \bowtie d_1 \rightarrow c_2 \bowtie d_2 \text{ and} \quad (4.4)$$

$$c \rightarrow d \implies \pi_{X \cap \text{vars}(c)}(c) \rightarrow \pi_{X \cap \text{vars}(d)}(d), \quad (4.5)$$

for any set of variables X .

We are now prepared to prove a further main result concerning RCS:

Theorem 6 (Monotonicity of the RCS Framework)

Suppose, we are given a concrete implementation \mathcal{I} of RCS with operations *join* and *project* that always produce overestimations of the relations defined by Defs. 4 and 5. Be C , $|C| \geq 2$, a constraint problem and assume that \mathcal{I} computes all forward relations, backward relations and tightest restrictions according to the formal framework given in Chap. 3. Then

1. Any forward relation computed by \mathcal{I} is an overestimation of the respective theoretical result.
2. If \mathcal{I} computes a forward relation that equals \emptyset , then C is indeed inconsistent.
3. In the case of consistency, \mathcal{I} produces overestimations for all backward relations and for all tightest restrictions.

Proof:

Lemma 10 provides us with the monotonicity of both *join* and *project*; cf. (4.4) and (4.5). The respective implementations of \mathcal{I} are assumed to always overestimate the theoretical result, according to Defs. 4 and 5. A simple inductive argument shows then that also arbitrary successive invocations of the methods *join* and *project* produce overestimations of all respective theoretical relations.

By taking a look at Def. 10, we note that any forward relation is the result of successive applications of *join* and *project*. Therefore, all forward relations computed by \mathcal{I} must - by the above argument - be overestimations of the theoretical results. Likewise, by recalling Def. 12 and Th. 2, one observes that the same is true for all backward relations and tightest restrictions. This proves the items 1. and 3.

Suppose now, \mathcal{I} computes \emptyset as a forward relation. According to item 1., this must be the overestimation of the correct result r , i.e. $r \rightarrow \emptyset$. But then Def. 19 yields $r \sim \emptyset$. Hence, processing a subset of C by means of *join* and *project* as defined in Defs. 4 and 5, reveals a conflict. Therefore, C must be inconsistent, proving item 2. This completes the proof of the theorem. q.e.d.

Since, the precondition is true for our prototype (see above), we are going to obtain, for each appearing variable x , an overestimation of its tightest restriction $t_x = (\{x\}, \mathcal{A}_x)$. Note that, for poor implementations of *join* and *project*, we may even end up with the trivial overestimation $(\{x\}, \{(x \mapsto v) \mid v \in \text{dom}(x)\})$. The aim of a good implementation is therefore to approximate t_x by some $\bar{t}_x = (\{x\}, V_x)$ such that $V_x \supseteq \mathcal{A}_x$ and $V_x \setminus \mathcal{A}_x$ is as "small" as possible.

4.1.2 Soundness and Completeness

The typical means for capturing approximations as the ones described in the previous section, are the terms *soundness* and *completeness*. However, one must be careful, since converse definitions exist.

In the sense of Tsang's definition of *soundness*; see [70]; our implementation of RCS is not sound since it may overestimate the empty set of solutions and claim that

there exists a solution although there is actually none. However, with regard to other common definitions; cf. e.g. [64, p. 131-132]; as usually used in the diagnosis community, our implementation would be considered sound. This is because whenever it claims inconsistency of a constraint problem C , C will indeed be inconsistent in the sense of Def. 7; cf. Th. 6, 2. Conversely, not all inconsistencies will necessarily be discovered, implying that our implementation is, in the sense of [64, p. 131-132], not *complete*.

[70] also provides a definition of *completeness*, ensuring that any existing solution will actually be found by the constraint solver. In that sense, our implementation of RCS is complete since the computed restriction for a variable x overestimates t_x .

Summarising, according to Tsangs definitions, our prototypic implementation of RCS is *not sound but complete*. Those definitions focus on the set of solutions to a constraint problem, whereas other definitions, as used by the diagnosis community, focus mainly on the problem of deciding consistency. With regard to the latter, our prototype is *sound but not complete*.

Other existing frameworks clearly commit themselves to Tsangs definitions; see e.g. [72, pp. 113-115]. Instead of making a choice, we will avoid the usage of the terms *soundness* and *completeness* and rather speak, for example, of overestimations as defined in Def. 19.

4.1.3 The Class Engine

Certainly, the most important class of our relational engine is the class **Engine**; see Fig. 4.1. It maintains a pool of active constraints that can be manipulated using the operations

- **reset**, for clearing the pool,
- **addRelation**, for adding, and
- **removeRelation**, for removing a pre-defined, named constraint from the pool; see also the subsection concerning the class **Heap**.

Associated with any instance of **Engine** is a **state** that can be either **unknown**, **inconsistent**, **consistent** or **solved**, reflecting gathered knowledge as to whether the constraint problem formed by the pool of active relations is known or not known to be consistent or not, and whether solutions have already been computed. Of course, whenever a pool of constraints has been found to be consistent, and we remove a constraint afterwards, the resulting, decreased pool represents a relaxed constraint problem that must hence still be consistent. This as well as all other transitions deploying **reset**, **addRelation** and **removeRelation** is depicted in Fig. 4.2. This figure takes the view on an instance of **Engine** as a finite state machine with the mentioned transitions.

Figure 4.2 includes, in a lighter grey, the further operations **forward** and **backward**. **forward** is to determine consistency for any given pool of active constraints by building an aggregation forest. An **Engine** will maintain a link to that aggregation forest, represented by an instance of **AggForest**. **Forward** implies a transition to either the

consistent state or the inconsistent one. As has been explained in a previous subsection, the **Engine** is not allowed to remain in the **unknown** state after imposing **forward**. We shall come back to the implementation of a consistency check in the next section.

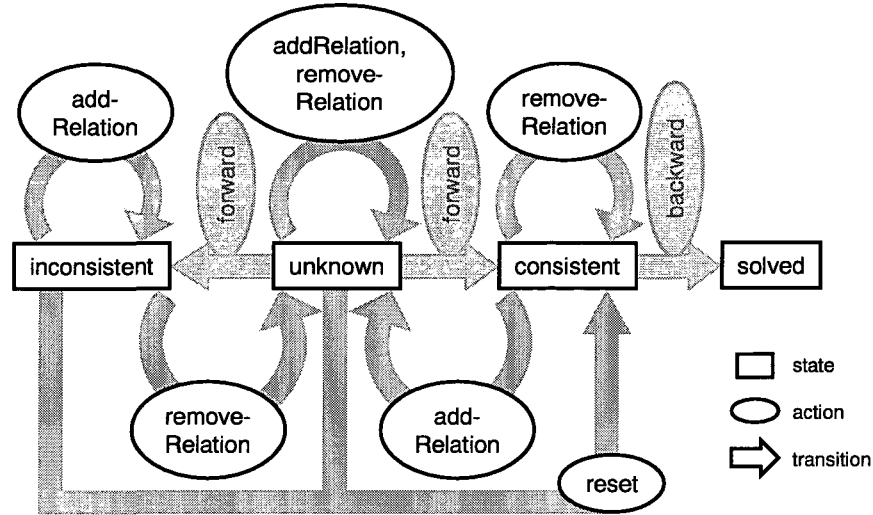


Figure 4.2: The Relational Engine seen as a Finite State Machine

Suppose an instance of **Engine** has detected consistency, then asking for the solutions of the constraint problem at hand makes sense. Deploying the operation **backward** will fill the engine's attribute **solutions** that represents a mapping

$$\begin{aligned} X &\longrightarrow \mathcal{P}(\bigcup_{x \in X} \text{dom}(x)), \\ x &\longmapsto V_x \subseteq \text{dom}(x), \end{aligned}$$

where $X = \bigcup_{c \in C} \text{vars}(c)$ contains all appearing variables, C is the set of constraints in the pool.¹ As a further consequence of that action, the state will be altered from **consistent** to **solved** which can only be reached from **consistent**; cf. Fig. 4.2.

Backward will be discussed in more detail in a subsequent section. Clearly, as has already been stressed, for any appearing variable x , the set of values V_x is to approximate from above the corresponding values in t_x .

After a call of **backward**, a user may retrieve all approximations at once, via the method **getSolutions** which is just an accessor of the attribute **solutions**. An additional operation **getSolution(Variable)** is thinkable, retrieving V_x for the parameter variable x . In Java, **solutions** will typically be represented by some implementor of the *interface* **Map** which maps so-called *keys* to *values*. Good implementations of **Map** allow for the retrieval of a value - given its key - in an almost constant time.

The remaining operations, as listed in Fig. 4.1 are going to be discussed in connection

¹As usual, $\mathcal{P}(M) = \{N \mid N \subseteq M\}$ denotes the powerset of any given set M , i.e. the set of all subsets of M .

with providing minimal conflicts and explanations and the handling of aggregation strategies and context spaces, respectively.

4.1.4 Heap and AggController

Figure 4.1 does not show any direct association connecting the classes **Heap** and **AggController** to any other class in the diagram. Still, they play their part in the implementation of RCS. So, what is the purpose of those two classes?

First of all, **Heap** provides an operation for creating a new wrapper for a constraint, that is, a new instances of **LeafNode**. The call for creation will be accompanied by a **String** containing the name of the new **LeafNode**. **Heap** maintains a sort of dictionary, from which **LeafNodes** can be retrieved via that given name. This simplifies the management of the content of an **Engine**'s pool of active constraints. Note that the corresponding operations, implemented at **Engine**, expect as parameter a **String**.

Likewise, there are other creators for obtaining new named instances of **OneOf**, **Projection** and **Cluster** which will be explained below. Therefore, **Heap** can be seen as the part that keeps track of all known objects that are important for all activities of a relational engine. Furthermore, it provides appropriate creation and retrieval services.

We will come across the class **AggController** again in the next chapter that deals with issues of a relational processor. There its central role is to maintain user settings and preferences for controlling the processing of relations, which also gave the class its name.

For a relational engine, it provides static methods for comparing **LeafNodes** and potential next aggregations, represented by pending instances of **AggNode**. This addresses the linearisation of one-the-fly aggregation strategies described in Subsect. 3.5.1. According to the ideas presented there, comparing two instances of **LeafNode** will basically work by taking a look at their constrainedness. This measure is provided by **getWeight** as implemented for any **Relation**; see Fig. 4.1.

Moreover, **AggController** implements some static operations for measuring the quality of existing aggregation trees in view of a possible reuse. In that sense, a tree may be judged "poor", e.g. due to its extreme unbalancedness: Subsection 3.4.2 suggests that an alteration of a tree in some leaf at the bottom end of a long path will leave only few and small reusable subtrees. Hence, a relational engine might, in this situation, prefer to build a new aggregation tree from scratch.

4.1.5 Representation of Aggregation Forests

Obviously we need to be able to represent aggregation forests, trees and all nodes of a tree. Chapter 3 makes clear that the set of all leaf nodes of an aggregation forest form the initial constraint problem, whereas all non-leaf nodes will only later be generated, in order to decide its consistency.

Therefore, leaf nodes and non-leaf nodes play different roles and shall be represented by two distinct classes. However, both hold a forward relation and also, both may be roots in an aggregation forest. To avoid redundancies in the code and to capture

common aspects, there exists thus the abstract superclass **Node**.

In addition to the forward relation, any instance of **AggNode** can host a backward relation. Aggregation trees are binary trees, and so each **AggNode** is the father of exactly two subtrees, rooted at two nodes which are again represented by the appropriate subclass of **Node**.

We have already mentioned above, that potential aggregation tasks will also be represented by instances of **AggNode**. Therefore, at runtime there will exist two sorts of **AggNode** instances: those for which **aggregate** had not yet been called and those for which it already had been. Only in the latter case will the forward relation be valid. We can distinguish between the two by asking an instance whether it is pending (former case) or not (latter case), using **isPending**. If an **AggNode** is still pending, it represents a potential, unperformed aggregation task. But then, the set of variables to be eliminated when imposing **aggregate**, is known beforehand and will be stored in the attribute **eliminableVariables**. At this point, **freeVariables** does not contain any useful information.

After having performed the aggregation, the situation is the other way round: **eliminableVariables** is no longer valid; instead, that aggregation may have produced free variables in the sense of Def. 2 which are going to be stored in **freeVariables**.

Obviously, we can keep track of a whole aggregation forest by holding links to all its root nodes. This is done by **AggForest**. Each instance of that class has a state, similar to **Engine** which facilitates a link to an **AggForest**. A forest's state will hence either be **unknown**, **inconsistent**, **consistent** or **solved**.

As soon as a root node is detected for which the forward relation is unsatisfiable, according to **isEmpty**, the state will be set to **inconsistent**. If this is not the case, and there exists still a potential aggregation task, the state is **unknown**. Otherwise, the forward phase has come to an end, and the analysed constraint problem will be considered consistent. Depending on whether the backward phase has already been executed, the state is going to be either **consistent** or **solved**. Note again, that both states here mean only that our prototype failed to prove inconsistency.

Solved can be seen as the substate of **consistent** in which approximations V_x for all value sets in the tightest bounds t_x have become known. Only then, the backward relations of **AggNodes** as well as the **solutions** attribute of **Engine** will contain valid information.

Having full information about all active constraints and their scopes, an **AggForest** can compute all reusable subtrees of any other **AggForest**. We shall see the pseudocode, based on the ideas developed in Subsect. 3.4.2, below.

Each aggregation will reduce the number of roots in an aggregation forest by one. For that, we must know which potential aggregation to perform. That piece of information may be provided by a clustered or even generic aggregation strategy, represented by **Cluster**; see below. If such information is missing, or in the case where that strategy is underspecified, a built-in on-the-fly strategy must be used to determine a good pending **AggNode**. This is what **getBestNextAggregation** returns. The linearised implementation of **getBestNextAggregation**, according to Subsect. 3.5.1, involves the already mentioned comparators implemented at **AggController**.

4.1.6 Representation of Context Spaces and Strategies

The classes `Cluster`, `OneOf`, `Projection` and `LeafNode` are responsible for representing context spaces as well as generic aggregation strategies.

According to Def. 17, in order to represent a context space, a relational engine needs to represent the fixed portion and a set of `oneOfs`. The former is just a collection of `LeafNodes`, one for each constraint of the fixed portion. Likewise, the `oneOfs` are represented by a collection of instances of `OneOf`. Both collections will be wrapped in an instance of `Cluster` that does not have a subcluster. Note that in Fig. 4.1, the possibility of a `Cluster` is included that has no `OneOf`; see multiplicity $0..∞$. This is however not allowed for a context space, and thus due to the second role that `Cluster` can play, when representing strategies. Note that, for representing context spaces, neither the class `Projection` nor the attribute `interfaceVariables` of `Cluster` is going to be needed.

The more subtle case is the representation of generic aggregation strategies. This time, we follow Def. 18, where we had already witnessed four different types of nodes, that will obviously be represented by instances of the classes `Cluster`, `OneOf`, `Projection` and `LeafNode`. The proof of Lem. 9 revealed that indeed, the root of any generic aggregation strategy must be a cluster node, provided that the context space is big enough. Furthermore, Def. 18 implies that `Projection` has a pointer to a single `LeafNode`, and a `OneOf` is linked to at least two. Also, any instance of `Cluster` that participates in the representation of a strategy, may be followed by some set of subclusters (again represented by `Cluster`), some `Projections`, `OneOfs` and `LeafNodes`. Definition 18, 5., imposes an additional constraint that is not displayed in Fig. 4.1 yet observed by our prototype implementation: The number of successor objects must be at least two.

The sets of variables $X(n)$ of Def. 18 are going to be stored in the attributes `interfaceVariables` of `Cluster` and `Projection`, respectively.

Note that the class `OneOf` allows for specifying a current context via `setCurrentLeafNode` which sets the attribute `currentLeafNode`, i.e. the current alternative, in the sense of Def. 17. In our prototype, the alternatives of a `OneOf` are maintained in an *array*. Therefore, specifying an alternative can be done by providing an integer index, as in `setCurrentLeafNode`.

4.2 The Forward Phase

Be, in what follows, C the set of active constraints in the pool maintained by some instance of `Engine`. In order to assume $|C| \geq 2$, as premised in Th. 1 of Section 3.3, the cases $|C| \in \{0, 1\}$ need to be dealt with: For $|C| = 0$ the `Engine` is confronted with the empty problem and can immediately move to the consistent state, cf. Fig. 4.2.

If $|C| = \{c\}$ has just a single element, the operation `isEmpty()`, applicable for any instance of `Relation`, is used to determine whether $c \sim \emptyset$ and thus whether or not C is consistent.

Clearly, utilising Th. 1 shall work by building a bunch of aggregation trees, one for each connected subset of C . This building process is called *forward phase*, since it

generates the forward relations as defined by Def. 12. As already mentioned above, each element $c \in C$ will be wrapped in an instances of **LeafNode**. Let us say that its **forwardRelation** is the relation c . Note that this extends the definition of forward relations, as given by Def. 12, to leaf nodes of aggregation trees. However, this is not going to cause trouble.

Figure 4.3 shows a pseudo-code variant of the operation **forward**, implemented at **Engine**, that assumes $|C| \geq 1$. Furthermore, **r.isEmpty()** must yield **false**, for any **Relation** **r** representing an initial constraint $c \in C$.

It starts with the trivial aggregation forest that has one single-node tree for each $c \in C$. The forest is then going to be manipulated until a termination criterion fires, forcing the engine's state to move to either **consistent** or **inconsistent**; cf. Fig. 4.2.

Let us take a brief look at the termination of **forward**. The only case in which the implementation at **AggForest** does not immediately terminate, is the one in which the method is called recursively; see line (5) in Fig. 4.3. However, in that situation, the corresponding **AggForest** has one less root, i.e. one less aggregation tree, due to the aggregation in line (3) and the subsequent pseudo-code that manipulates the given forest.

```

Engine:    procedure forward(Set of LeafNodes C)
  F ← new AggForest(set of roots: C)
  F.forward()
  state ← F.state
end

AggForest: procedure forward()
(1)  n ← getBestNextAggregation()
     if (previous line threw exception)
(2)    state ← consistent
     else
(3)    n.aggregate()
         remove successors of n from this forest
         add n as new root to this forest
(4)    if (n.forwardRelation.isEmpty())
         state ← inconsistent
     else
(5)    forward()
end

```

Figure 4.3: Pseudo-Code for the Forward Phase

So finally, if the execution has not yet terminated for other reasons, there will only be one root node left in the forest, in which case (1) throws an exception, forcing termination in line (2). This shows that termination is always guaranteed.

Line (1) captures the utilisation of an on-the-fly aggregation strategy, as introduced in Subsect. 3.5.1. There, we listed the three cases in which `getBestNextAggregation` is to throw an exception. In the pseudo-code given in Fig. 4.3, an exception can only be thrown in two of those cases: due to the number of roots being 1, or due to the fact that no two root nodes share a variable. The third scenario, in which some root relation is discovered to be unsatisfiable, can never occur. This is because initially each $c \in C$ is satisfiable, at least according to the predicate `isEmpty`; see above assumption. And whenever a new root is generated, it is immediately tested in line (4). This also makes clear, why (2) can always assign the consistent state.

In order to consider the complexity, we note that the worst case, in terms of cost, happens when we have a connected, consistent constraint problem C . Then, line (4) will trigger until (1) produces an exception (due to one remaining root node). Then, the forest has just one tree that has \square as its root relation, cf. Th. 1. Clearly, the algorithm in Fig. 4.3 will then perform the maximal number of aggregations, namely $|C| - 1$. Apart from that observation, one should consider the following facts.

- If the *low density assumption* holds, newly created root nodes will host a `forwardRelation` that relates, in its structural complexity, to the initial constraints in C ; see also again Ex. 3. Thus, the effort for the execution of line (3) does not depend on the number of roots in the current `AggForest`, and may be assumed to be $O(1)$.
- Testing a relation for emptiness; line (4); shall only deploy few simple syntactical checks as described above, in Subsect. 4.1.1. Therefore, this can also be assumed to consume a time of only $O(1)$.
- An exhaustive implementation of `getBestNextAggregation` is going to take time $O(|R|^2)$, where R is the set of root nodes. Hereby all pairs of root nodes need to be considered. However, the method may be linearised according to Subsect. 3.5.1, and as implemented in our prototype. Therefore, let us calculate here with an effort of $O(|R|)$.

Putting the bits and pieces together, we end up with an effort of $O(|C|^2)$. However, as our experimental results show, we sometimes get measurements that suggest a linear time consumption for checking consistency; see Sect. 6.2. On the one hand this is evidence for the validity of our *low density assumption*. On the other hand, choosing a best next aggregation, seems then to take only a negligible period of time, compared to that for actually performing the aggregations.

If we could ensure a constant bound for `getBestNextAggregation`, this would result - according to the above argumentation and still assuming a *low density* of our constraint problems - in a linear time complexity for `forward`. Clearly, this will be the case, when we provide a global aggregation strategy that is “not too underspecified”. This means that the execution of the strategy does not involve too many calls of a local aggregation strategy, as implemented by `getBestNextAggregation`; cf. Lem. 9, part 4.

4.3 The Backward Phase

Definition 12 presents a recursive definition of an aggregation node's backward relation. Assuming an aggregation tree for a connected constraint problem, for which the forward phase failed to prove inconsistency, this definition starts by assigning a relation at the root node, and continues in a top-down manner, stopping at the lowest non-leaf nodes in the tree. Obviously, it is rather intuitive to capture the procedure in terms of pseudo-code, see Fig. 4.4.

Theorem 2 proves that, given non-overestimating implementations of **join** and **project** at the class **Relation**, the presented algorithm will derive tightest restrictions for all variables. We have already discussed issues of soundness and completeness in a previous subsection, and the point has been made that our prototype computes, at the worst, overestimations. As a consequence, a computed value set V_x , for a variable x , is always going to subsume the correct set of values as captured by t_x ; cf. Th. 6, 3.

Note that the top method in Fig. 4.4 calls **backward** for each root node; see line (2). Possibly, those may also be instances of **LeafNode**, which happens whenever there exists a connected subset of the entire constraint problem, that consists of a single constraint. The respective pseudo-code for **backward** implemented at the class **LeafNode** is only for those instances actually going to do something, cf. the test in line (9). However, it will also be called during the top-down recursion for any other leaf node in the given aggregation forest.

Line (1) resets the map **solutions** that will be filled during the backward phase as a side effect of **backward**. In line (5), the set of variables X is computed, that have been eliminated during the given **AggNode**'s aggregation. Speaking in terms of Lem. 5, 1., that node coincides with $\epsilon(x)$, for all $x \in X$. Since there is a unique $\epsilon(x)$ for each variable x that has been eliminated during the forward phase, the map **solutions** will be filled monotonously.

The recursion is realised by means of the lines (7) and (8) which call the method for both successors of the **AggNode** at hand. Since the aggregation forest consists of finite trees, this proves termination of the given algorithm.

As already mentioned, there may exist isolated constraints c_0 that do not share a variable with any other constraint. For those, the only possibility to derive tightest restrictions for a variable $x \in vars(c_0)$ is to compute a representation of $\pi_{\{x\}}(c_0)$. This is accomplished by the bottom method of Fig. 4.4.

Let us consider the complexity of the backward phase, assuming that the initial constraint problem $C, |C| \geq 2$, has no isolated constraints and is moreover connected, and that it has been judged **consistent** by our prototype.

We have to focus on the lines (3), (4), (6) and (10), since those mention the basic relational operations **join** and **project**. For all other commands, the effort spent is dominated by that for those lines of pseudo-code.

First of all, it shall be clear that the **join** computed in line (3) has already been computed during the forward phase. Hence, the result can be stored in an extra attribute of **AggNode**, and we may replace line (3) by an appropriate accessing statement. Concerning line (4), we shall argue again, as in the forward phase, that the

effort does not depend on the instance of `AggNode` for which the method has been called. The reason is once more our *low density assumption*, ensuring that both `r12` and `r` represent relations with scope sizes bounded by some constant.

```

Engine:    procedure backward()
    full ← representation of □
    (1) solutions ← new Map(entries: none)
    F ← AggForest computed during forward phase
    for each root node n in F
    (2)      n.backward(full, solutions)
    F.state, this.state ← solved
end

AggNode:   procedure backward(Relation r, Map s)
    n1, n2 ← 1st, 2nd successor of this node
    r1, r2 ← (n1, n2).forwardRelation
    (3) r12 ← r1.join(r2)
    Z ← this.forwardRelation.variables
    if (this.isRoot)
        this.backwardRelation ← r12
    else
    (4)      this.backwardRelation ← r12.join(r.project(Z))
    (5) X ← (r1.variables ∪ r2.variables) \ Z
    for each x ∈ X
    (6)      Vx ← this.backwardRelation.project({x})
    s.addMapping(x ↦ Vx)
    (7) n1.backward(this.backwardRelation, s)
    (8) n2.backward(this.backwardRelation, s)
end

LeafNode:  procedure backward(Relation r, Map s)
    (9) if (this.isRoot)
        for each x ∈ this.forwardRelation.variables
    (10)      Vx ← this.forwardRelation.project({x})
    s.addMapping(x ↦ Vx)
end

```

Figure 4.4: Pseudo-Code for the Backward Phase

Although a thorough proof will not be given, there is another argument that supports the hypothesis of a $O(1)$ -cost for line (4): The propagated relation `r` captures global knowledge about the entire constraint problem and is hence likely to be highly constrained. Actually, we expect for most engineering problems arising from tasks of system analysis, that they have only few solutions: Fig. 3.3 is an example where, at the top of the tree, some variables have even been fixed to concrete values. Thus,

computing the join in line (4) is basically done by substituting known values into r_{12} , that are propagated from above inside r . Indeed, that join is a semi-join, since Z is subsumed in the variables of r_{12} .

Consequently, all backward relations are expected to represent small finite sets of tuples. Therefore, the projection in (6) becomes a simple operation. In addition to that, line (10) will never be approached since, under the above assumption concerning C , the test in (9) will always force the immediate termination of the bottom method of Fig. 4.4.

Clearly, when all critical commands have indeed a cost of $O(1)$, then the backward phase takes a time of $O(|C|)$. This is because each of the $|C| - 1$ inner nodes of the aggregation tree is visited exactly once. The above lax discussion is not to replace a detailed proof. Still, it raises the hope that the computations undertaken in each instance of **AggNode** take a time bounded by some $O(1)$. The experimental results in Sect. 6.2 support this hypothesis whenever we analyse problems that have only few solutions, as is typically the case in engineering tasks of system analysis.

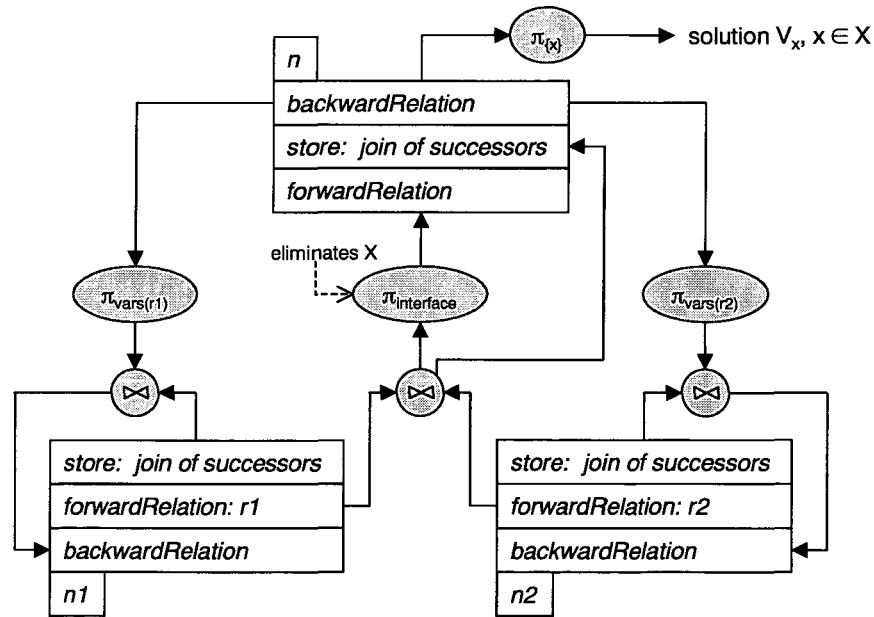


Figure 4.5: A Local View on the Forward and the Backward Phase

Before we close this section, let us summarise forward and backward phase. Figure 4.5 shows the computational information flow in the vicinity of an aggregation node n with successors $n1$ and $n2$. The grey shapes stand for the two basic relational operators. As can be seen, three calls of **join** and three calls of **project** suffice to determine all relevant relations. The top projection computes tightest restrictions for all $x \in X$ that have been eliminated by the centre projection, during the forward phase. The helper relation *store* contains the join of the respective successor relations and eases the computation of the backward relations for $n1$ and $n2$.

4.4 Minimal Conflicts and Explanations

4.4.1 Computing All Minimal Conflicts

Previous pseudo-code was formulated in terms of **procedures** that manipulate one or several attributes of certain instances of **Engine** and **AggForest**. In this section, a pseudo implementation for computing minimal conflicts and explanations will be provided, that returns representations of the desired objects. To distinguish that from pure manipulation of existing instances, the pseudo-code will speak of **functions** instead of **procedures**.

As in Chap. 3, we shall first focus on computing *all* minimal conflicts. Figure 4.6 shows the respective pseudo-code which is an immediate translation of Def. 14. Thanks to Th. 3, we know that the temporary variable **treeConflicts** is indeed filled with all minimal conflicts that consist of leaves of the aggregation tree rooted

at n . However, Th. 3 also allows for members of $con_{\uparrow}(\rho)$ that are proper supersets of minimal conflicts. Those are explicitly removed by an unrefined statement in line (2).

The **function** implemented at **Engine** assumes that the forward phase has already been run, and that it discovered the inconsistency of the underlying constraint problem.

We have seen above, as termination condition for the forward phase, that it suffices to produce *one* aggregation node for which the forward relation turns out unsatisfiable, according to **isEmpty**. So, our forward implementation ensures that there be *at most one* unsatisfiable forward relation. Hence, the iteration in line (1) will usually only be over one node.

However, we may alter the forward phase, in order to maximise the number of conflicts we can find: We continue to perform aggregations in other connected portions of the initial problem, even when we have already produced an unsatisfiable forward relation. Thus, we omit the termination condition that is due to inconsistency. Thereby, we may obtain additional unsatisfiable forward relations. Then the iteration condition in line (1) may hold for more than just one node.

The implementations at **AggNode** and **LeafNode** expect as argument a representation of the set $con_{\uparrow}(r)$, where r is the forward relation of the respective node. They return the corresponding set $con_{\uparrow}(r)$, cf. Def. 14. For the bottom method in Fig. 4.6 this should immediately be clear. The more complicated implementation at **AggNode** forms the sets T_1 , T_2 from the argument set T , makes two recursive calls that produce T_1' , T_2' and builds from those the set T' that is being returned. Again, the derivations of T_1 , T_2 and T' , respectively, are strictly according to Def. 14.

A more detailed look at the pseudo-code reveals that there are several tests of the form $(\bowtie M).isEmpty()$, where M is a set of relations. Obviously, this coincides with investigating whether M is a consistent constraint problem, and can be done by invoking an appropriate forward phase. This suggests that Fig. 4.6 presents rather expensive methods, in terms of time consumption.

Here, we shall not go in for analysing that complexity. It is clear that the number of minimal conflicts is strongly problem-dependent. And since the presented algo-


```

Engine:    function getAllConflicts()
            F ← AggForest computed during forward phase
            forestConflicts ← {}
(1)  for each root node n in F with n.forwardRelation.isEmpty()
            treeConflicts ← n.getAllConflicts({})
(2)      eliminate all proper supersets from treeConflicts
            forestConflicts ← forestConflicts ∪ treeConflicts
            return forestConflicts
end

AggNode:   function getAllConflicts(Set of Sets of Relations T)
            n1, n2 ← 1st, 2nd successor of this node
            r1, r2 ← (n1, n2).forwardRelation
            T1, T2 ← {}
            for each t ∈ T
                if (⊗(t ∪ {r1})).isEmpty() insert t as element in T1
                insert (t ∪ {r2}) as element in T1
                if (⊗(t ∪ {r2})).isEmpty() insert t as element in T2
                insert (t ∪ {r1}) as element in T2
            T1' ← n1.getAllConflicts(T1)
            T2' ← n1.getAllConflicts(T2)
            T' ← {}
            for each t1' ∈ T1'
                if (r2 ∉ t1') insert t1' as element in T'
            for each t2' ∈ T2'
                if (r1 ∉ t2') insert t2' as element in T'
            for each t1' ∈ T1' and each t2' ∈ T2'
                if (r1 ∈ t2' ∧ r2 ∈ t1')
                    t' ← ((t1' \ {r2}) ∪ (t2' \ {r1}))
                    if (⊗ t').isEmpty() insert t' as element in T'
            return T'
end

LeafNode:  function getAllConflicts(Set of Sets of Relations T)
            T' ← {}
            for each t ∈ T
                insert (t ∪ this.forwardRelation) as element in T'
            return T'
end

```

Figure 4.6: Pseudo-Code for Computing All Minimal Conflicts

rithm is guaranteed to find them all, besides a lot of useless proper supersets, its complexity will always dominate that number. The more practicable approach is to ask for just one minimal conflict, which is dealt with in the next subsection.

4.4.2 Computing One Minimal Conflict

Figure 4.7 presents a pseudo-code view of our actual prototype implementation of `RCS`, for finding one minimal conflict. Again, as in the previous subsection, that code is assumed to be run only when the `Engine`'s state is `inconsistent`.

Let us make plausible why line (1) will eventually return a minimal conflict, provided that `isEmpty()` always returns the correct answer. In that case, we shall verify, for any node `n` with forward relation `r`, the following invariants concerning the invocation of `n.getOneConflict(c)` where the argument `c` represents some relation `c`:

$$\bowtie \widehat{con}_\downarrow(r) = c, \text{ where we set } \bowtie M \stackrel{\text{def}}{=} \square, \text{ whenever } M = \emptyset. \quad (4.6)$$

$$\text{The returned set represents the set of relations } \widehat{con}_\uparrow(r) \cap \Lambda(r). \quad (4.7)$$

The proof of these two invariants makes use of Def. 15 and the results in Lem. 7. It can be found in the appendix.

Knowing that (4.7) holds, we conclude that line (1) in Fig. 4.7 returns $\widehat{con}_\uparrow(\rho) \cap \Lambda(\rho)$, where ρ be the forward relation of the root node. But then, Th. 4 together with the fact that $\widehat{con}_\uparrow(\rho)$ consists only of leaves of $\Delta(\rho)$, shows that line (1) returns indeed a minimal conflict. Let us once again note that, for this to work, we need an implementation of `isEmpty` that always answers correctly.

The definition of the logic predicate $E_\downarrow(\cdot)$ in Lem. 7 shows that $\bowtie(\widehat{con}_\downarrow(r) \cup \{r\})$ is unsatisfiable. This provides us with a good intention for the above relation `c`: According to (4.6), $r \bowtie c \sim \emptyset$, and `c` can hence be seen as a *conflict relation* for the given node.

Before we discuss the complexity of the implementation in Fig. 4.7, some optimisations of the code can be made.

First, it should be clear that only the portion of `c` is needed that joins with the forward relations of both `n1` and `n2`.² Thus, we may insert an additional first line of code in the implementation at `AggNode` that replaces `c` by `c.project(X12)`, where `X12` represents the unioned set of variables of those two forward relations.

Another effective optimisation removes the explicit computation of $\bowtie S1$ in line (5): The recursive call in line (4) will only by and by collect the members of the set `S1` by means of its own recursions. We could hence alter `getOneConflict(c)` so that it return not only that set `S1` but additionally also its combined join $\bowtie S1$. As sole consequence, we need to replace line (6) by

`return (S1 \cup S2, s1 \bowtie s2),`

where `s1` and `s2` denote the combined joins $\bowtie S1$ and $\bowtie S2$, computed during the recursion in line (4) and (5).

²The proofs for the invariants (4.6) and (4.7) given in the appendix are based on the unmodified code as presented in Fig. 4.7 but can easily be adjusted to the suggested modifications.

Concerning complexity in terms of time consumption, the worst case is the one in which lines (2) and (3) are never approached but the conflict always spreads over both subtrees. This happens indeed, e.g., when there is only one minimal conflict that involves all leaves of the subtree rooted at the root node. In other words, no constraint can be suspended. Moreover we may utilise a *low density assumption*: With the above optimisation based on the variable elimination $c.project(X_{12})$, the time spent for any one join can again be assumed to take constant time.

```

Engine:    function getOneConflict()
            full  $\leftarrow$  representation of  $\square$ 
            F  $\leftarrow$  AggForest computed during forward phase
            n  $\leftarrow$  root node of F with n.forwardRelation.isEmpty()
(1)        return n.getOneConflict(full)
end

AggNode:   function getOneConflict(Relation c)
            n1, n2  $\leftarrow$  1st, 2nd successor of this node
            if n1.forwardRelation.join(c).isEmpty()
(2)         return n1.getOneConflict(c)
            if n2.forwardRelation.join(c).isEmpty()
(3)         return n2.getOneConflict(c)
(4)         S1  $\leftarrow$  n1.getOneConflict(n2.forwardRelation.join(c))
(5)         S2  $\leftarrow$  n2.getOneConflict( $\bowtie$  S1).join(c))
(6)         return S1  $\cup$  S2
end

LeafNode:  function getOneConflict(Relation c)
            return {this.forwardRelation}
end

```

Figure 4.7: Pseudo-Code for Computing One Minimal Conflict

Furthermore, the number of joins can easily be counted: There are two joins for the preconditions of lines (2) and (3). With the above optimisation concerning the computation of \bowtie S1, we end up with one further join in line (5). Note that the one in line (4) has been computed earlier. Finally, the altered version of line (6) - see above - computes one join.

Assuming a connected inconsistent constraint problem C there are thus, in the worst case, four joins per aggregation node. This gives a total number of joins to be performed of $4 \cdot (|C| - 1)$, since there are exactly $|C| - 1$ inner nodes in the aggregation tree. The resulting linear worst case time consumption of $O(|C|)$ has also been found in [59].

The worst case assumes that any invocation of the middle method in Fig. 4.7 will approach line (6). Although this will not always be the case, it certainly is after the initial call in line (1), provided that the root node n is not a leaf node. This is because

the preconditions of the lines (2) and (3) simplify, in that case, to testing whether `n1.forwardRelation.isEmpty()` and `n2.forwardRelation.isEmpty()`. Neither should be the case, for otherwise the forward phase should have terminated before creating a father node for `n1` and `n2`.

4.4.3 Providing Minimal Explanations

Chapter 3 traces the problem of computing minimal explanations back to providing minimal conflicts for a slightly altered constraint problem; see Th. 5. The main task of this subsection will hence be to provide pseudo-code for the construction of an aggregation tree for the altered problem. The respective procedure has already been described in Subsect. 3.3.5, right after the proof of Th. 5. The pseudo-code can immediately be written down, as presented in Fig. 4.8.

Obviously, the method name in line (5) is intended to be in accordance with the name of the hosting method of Fig. 4.8. Thereby, we can provide all or just one minimal explanation, respectively, for the tightest restriction that has been found for the variable x during the backward phase.

```

Engine:      function get[One/All]Explanation[s] (Variable x)
(1)   Vx ← solutions.entryFor(x)
(2)   sx ← Vx.complement()
(3)   if sx.isEmpty() throw exception "nothing to explain"
      F ← AggForest computed during forward phase
      n ← node in F representing  $\epsilon(x)$ ; see Lem. 5, 1.
(4)   n' ← n.alterTree(x)
(5)   return n'.get[One/All]Conflict[s](sx)
end

AggNode:     procedure alterTree(Variable x)
      n1, n2 ← 1st, 2nd successor of this node
      r1, r2 ← (n1, n2).forwardRelation
(6)   X ← this.forwardRelation.variables ∪ {x}
(7)   this.forwardRelation ← r1.join(r2).project(X)
(8)   if this.isRoot() return this
(9)   else this.father.alterTree(x)
end

```

Figure 4.8: Pseudo-Code for Providing One or All Minimal Explanations

The top method in Fig. 4.8 expects thus the Engine's state to be **solved**, that is, both the forward and backward phase have already been performed. In particular, this implies that the problem at hand could not be proved to be inconsistent. And the attribute **solutions** of the given instance of Engine is indeed filled. Consequently, the accessor in line (1) returns the representation V_x of some approximation V_x of the tightest restriction t_x for x .

sx, as computed in line (2) is supposed to represent s_x as defined in (3.16). So, in order to make this work, we need an operation **complement** implemented for any **Relation** that has the special structure of **sx**. It is noteworthy that we need not implement that operation for all instances of **Relation**.

Line (3) performs the test whether **sx** is found to be unsatisfiable which is equivalent to $\forall x$ representing a relation that allows for any sensible assignment to x . There is nothing to explain whenever the test yields **true**. Only otherwise, the algorithm continues.

In order to verify that line (5) returns indeed all or one minimal explanation, respectively, we need to take a closer look at line (4) and thus at the alteration procedure at the bottom of Fig. 4.8.

As has already been described in Subsect. 3.3.5, **alterTree** starts at the aggregation node representing the unique node $\epsilon(x)$ that eliminates x . Line (6) computes the new scope that differs from the old one in that it now contains the variable x . Consequently, line (7) re-performs the aggregation. This process is repeated in a bottom-up manner - see line (9) - until the root node of the aggregation forest at hand is reached; line (8). Note that this is the exact translation of the procedure given in Subsect. 3.3.5. The resulting node **n'** in line (4) will be a representation of the altered root, there denoted by $\rho' = n'_1$.

However, here is a little modification. The aggregation tree C_x constructed there was rooted at some node $\rho^* \stackrel{\text{def}}{=} \pi_{\emptyset}(\rho' \bowtie s_x)$ with successors ρ' and s_x . We can omit that construction here, because invocation of **get[One/All]Conflict[s](full)** at ρ^* would immediately result in the call **get[One/All]Conflict[s](sx)** at ρ' ; see line (4) of Fig. 4.7. Moreover, line (5) in that figure would assign to **S2** the singleton set $\{\mathbf{sx}\}$ which would have to be excluded from any minimal conflict in order to provide a minimal explanation, cf. Th. 5.

Therefore, line (5) of Fig. 4.8 is the appropriate measure to derive one or all minimal explanations, respectively. Furthermore, none of the resulting explanations is going to contain **sx**, as desired.

Clearly, the bottom method **alterTree** will modify all nodes on the path from the root down to the initial node **n**. This needs to be taken care of by any subsequent method that assumes a valid aggregation forest.

As to the time complexity of the given pseudo-code, it is straightforward that the cost of the top method is the same as that of the respective conflict-computing method. In order to see that, we argue that the cost for line (4) is dominated by that of line (5):

The operations in line (7) can be compared to one aggregation of the forward phase. The fact that we increase the goal interface by one variable does not matter. So, **alterTree** consists basically of k aggregations, where k is in the worst case some $O(|C|)$, where C denotes the initial constraint problem.³ In other words, line (4) has a worst case cost that relates to that of the forward phase, i.e. $O(|C|)$.

We have seen that deriving one minimal conflict takes a linear number of joins, i.e. also at least a linear cost. Therefore, the complexity of line (5) dominates that

³This worst case materialises for an extremely unbalanced tree. Otherwise, we should rather expect $k = O(\log(|C|))$.

of line (4), as hypothesised above.

4.5 Analysis of Problem Sequences

4.5.1 Context Management

Section 3.4 is dedicated to solving sequences of numerous constraint problems. The main characteristic of problem sequences for the engineering tasks we are addressing, is the similarity of neighbouring problem instances, which we also called contexts. Typically, and as carried out in Subsect. 3.4.1, those multiple problems can be captured by means of context spaces; see Def. 17.

Even when the distinct contexts are rather different in the sense that they do not share too many constraints, our prototype implementation of RCS is equipped with operations that support a quick switch-over from one context to the next. Those are the operations illustrated in Fig. 4.2 for altering the pool of active constraints. Basically, in order to analyse the family of constraint problems $\{C_i\}_{i \in I}$, any $c \in \bigcup_{i \in I} C_i$ needs to be made known to the class **Heap** that is described in a previous subsection. Afterwards, RCS can be prepared for analysing the constraint problem C_k by activating all $c \in C_k$.

As has been pointed out in Sect. 3.4, a switch-over from C_a to C_b comprises the potential for reusing certain computations. Those basically concern all derivations based on the constraints in $C_a \cap C_b$. The discussion in Subsect. 3.4.2 leads to the algorithm presented in the next subsection, which collects all reusable subtrees of an existing aggregation forest.

Although the alteration of the pool of active constraints can always be accomplished by means of **addRelation** and **removeRelation**, we expect our problem sequences to fit nicely into the structure of a context space as defined by Def. 17. Therefore, facilities for representing context spaces as well as special methods for a switch-over from a context C_a to the next, C_b , have been implemented in our prototypic implementation of RCS.

The appropriate measure for a context switch is the **Engine**'s method **setContext**; see Fig. 4.1. That operation expects as argument an array of **integers** with as many entries as there are instances of **OneOf** in the context space maintained by the **Engine**. If the i^{th} entry is j , then a call of **setContext** will set the attribute **currentLeafNode** of the i^{th} **OneOf** to point to its j^{th} alternative.

Of course, on a lower level this will again be implemented via appropriate invocations of **addRelation** and **removeRelation**. Also in this scenario, where the **Engine** maintains an instance of **Cluster** that represents a context space, an effective analysis of the entire space can and must be supported by reusing certain subtrees of an existing aggregation forest.

4.5.2 Reuse

The situation RCS is facing is the following. A context C_a has been analysed, resulting in some aggregation forest \mathcal{F}_a . The solving instance of **Engine** holds a link to that forest. Then, in order to analyse the next context C_b , each $c \in C_a \setminus C_b$ has

to be removed from and each $c \in C_b \setminus C_a$ needs to be added to the pool of active constraints.

Clearly, we need to identify all reusable subtrees of \mathcal{F}_a , given the goal context C_b which is already activated. This is done by the pseudo-code presented in Fig. 4.9. That code alters the instance of **AggForest**, which represents \mathcal{F}_a before the invocation, and a valid aggregation forest \mathcal{F}_b for C_b afterwards. The alteration itself takes place in line (6), where the set of the forest's trees is modified.

```

AggForest: procedure reusableSubtrees(Set of LeafNodes newL)
    oldL  $\leftarrow$  set of LeafNodes of this forest
    (1) for each  $n \in \text{oldL} \setminus \text{newL}$ 
        n.markUpwardPath()
    (2)  $X \leftarrow \bigcup \{n.\text{forwardRelation.variables} \mid n \in \text{newL} \setminus \text{oldL}\}$ 
    (3) for each  $x \in X$ 
        n  $\leftarrow$  node in this forest representing  $\epsilon(x)$ 
        n.markUpwardPath()
    (4) newRoots  $\leftarrow \text{newL} \setminus \text{oldL}$ 
    (5) for each root node n in this forest
        n.collectUnmarkedSubtrees(newRoots)
    (6) (set of root nodes of this forest)  $\leftarrow \text{newRoots}$ 
    end

Node: procedure markUpwardPath()
    mark this node
    if (this.isRoot) return
    else this.father.markUpwardPath()
    end

Node: procedure collectUnmarkedSubtrees(Set of Nodes N)
    (7) if (this node is unmarked)  $N \leftarrow N \cup \{\text{this}\}$ 
    (8) else if (this is not a leaf)
        n1, n2  $\leftarrow$  1st, 2nd successor of this node
        n1.collectUnmarkedSubtrees(N)
        n2.collectUnmarkedSubtrees(N)
    end

```

Figure 4.9: Pseudo-Code for Collecting Reusable Subtrees

The presented pseudo-code is, once again, a direct translation of a procedure that has already been given in natural language in Subsect. 3.4.2. The iteration in line (1) captures step 1 of the algorithm stated there. Hence, upward marking affects all those nodes that are later to be abandoned. Step 2 is realised by means of line (2) and the iteration in line (3). That is, for any variable x eliminated in the existing aggregation forest, which is re-introduced by some new constraint, the upward path from $\epsilon(x)$ must be discarded.

The collection of newly introduced constraints as well as of maximal reusable subtrees is done by line (4) and the iteration in line (5), respectively, which performs a call for each aggregation tree in the forest.

The procedure for upward marking is straightforward. Note that a node may be marked several times. Any node that has been marked at least once is going to be discarded.

The bottom procedure `collectUnmarkedSubtrees` traverses an entire aggregation tree. Whenever a node n is approached which is unmarked, the implementation of `markUpwardPath` guarantees that there is no marked node in the subtree $\Delta(n)$ rooted at n . Thus, line (7) is the only case, in which the traversal of the tree is stopped. Otherwise we descend - see line (8) - into both subtrees of n , continuing the traversal.

Lem. 8 states that this algorithm provides us indeed with a valid aggregation forest for the new context, provided by the argument `newL`.

Note that we may call `forward` as depicted in Fig. 4.3, right after `reusableSubtrees`, in order to execute the forward phase and to decide whether the new context is consistent. Also in general, none of the backward relations attached to nodes in the altered forest will be valid any longer. We need thus to re-run the backward phase; of course only in the case that the new context turned out consistent.

How many root nodes are there going to be in the altered aggregation forest. Obviously,

- For each newly introduced constraint, there is a new root.
- Suppose, we discard a single leaf node λ in some tree Δ and consequently the entire upward path. Then we obtain $l - 1$ reusable subtrees, where l is the length of the path $\rho(\Delta) \xrightarrow{*} \lambda$, i.e. the number of nodes on that path.
- If we discard some eliminating node $\epsilon(x)$ and its upward path, we get $l + 1$ reusable subtrees with l again being the path length.

In the worst case, when Δ is extremely unbalanced, l relates to the number of leaves, i.e. at worst $l = O(|C|)$, with C denoting the old context. For a balanced tree, we expect however $l = O(\log(|C|))$. It is easy to verify that those results will also hold for context switches that discard more than one but still a small, bounded number of constraints.

Therefore, assuming a context switch with few changes, we can expect a logarithmic number of trees in the altered tree. At the worst, it is going to be linear. In our implementation of RCS, any `AggForest` can provide a statistic that lists minimal, maximal and average lengths of all root-leaf paths. This has already been mentioned in Subsect. 4.1.4, where `AggController` was pointed out as the class that implements the respective static measuring methods. Equipped with information of that sort, we may prevent the invocation of `reusableSubtrees` for any instance of `AggForest` with a "poor" statistic.

Let us finally take a brief look at the time consumption of the pseudo-code given in Fig. 4.9. For that, we shall consider only a very simple case. Hereby, the context

switch changes the `currentLeafNode` of only one `OneOf`. Furthermore, all alternatives of that `OneOf` are assumed to mention the same variable(s).⁴

Clearly, the loop in line (1) marks a certain root-leaf path in some tree of the given forest. But any node `n` in the loop at line (3) is going to lie on the same path. Therefore, that loop's marking activity will not mark any previously unmarked node. The collection part in line (5) will take a time of $O(l)$, where l denotes, as above, the path length of the marked path.

Summarising, in this simple scenario, we end up with a worst case time consumption of $O(|C|)$, where C denotes the old or the new context. Again, for a well-balanced tree, we can expect this effort to drop to $O(\log(|C|))$.

This result will similarly hold when the performed context switch modifies not just one `currentLeafNode` but a certain number which is bounded by some constant. Additionally, not all alternatives of the respective `OneOfs` need to have coinciding sets of variables, as in the above simple scenario. We just need, again, a constant bound for the number of newly introduced variables when moving from one alternative to the next.

It should be clear that, for most practical applications, the analysis of the respective context spaces can be organised in such a way that those constant bounds may indeed be provided.

4.6 Utilisation of Aggregation Strategies

We shall complete this chapter by presenting the pseudo-code for the utilisation of a generic aggregation strategy. Once again, the code is a straightforward consequence of previous formalisations as carried out in Chap. 3.

The assumption here is that we are provided with an instance of `Cluster` that represents a generic aggregation strategy as defined by Def. 18. Aspects of this representation have already been discussed in Subsect. 4.1.6. Lemma 9 gives a clue as to how to interpret and process the given `Cluster`. The local aggregation strategy σ is provided by an appropriate implementation of `getBestNextAggregation` at the class `AggForest`. It is going to be needed only in connection with item 4. in Lem. 9.

Figure 4.10 shows, at the top, the operation `getAggForest` that returns a new instance of `AggForest` which is the representation of $\Theta(n)$; see Lem. 9. That method first collects all immediate `LeafNode` successors of the `Cluster`; see line (1). This corresponds to item 1. of Lem. 9.

The iteration in (2) reflects item 2. of the same lemma. Likewise, the loop in line (3) accomplishes the interpretation of all hosted `Projections`. Note here, that the bottom method takes also care of the correct computation of X as stated in item 3. of Lem. 9.

The interesting iteration in line (4) involves recursive calls for all subclusters of the current strategy cluster. In line (5), all partial aggregation trees are known and collected in the set of root nodes `N` which is used to form a new instance of

⁴Note that this is typically the case for a `OneOf` that captures all positions of a switch or all possible assignments for some behavioural mode variable in diagnosis.

AggForest. Finally, line (6) realises the forward phase for that forest according to the implementation given in Fig. 4.3 with the following minor changes:

Note that we make here, in Fig. 4.10, a parametrised call to the method **forward** which was there stated without any argument. Still, that mismatch can easily be recovered. We can modify the bottom procedure in Fig. 4.3 to make it expect a set of *protected variables* that must not be eliminated during the entire forward process. The only other modification we need to make concerns line (1) in Fig. 4.3: The local strategy implemented via **getBestNextAggregation** must also be provided with that set of variables. As a consequence, the proposed aggregation node **n** will contain no protected variable in its set of **eliminableVariables**; cf. Fig. 4.1. Then, the aggregation in line (3) of Fig. 4.3 will leave all protected variables uneliminated.

```

Cluster:    function getAggForest()
(1)  N ← LeafNodes hosted in this Cluster
(2)  for each OneOf o hosted in this Cluster
      N ← N ∪ {o.currentLeafNode}
(3)  for each Projection p hosted in this Cluster
      N ← N ∪ {p.getLeafNode()}
(4)  for each (Sub-)Cluster c hosted in this Cluster
      F ← c.getAggForest()
      N ← N ∪ (set of root nodes of F)
(5)  newForest ← new AggForest(roots: N)
(6)  newForest.forward(this.interfaceVariables)
      return newForest
end

Projection: function getLeafNode()
  n ← LeafNode hosted in this Projecion
  r ← n.forwardRelation
  X ← this.interfaceVariables ∩ r.variables
  return new LeafNode(forwardRelation: r.project(X))
end

```

Figure 4.10: Pseudo-Code for Processing a Generic Aggregation Strategy

Chapter 5

Architecture and Algorithms of a Relational Processor

The previous chapter presents the high-level algorithms of our relational constraint solver, as suggested by definitions, lemmas and theorems proved in Chap. 3. All of them assume that there be a special module that allows for the representation of constraints, their efficient pairwise combination according to a join operator and their manipulation by a project operator. This chapter shall explain how such a module can be designed and implemented. Constraints will observe a disjunctive normal form in which each disjunct is in a so-called partially solved form. That form is inspired by existing canonical forms for the representation of conjunctive constraints. Also, issues of arithmetic terms and of values will be discussed.

5.1 Overview

In the previous Chaps. 3 and 4, we devised a separation between the formal foundations of our algorithms and practical aspects of their prototypic realisation. This was due to the fact that the main focus of this dissertation is indeed on the higher level that deals with a relational engine.

Still, aspects of the lower level concerning a relational processor are also not to be underestimated. And since all high level algorithms rest upon the lower level implementations of our fundamental relational operators, great care needs to be taken. However, we shall here not follow the above separation of formal and practical issues. Instead, formal investigations will be directly included in the general train of thought of this chapter, whenever necessary.

We will start with an overview, presenting a class model in UML. Afterwards, the main services are recapitulated, that must be provided by a relational processor. The presented class architecture reflects our intention to represent constraints in disjunctive normal form. With respect to this canonical form, so-called atomic relations constitute the terminal elements. Those always relate two terms. Consequently, our relational processor facilitates further modules for the representation and manipulation of terms and, as their ground representatives, of values.

In order to implement the desired relational operator *project*, conjunctions of atomic constraints are maintained in a so-called partially solved form which is going to be

a bunch of static methods for creating instances of values, terms and constraints. Those creation operations will perform certain manipulations with the above goal, and e.g. return the singleton instance of **Empty** when asked to instantiate the constraint $x \neq x$. As in the example, most of the undertaken reformulations will be triggered by simple syntactical patterns.

Apart from **Empty** and **Full**, Fig. 5.1 depicts the class **AbstractOr**, instances of which represent disjunctive constraints. Each disjunct will be represented by an instance of **AbstractAnd**. An **AbstractOr** with at least two disjuncts - thus a *true* disjunction - concretises to **Or**, whereas any single-disjunct disjunction collapses to an instances of **AbstractAnd**.

Similarly, an **AbstractAnd** is to represent a conjunction of atomic constraints. As long as there are at least two conjuncts in a conjunction, it is going to be represented by the class **And**. Again, single-conjunct conjunctions are just single atomic constraints that are captured by **Atom**.

In our implementation, an **Atom** relates exactly two instance of **Term**, where the sort of relation, i.e. equality ($=$), inequality ($\leq, <, \geq, >$) etc., is represented by the class **RelationType**. There are only as many instances of that class as there are sorts of binary relations we intend to represent. For example, two atomic relations capturing equations will maintain links to a coinciding instance of **RelationType**, namely the one that stands for equality ($=$).

As for terms, Fig. 5.1 indicates, by the light grey box, that there are actually numerous classes for representing terms. All of them are derived from **Term**, and we shall have a look at terms later on. Two ground representatives of terms are variables and values. Also for values, there is a whole value module - in Fig. 5.1 depicted by the dark grey box - with several classes, that is discussed below.

Any relation either directly maintains its set of variables in an appropriate attribute, or can provide that set on demand, by means of a retrieval method. Note that Fig. 5.1 also includes the possibility of relations without variables (cf. multiplicity $0..∞$), as is the case of **Empty** and **Full**, and only for those.

5.1.1 Services for a Relational Engine

Going back to Fig. 4.1, Fig. 5.1 consequently lists the methods

- **join:** for computing the representation of two relations' join,
- **project:** for obtaining projections of a given relation onto subsets of its variables,
- **isEmpty:** for determining whether a relation may be regarded unsatisfiable,
- **getWeight:** for providing a measure for "how constraining" a given relation is. A smaller result means a higher constrainedness.

Definition 4 shows that the join of two relations is reflected in the logical *and* of two conditions: We say that the join of r_1 and r_2 contains all tuples, appropriate projections of which are tuples of r_1 *and* of r_2 , respectively. Therefore, **join** is not only important in connection with the high-level algorithms developed in Chap. 4,

but is the natural means to produce a representation of the conjunction of two existing relations.

Likewise, we are going to need an operation for computing their disjunction. This is the purpose of `or`, as listed in Fig. 5.1, see class `Relation`. So, by mentioning `join` and `or`, we intend to address also the aspect of constructing arbitrary *and/or-junctions* of existing relations. Clearly, this covers tasks of instance creation and should hence also be considered a service for a relational engine.

Let us, for completeness sake, state a formal definition for the disjunction of two arbitrary constraints.

Definition 20 (Disjunction)

Let $c = (X, \mathcal{A})$ and $d = (Y, \mathcal{B})$ be two non-trivial constraints. By overloading the symbol “ \vee ” for logical or, we define the **disjunction** of c and d according to

$$c \vee d \stackrel{\text{def}}{=} \left(X \cup Y, \left\{ \alpha : X \cup Y \rightarrow \bigcup_{z \in X \cup Y} \text{dom}(z) : \alpha|_X \in \mathcal{A} \vee \alpha|_Y \in \mathcal{B} \right\} \right).$$

Set furthermore for any constraint c ,

$$\begin{aligned} c \vee \emptyset &= \emptyset \vee c \stackrel{\text{def}}{=} c \\ c \vee \square &= \square \vee c \stackrel{\text{def}}{=} \square. \end{aligned}$$

Note the similarity to Def. 4 for non-trivial constraints; we just replace logical *and* by *or* in the right-hand side set of assignments. The disjunction operator is - just like \bowtie - commutative and associative. A proof is omitted but can be obtained by some simple modifications in the proof of Lem. 2; see appendix.

Finally, `complement`, implemented at `Atom`, has already been mentioned in Chap. 4. The provision of minimal explanations, as proposed by the code of Fig. 4.8, requires that method. Already there, we pointed out that `complement` needs to be implemented only for a certain category of relations. More concretely, of interest are those relations that represent tightest bounds for single variables. In our prototype those are always going to be instances of `Atom` which relate an instance of `Variable` and one of an appropriate subclass of `Value`; see the description of the value module in a later section.

Summarising, Fig. 5.1 mentions operations that are essential for constructing more complex constraints and for the implementation of algorithms of a relational engine. However, more methods will be needed in the context of actually realising those important operations. For the sake of clarity, these have been omitted in Fig. 5.1 but will be introduced in subsequent paragraphs.

5.2 How to Implement the Join Operator

5.2.1 Relations in Disjunctive Normal Form

As already mentioned, in our Java implementation of RCS, any constraint is represented in the so-called *disjunctive normal form (DNF)*. This means that a constraint is, at top level, separated into a number of disjuncts. Those, in turn, are conjunctions of atomic constraints. Fig. 5.2 illustrates this structure for the example of the bulb constraint that was part of a simple component library, in Fig. 2.3, for the construction of electric circuits. The constraint decomposes into three distinct alternatives, depending on whether the mode M equals *ok* or *broken*, and - in the former case - the bulb is lit or not. Moreover, we see that atomic constraints common to two or all three disjuncts are represented by the same instance of **Atom** which turns the naive notion of a tree into the DAG depicted in Fig. 5.2.

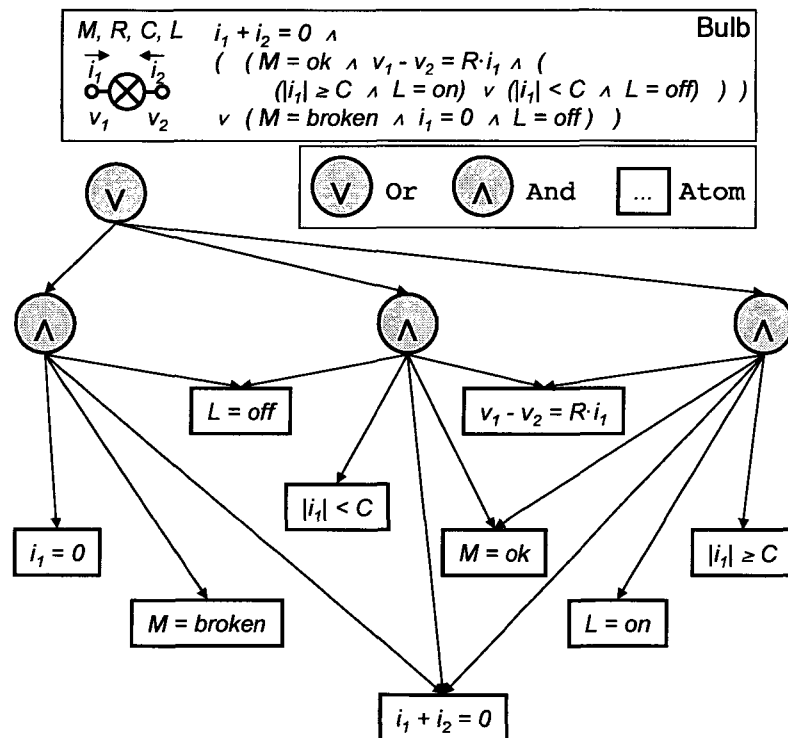


Figure 5.2: The Bulb Constraint of Fig. 2.3 in Disjunctive Normal Form

What happens, when there is only one disjunct in a disjunction? Referring to Fig. 5.1, this is clearly the case, when **AbstractOr** is specialised to the subclass **AbstractAnd** and not to **Or**. Likewise, **AbstractAnd** may be specialised to **Atom** instead of **And**. Consequently by transitivity, **Atom** is a specialisation of **AbstractOr**. In that sense, we may say that instances of **AbstractAnd** and of **Atom**, although no disjunctions, are also in DNF.

It is well-known that any and-/or-junction of logical formulas can be translated into

DNF. Hence, our choice is not a limitation. Moreover, there are some reasons for choosing the DNF instead of other common canonical forms.

First of all, when modelling a typical engineering component, there are almost always a certain set of possible customisations. In addition, a component typically has - for the purpose of failure detection and the analysis of failure effects - different behavioural modes, as the bulb in our electric library. Moreover, many components, as e.g. a switch, facilitate different behaviours according to different control actions. All these aspects suggest to view a component as a set of alternative actual embodiments which gives rise to modelling components mathematically as disjunctions.

Another argument is provided by the way we intend to decide consistency: Suppose that the entire constraint problem of interest is represented in a single constraint in DNF. Then, in order to decide whether there is a solution, we must decide whether at least one of the disjuncts has a solution. This, in turn, is done by building an aggregation forest from all atomic constraints in the respective conjunction. Obviously, the fact that a relation in DNF explicitly contains conjunctions, makes the forward phase more immediately applicable. Note that in this respect, the conjunctive normal form would be less advantageous.

Apart from the consequent representation of our relations in DNF, we experimented with a *factored-out DNF*. This is a DNF, where atomic constraints common to all disjuncts are separated from the rest of the respective and-/or-junction. In our example in Fig. 5.2 the equation of currents, $i_1 + i_2 = 0$, would be identified as belonging to all three disjuncts and factored out. However, the advantage of a more economical representation by means of the factored-out DNF, was in our experiments outweighed by the additional cost for identifying common atomic constraints and the maintenance of that normal form, especially after applications of the join and project operators.

On the contrary, facilitating the DNF comes at an acceptable cost. We shall see below that the operations join and or for combining two relations in DNF can be implemented relatively straightforwardly.

Before that, we remark that in previous versions of our prototype, there had been an additional class named **Table** for representing a special sort of relations. Those resemble typical tables of a database: An instance captures a matrix of values with one column for each variable of the relation and finitely many rows. Each table cell contains a value of the corresponding variable's domain. Then, a row is interpreted as the assignment that maps each variable to the respective value entry. Furthermore, the entire table captures a set of alternative assignments, and is thus an extensional representation of a non-trivial constraint in the sense of Def. 2.

Note that any instance of **Table** is ad hoc in DNF. The removal of the class from our design took place in a certain stage of consolidation. At that time, the DNF was introduced and any table could suddenly also be represented by a disjunction of conjunctions of atoms, where each atom is just an equation relating a variable and a value. During this phase of consolidation, our intention was mainly to remove redundant facilities for representing relations.

Today, the prototype is - at the level of relations - relatively stable in its class architecture, and we could re-introduce **Table**. Even though this would clearly also

re-introduce redundancy, we could now better exploit the advantageous of a more immediate representation of finite sets of assignments. For instance, by using **Tables**, we would ease the handling of constraint problems with a considerable amount of finite domain constraints as common in constraint satisfaction problems.

5.2.2 Join and Or

Let us now take a look at the implementation of **join** and **or** for the combination of two **Relations** is DNF. Our implementation has to deal with all possible pairs

```

Empty:      function join(Relation r)
             return (sole instance of Empty)

Full:       function join(Relation r)
             return r

Or:         function join(Or/And/Atom r)
             result ← sole instance of Empty
             for each disjunct d of this Or
(1)         result ← result.or(r.join(d))
             return result

AbstractAnd: function join(And/Atom r)
(2a) A ← (set of Atoms of this AbstractAnd)
(2b) A ← A ∪ (set of Atoms of r)
(3)  conjuncts ← minimalConjuncts(A)
      if |conjuncts|=1 return (the sole conjunct)
      else return Factory.makeAnd(Atoms: conjuncts)

Empty:      function or(Relation r)
             return r

Full:       function or(Relation r)
             return (sole instance of Full)

AbstractOr:  function or(Or/And/Atom r)
(4a) A ← (set of AbstractAnds of this AbstractOr)
(4b) A ← A ∪ (set of AbstractAnds of r)
(5)  disjuncts ← minimalDisjuncts(A)
      if |disjuncts|=1 return (the sole disjunct)
      else return Factory.makeOr(AbstractAnds: disjuncts)

```

Figure 5.3: Combination of Two Relations via Join and Or

of terminal subclasses of **Relation**. Since the operations are commutative, we will need just one implementation for both a pair of subclasses (S_1, S_2) and its sole

permutation (S_2, S_1) . In our implementation, we introduced the following total ordering among all subclasses of `Relation`:

`Empty < Full < AbstractOr < Or < AbstractAnd < And < Atom.`

We made furthermore sure that, e.g., an invocation of `myAtom.join(myOr)` with `myAtom` and `myOr` being instances of `Atom` and `Or`, respectively, is always dispatched to an appropriately parametrised implementation at the *smaller* class, i.e. in the example at the class `Or`.

The pseudo-code presented in Fig. 5.3 is also to be understood in that sense, that is, by considering the above ordering and the described dispatching algorithm. So, for example, the pseudo-code does not seem to cover the invocation `myAtom.join(myOr)`, but this is done by swapping the arguments and invoking `myOr.join(myAtom)` which is covered by Fig. 5.3.

Let us clarify the pseudo-code of Fig. 5.3. In lines (2a) and (2b), both `this` and the argument `r` are expected to be conjunctions of one or more `Atoms`. Those lines hence collect all those atomic relations in a single set, namely `A`.

Likewise, lines (4a) and (4b) collect all disjuncts of `this` and `r` in the set `A`. Some of the given methods' parametrisations do not determine the actual class of the respective argument. For example, the bottom method just expects an instance of any subclass of `AbstractOr`. So, in order to collect all conjuncts or disjuncts, we need to implement appropriate accessors.

We shall also state the purpose of lines (3) and (5) here. Line (3) is to remove all those atomic constraints from the set `A`, that are implied by some other member of `A`. Implication is here meant according to Def. 19. Thus, the invocation will discard all overestimations and e.g. simplify the conjunction $x = 3 \wedge x \in [2, 4)$ to $x = 3$.

In the bottom implementation of `or`, the intention of line (5) is analogous: This time, the weaker constraints will be kept. `minimalDisjuncts` is to discard all stronger, that is, implying, relations. As an example, the disjunction $x = 3 \vee x \in [2, 4)$ will collapse to the weaker of both relations, i.e. to $x \in [2, 4)$ which is clearly implied by $x = 3$.¹

Of course, in order to maintain the DNF also for any newly instantiated relation, we have to make sure that all returned instances be again in DNF. This is indeed easy to see by investigating all returning lines of code. Hereby, the only unobvious exception is the line that immediately follows line (1).

But also here `result` is going to be in DNF, provided that line (1) always produces a relation in DNF. This, however, may be assumed by an inductive argument which will work as long as we can guarantee the termination of `join` and `or` invoked for any pair of relations.

So, let us discuss - as the final issue in this subsection - the termination of the given pseudo-code. Assuming that neither the execution of line (3) nor that of line (5)

¹The actual implementation of the two static methods `minimalConjuncts` and `minimalDisjuncts` is straightforward. As the basic building block of both methods we will need a boolean function implemented at `Atom` that expects a second instance of `Atom` and decides whether the first atom implies the second.

involves invocations of `join` and `or`, the only problem with regard to termination, is again line (1). Here, the hard bit is the embedded invocation `r.join(d)`. Note that the `or` part of line (1) is trivial since the implementation of `or` does not deploy recursion.

For that embedded call, the worst case materialises for the most complex choice of `r`, that is, for `r` being also an instance of `Or`. But then, any embedded execution of line (1) will just call `join` for a pair of conjunctive constraints, i.e. for a pair of `AbstractAnds`. But this invokes the method starting at line (2a) which does not impose recursive calls.

The entire argument shows that the execution of line (1) will result in a bunch of invocations of the `join` method implemented at `AbstractAnd`, and will hence always terminate. Consequently, the implementations of `join` and `or` are always going to terminate and produce `Empty`, `Full` or a relation in DNF, no matter for what pair of relations invoked.

5.3 A Partially Solved Form for Conjunctive Constraints

We have seen above that in our implementation of `RCS`, we chose the DNF as canonical form for all relations.

Furthermore, each disjunct, that is, each conjunction of atomic constraints may in turn be represented in a special form. Clearly, such an additional standardisation should only then be introduced when we expect advantages for the implementation of important services.

This is the starting point for the so-called *partially solved form (PSF)*. As will soon become clear, that form has been tailor-made to support the *project* operation.

The PSF has a strong correspondence to *van Hentenrycks* and *Imberts solved form* for linear equations, disequations and inequalities; see [43]. However, whereas there inequalities (\leq , $<$, \geq , $>$) are translated into equations ($=$) using slack variables, the PSF presented here does also allow for explicit inequalities. Moreover, our PSF may also mention non-linear arithmetic constraints. [43] guarantees that a conjunction of atomic constraints is satisfiable if and only if it can be transformed into solved form. Secondly, any system in solved form that imposes a fixed value for some variable, e.g. x to be 3, will explicitly mention the corresponding assignment, i.e. the constraint $x = 3$. We shall below verify a few somewhat weaker but still useful properties for our PSF.

First, we need to fix some terms by means of exact definitions.

Definition 21 (Arithmetic Term etc.)

Each of the following objects will be called an *arithmetic term*;

1. any variable x with $\text{dom}(x) \subseteq \mathbb{R}$,
2. any value $v \in \mathbb{R}$, and
3. $t_1 \circ t_2$, for any (previously defined) arithmetic terms t_1, t_2 and each binary arithmetic operator $\circ \in \{+, -, \cdot, \div\}$.

Any arithmetic term t has a set of **variables**, $\text{vars}(t)$, which is in the first case defined to be $\{x\}$, in the second \emptyset and in the third case $\text{vars}(t_1) \cup \text{vars}(t_2)$.

Given an assignment $\alpha : \text{vars}(t) \rightarrow \bigcup_{x \in \text{vars}(t)} \text{dom}(x)$ that maps each x to some element of $\text{dom}(x)$, we can define the **evaluation** $t(\alpha)$, by replacing in t each variable by the corresponding value, and ordinarily evaluating the resulting arithmetic term of real numbers. Note however that this may fail in case of division by zero.

An **arithmetic atomic constraint** is a binary relation of two arithmetic terms, $t_1 \diamond t_2$, where $\diamond \in \{=, \neq, \leq, <, \geq, >\}$. In the case $\diamond \equiv =$, we call it an **equation**, for $\diamond \equiv \neq$ a **disequation** and otherwise an **inequality**. For $\text{vars}(t_1) \cup \text{vars}(t_2) \neq \emptyset$, $t_1 \diamond t_2$ is a shorthand for the non-trivial constraint (X, \mathcal{A}) , where

$$\begin{aligned} X &= \text{vars}(t_1) \cup \text{vars}(t_2), \text{ and} \\ \mathcal{A} &= \left\{ \alpha : X \rightarrow \bigcup_{x \in X} \text{dom}(x), \right. \\ &\quad \left. x \mapsto v \in \text{dom}(x) \mid t_1(\alpha) \diamond t_2(\alpha) \right\}. \end{aligned}$$

Thereby, the validity of the latter condition $t_1(\alpha) \diamond t_2(\alpha)$ is to subsume the assertion that neither of the two evaluations $t_1(\alpha), t_2(\alpha)$ fails.

If contrariwise, $\text{vars}(t_1) \cup \text{vars}(t_2) = \emptyset$, then $t_1 \diamond t_2$ stands for \odot or \square depending on whether the ordinary evaluation of the variable-free formula $t_1 \diamond t_2$ yields false or true, respectively.

Before an example will be given, we will now define our *partially solved form*.

Definition 22 ((Partially) Solved Form)

Let c be an arithmetic atomic constraint. Then $x \in \text{vars}(c)$ is called a **basic variable** of c if and only if $c \equiv x \diamond t$, where t is an x -free arithmetic term, that is, $x \notin \text{vars}(t)$, and c is not a disequation, i.e. $\diamond \in \{=, \leq, <, \geq, >\}$.

Let now $A \equiv a_1 \wedge a_2 \wedge \dots \wedge a_n$, $n \in \mathbb{N}_+$, denote a conjunction of arithmetic atomic constraints. A variable $x \in \bigcup_{1 \leq i \leq n} \text{vars}(a_i)$ is named **basic variable** of A if either

1. $x \in \text{vars}(a_i)$ holds for exactly one $i \in \{1, 2, \dots, n\}$, a_i is an equation, and x is a basic variable of a_i , or
2. $\forall i \in \{1, 2, \dots, n\} \quad x \in \text{vars}(a_i) \implies (a_i \text{ is an inequality and } x \text{ a basic variable of } a_i)$.

Let B denote the set of all basic variables of A , and $X = \bigcup_{1 \leq i \leq n} \text{vars}(a_i)$ be all variables in A . Then, we say that A is in **partially solved form (PSF)** for Y , for any $Y \neq \emptyset$, $Y \cap X \subseteq B$. Moreover, if $B = X$, then A is said to be in **solved form (SF)**.

Finally, an arbitrary disjunction of conjunctions of arithmetic atomic constraints is in **PSF** for X / in **SF** if and only if each of its conjunctions is in PSF for X / in SF. In this case, any $x \in X$ is a **basic variable** of the disjunction.

Although this section stresses the view on conjunctive constraints, the last part of Def. 22 actually defines the terms (P)SF also for any arithmetic relation in DNF. This is due to the necessity to implement *project* for any relation in our framework and hence for any relation in DNF. Still, the (P)SF is primarily a property of conjunctions.

Example 10: The arithmetic term $(x + y) - y$ has, according to the above Def. 21, the variables $\{x, y\}$. Obviously, y cancels out; therefore the arithmetic atomic constraint $z = (x + y) - y$ with the variables $\{x, y, z\}$ contains y as a free variable.

Note however that one must be careful with cancellation in the case of quotients: $x \div x$ and 1 must be regarded distinct terms since evaluating the former, given the assignment $(x \mapsto 0)$, will fail but not pose a problem for the latter. Consequently, x is not a free variable in $y = x \div x$ since this constraint implicitly forbids x to take the value 0. This point has already been stressed in Def. 21, where we mentioned the possibility of an evaluation to fail. The validity of $t_1(\alpha) \diamond t_2(\alpha)$ was there meant to exclude the failure of either term's evaluation. We will therefore sometimes also speak of *implicit constraints*: $y = x \div x$ implies the implicit constraint $x \neq 0$ which must be observed in order to prevent the evaluation of $x \div x$ from failing.

The above constraint $z = (x + y) - y$ as well as $z = x \cdot y$ have the basic variable z . Note that the *basic variable* of an arithmetic atomic constraint must always appear isolated on the left-hand side. Therefore, we will - on a syntactical level - distinguish between the two representations $z = x \cdot y$ and $x \cdot y = z$ of the same constraint. The latter has, according to Def. 22, no basic variable.

The conjunction²

$$x = \frac{y}{z} \wedge z \geq -1 \wedge w < 3 \cdot y \wedge w \geq y + 7 \wedge w > y - \frac{z}{y} \wedge y > 3 \quad (5.1)$$

is in PSF for $\{x, w\}$, but for no set of variables that intersects with $\{y, z\}$. Consequently, the entire conjunction is not in SF. x is basic according to the first criterion of Def. 22, w according to the second one. Note that y and z may be seen as parameters for x and w : Once those have become known, all atomic constraints can immediately be simplified in order to impose tightest restrictions for x and w .

Regard now the algebraic equivalence

$$z = x \cdot y \quad \sim \quad (x = \frac{z}{y} \vee (y = 0 \wedge z = 0)). \quad (5.2)$$

The left-hand side constraint has, as already mentioned, the basic variable z ; the right conjunction is in PSF for $\{x\}$ (although x does not appear in the second disjunct; cf. Def. 22). Again, $x = \frac{z}{y}$ implies the implicit constraint $y \neq 0$ which is therefore not made explicit in (5.2).

How will the PSF be used in the context of our implementation of RCS?

Suppose the conjunction (5.1) in Ex. 10 is the result of some join that takes place during the forward phase. This situation is depicted in Fig. 5.4. Suppose furthermore that the subsequent projection is to eliminate the basic variables x and w . (We shall soon see that elimination becomes trivial for basic variables.) Clearly, the remainder of the forward phase will deal with constraints involving y and z . So, those two variables will be eliminated later, that is, in a node that is closer to the

²When using the shorthand notation for arithmetic constraints, we will usually write " \wedge " instead of \bowtie .

root of the constructed aggregation tree, cf. Fig. 5.4.

Assuming a consistent problem, the backward phase will compute tightest restrictions for z and y *before* those for x and w ; suppose z turns out to be 1 and $y = 4$. As already suggested in Sect. 4.3, where we discussed issues of the backward phase, it becomes now even clearer why storing the forward join is highly beneficial; cf. also Fig. 5.4: In order to compute tightest restrictions for x and w , we just need to substitute all known values into the forward join which produces the bottom right conjunction. From that we can simply *read off* the desired tightest restrictions: $x = 4$ and $w \in [11, 12)$.

Already this example shows that tightest restrictions may be real intervals instead of fixed real numbers. Therefore, the proposed substitution of already computed tightest restrictions into the stored forward join, may actually replace some variables by intervals. We shall fix that problem when we discuss arithmetic terms in more detail. Basically, we can stick to the given procedure by implementing all arithmetic operators also for prominent subsets of \mathbb{R} , as e.g. intervals.

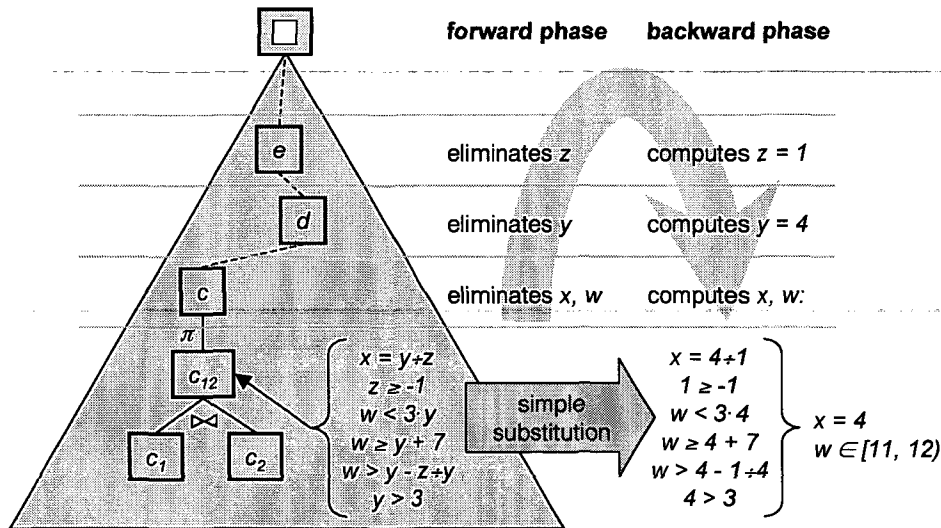


Figure 5.4: Advantage of the PSF in the RCS Framework

Example 10 also gives a clue as to how the idea explained in Fig. 5.4 can be extended to work also when the forward join is a disjunction of conjunctive constraints. Regard again the equivalence (5.2). RCS may be asked to eliminate x . Then we should prefer to store the right-hand side conjunction as forward join. Because then, the computation of x in the backward phase will also work by simply substituting tightest restrictions for y and z .

5.4 An Implementation of Projection Based on the Partially Solved Form

In the previous section, one strong argument for the PSF was explained that has to do with the simplicity of joins performed during the backward phase: At a given aggregation node n , we only need to store that representation of the forward join that mentions as basic variables all the variables eliminated at n ; see again Fig. 5.4.

A further advantage shall be explained in the first subsection, below. As will turn out, the elimination of basic variables becomes very easy. Also, we can guarantee that the relation returned by our algorithm represents indeed the correct result according to Def. 5. So, when eliminating a basic variable we will not just obtain an overestimation, as assumed in Th. 6.

Summarising, once we have managed to find a representation of our relation, in which all variables that we are going to eliminate are basic, the elimination itself can be accomplished efficiently and correctly. Moreover, that representation will significantly support the respective backward join.

The remaining question is then, how to establish that useful representation. This is going to be addressed in the second subsection, i.e. in Subsect. 5.4.2.

5.4.1 Projection

Let us once more consider the conjunction A of arithmetic atomic constraints in (5.1). Suppose we were to eliminate x and w from A .

The result of those eliminations will no longer mention x and w , but only y and z . In order to compute the correct projection, we need to keep all those (y, z) -tuples that can be completed to solution tuples over (y, z, x, w) . So, what are those (y, z) -tuples?

First of all, we need to pay attention to the domains of y and z . Secondly, all atomic constraints c in A with $\text{vars}(c) \subseteq \{y, z\}$ must hold, that is, $z \geq -1$ and $y > 3$. Thirdly, we are permitted to pick only those values for y and z that observe all implicit constraints, i.e. that allow for an evaluation of all arithmetic terms in A without failure. In our example, this excludes the choices $z = 0$ and $y = 0$.

Let us now pick an arbitrary (y, z) -tuple that observes the above three conditions, e.g. $y = 3.1$ and $z = 1$. We can then immediately compute $x = y \div z = 3.1 \div 1 = 3.1$; thus any (y, z) -tuple can be enlarged to a (y, z, x) -tuple.

For w the task is however more subtle since w and x are basic variables for different reasons, cf. conditions 1. and 2. of Def. 22. Substituting $y = 3.1$ and $z = 1$ gives us the bunch of inequalities

$$w < 3 \cdot 3.1 \quad \wedge \quad w \geq 3.1 + 7 \quad \wedge \quad w > 3.1 - 1 \div 3.1$$

that must hold for w but do not hold for any element of \mathbb{R} .

Obviously, the feasible (y, z) -region defined by the above three conditions is not restrictive enough, since it allows for a choice for y that violates constraints in A . It must hence further be narrowed so that it exclude at least $(y, z) = (3.1, 1)$. We can do so by ensuring that our choice for y satisfy the constraint $3 \cdot y > y + 7$ which is

implied by A , namely by $w < 3 \cdot y$ and $w \geq y + 7$. Then the choice $y = 3.1$ would be forbidden right from the start.

That kind of fixing the problem is well-known and is subsumed under the ideas related to *Fourier elimination*, [28]: For any two inequalities $w \prec t_{upper}$ and $w \succ t_{lower}$ with $\prec \in \{<, \leq\}$ and $\succ \in \{>, \geq\}$, we just add the implied inequality $t_{lower} \prec t_{upper}$, with $\prec \in \{<, \leq\}$ appropriately. After that kind of preprocessing our conjunction A , we can always complete a partial tuple for (y, z) to one over (y, z, x, w) , provided our choices for y and z observe all constraints c with $vars(c) \subseteq \{y, z\}$.

Fourier elimination has been the subject of study in many works. The main problem here is that we will have to add one implied constraint per pair of inequalities that provide a lower bound term and an upper bound term for the variable to be eliminated. In general, *Fourier elimination* becomes, due to that reason, doubly exponential in effort and thus impracticable. This is also mentioned in [46] which also cites work by Lassez, Černikov and Huynh related to the problem of keeping the number of added inequalities small; see also [42]. In [46], the identification of *all* those additional inequalities that are not implied by other constraints and must hence indeed be added, is pointed out to be impractical. Interestingly, [46, Sect. 3.2.] also alludes to the density of coefficient matrices for purely linear problems, with advantages in the case of a sparse matrix. In RCS, this relates again to our *low density assumption* in compositional system models.

Moreover, by adding just very few, namely the most “important”, implied inequalities, we may manage to keep *Fourier’s elimination* tractable. Clearly, by omitting some implied inequalities, we will obtain as projection an overestimation of the exact relation. But this fits nicely in our implementation of RCS; cf. Th. 6.

Note that the elaboration undertaken here is very similar to that in [46]. See also [35], in which the first figure presents a projection algorithm that combines *Gauss* and *Fourier elimination*, very much like in our implementation of RCS.

Before we provide the pseudo-code for our projection procedure and state that it yields the correct relation, let us gather the bits and pieces developed above in a single chart. Figure 5.5 explains in detail the actual elimination of x and w from the conjunctive constraint in (5.1).

Basically, there are three steps. The first one adds all implicit constraints and the inequalities that are due to *Fourier*, that is, one inequality for any pair of lower-bound and upper-bound inequalities for the basic variable w . The second step discards, from the enlarged conjunction, all original atoms that mention a basic variable; in Fig. 5.5 contained in grey boxes. The final step is a call to the previously introduced method `minimalConjuncts`; see line (3) in Fig. 5.3. It is to remove as many atoms as possible that turn out to be implied by some other atom in the set of conjuncts. Figure 5.5 indicates those implications by arrows. We can drop all atoms to which an arrow points. The result of the procedure is the conjunctive constraint on the right.

Figure 5.6 presents the pseudo-code that captures the illustration of Fig. 5.5. The shown methods eliminate a set of variables X for which the represented relation is assumed to be in PSF.

Since we prefer to represent any unsatisfiable relation by the singleton instance of the class `Empty`, and similarly any nonrestrictive one by `Full`, line (1) is obvious.

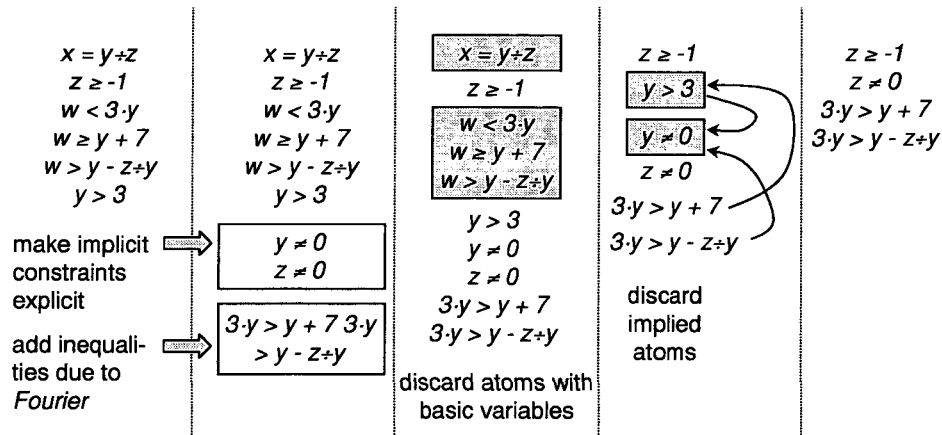


Figure 5.5: Elimination of Basic Variables from a Conjunction

Apart from trivial constraints, the given pseudo-code deals with relations in DNF, the atomic building blocks of which are arithmetic atomic constraints. We will not consider other types of atomic constraints here, but it is straightforward to extend the respective algorithms to a larger class of constraints, as had been carried out in our implementation of RCS. An overview over the constraint language accepted by it will be given in the last section, Sect. 5.6, of this chapter.

Just as the conceptual construction of the PSF as given in Def. 22 suggests, the implementation at `Or` is a simple iteration over all disjuncts. The result of that middle method is thus the disjunction of the set of results for the disjuncts.

The interesting part is the bottom method, that is, the one for conjunctive constraints. Following the algorithm depicted in Fig. 5.5, line (2) enlarges A by adding all implicit constraints. A formal definition of an *implicit constraint* could easily be given but is omitted since the term should be intuitively clear: Due to Def. 21, the only type of implicit constraint arises in connection with division by zero. Therefore, we will basically obtain one implicit constraint per quotient, stating that the divisor must not equal zero.³

Whenever the test in line (3) is positively answered, then the variable $x \in X$ is basic due to condition 2. of Def. 22. The loops in lines (4) and (6) will further enlarge A by inserting implied inequalities due to *Fourier*. The two remaining steps illustrated in Fig. 5.5, of discarding certain atoms, are both carried out in line (8). The remainder of the method deals with the number of resulting atomic constraints and is obvious.

What will be the result when the bottom method is invoked for a single atom in

³In our prototype, we actually allow for further functions like e.g. \log , represented by an additional subclass `Log` of the class `Term`. Then, any constraint $y = \log(x)$ implies the implicit constraint $x > 0$ that can be requested from the constraint and provided by an appropriate method implemented at `Log`.

solved form, for example for $x = y \cdot z$ with the aim to eliminate x ? There is no implicit constraint here, and the if-block starting in line (3) will completely be skipped. Line (8) leaves A empty, and the sole instance of Full is returned.

```

Empty/Full:    function eliminateBasic(Set of Variables X)
(1)    return this

Or:            function eliminateBasic(Set of Variables X)
result ← sole instance of Empty
for each disjunct d of this Or
    result ← result.or(d.eliminateBasic(X ∩ vars(d)))
return result

AbstractAnd:   function eliminateBasic(Set of Variables X)
A ← set of atoms in this conjunction
(2)    insert all implicit constraints in A
AX ← {a ∈ A | X ∩ vars(a) ≠ ∅}
for each x ∈ X
(3)    if (x appears in an inequality of AX)
        T≤ ← {t | (x ≥ t) ∈ AX}
        T< ← {t | (x > t) ∈ AX}
        T≥ ← {t | (x ≤ t) ∈ AX}
        T> ← {t | (x < t) ∈ AX}
(4)    for each (tlower, tupper) ∈ T≤ × T≥
        a ← Factory.makeAtom(tlower, ≤, tupper)
(5)    if (a.isEmpty()) return (sole instance of Empty)
        else insert a in A
(6)    for each (tlower, tupper) ∈ ((T≤ ∪ T<) × (T≥ ∪ T>)) \ (T≤ × T≥)
        a ← Factory.makeAtom(tlower, <, tupper)
(7)    if (a.isEmpty()) return (sole instance of Empty)
        else insert a in A
(8)    A ← minimalConjuncts(A \ AX)
        if |A| = 0 return (sole instance of Full)
        else if |A| = 1 return (sole atom in A)
(9)    else return Factory.makeAnd(atoms: A)

```

Figure 5.6: Pseudo-Code for Eliminating a Set of Basic Variables

Some words should be said concerning lines (5), (7) and (9). When we briefly introduced the class `Factory`, we pointed out that its creation methods perform some simple checks and manipulations. Note that the atom a in the lines (5) and (7) may have no solution, in which case `eliminateBasic` will return the sole instance of `Empty`. An example is the atomic constraint $t + 1 \leq t$ for any arithmetic term t . According to Lem. 4, (3.7), this means that the original relation was also unsatisfiable. In that case, our prototype was not able to detect that unsatisfiability before the

elimination. This is an example where we represented some relation r with $r \sim \oslash$ not by the unique instance of **Empty**.

Likewise, the creation process in line (9) may be left with the set $A = \{x < y, y < x\}$, and thus line (9) may also return the sole instance of **Empty**.

Let us state and prove that the code in Fig. 5.6 returns indeed a representation of the projection of the original relation:

Lemma 11

Let r be an arbitrary non-trivial constraint in DNF such that the terminal building blocks are arithmetic atomic constraints. Suppose furthermore that r is in PSF for some set of variables X .

Then, taking a representation of r and applying to it the pseudo-code in Fig. 5.6 produces a representation of $\pi_{vars(r) \setminus X}(r)$.

Suppose now that we are given an arbitrary arithmetic constraint r in DNF, as in Lem. 11. How can we implement the *project* operator as required by the relational engine?

Assuming a method that turns any given variable in $vars(r)$ into a basic variable, we can immediately utilise the above procedure. So, if we are to compute a representation of $\pi_{vars(r) \setminus X}(r)$, where $X \subseteq vars(r)$, there is an obvious way to do so: We pick some variable that is to be eliminated, $x \in X$, make it basic in r and eliminate it. Afterwards, we proceed with the result $\pi_{vars(r) \setminus \{x\}}(r)$; see Lem. 11. This implements a loop over all variables that are to be eliminated.

In our concrete implementation of RCS, we chose an alteration of that straightforward approach. There, we try in each iteration to make as many variables of X basic as possible. Then all those are eliminated at once. For completeness sake, we shall not omit the pseudo-code for the important *project* operator; see Fig. 5.7.

The set E contains all variables that need to be eliminated, in order to obtain the projection *onto* Y . Clearly, we try to apply **eliminateBasic**, see Fig. 5.6, with E as parameter. To this end, any element of E needs first to be made basic.

Line (1) captures an obvious termination criterion. If that does not apply, line (2) calls the method we are investigating in the next subsection. It tries to make basic in the given relation as many variables of the argument set, that is, of E . Afterwards, F records the portion of E for which **toPSF** succeeded. In line (4), we know that none of the elements of E could be made basic. Furthermore E is certainly not empty. Since our implementation depends on being able to make a variable basic before it can be eliminated, we clearly cannot complete the elimination. That is why line (4) throws an exception.⁴

Otherwise, that is, in line (5), $|F|$ must be at least 1 and the altered version of r will have one less variable. Therefore, the recursive call in line (6) will always approach a simpler task, and the entire method will hence always terminate.

Does the result indeed represent the desired projection? When terminating in line (1), the set Y must - by construction of E - subsume all variables of the original

⁴Note however that most high-level implementations at the relational engine, as for instance the forward phase, would still function even when we could not eliminate all proposed variables.

Relation, and returning *this* is the right measure.

Termination due to line (3) happens when *toPSF* discovered that the original relation is unsatisfiable, and the returned relation is obviously correct.

```

Empty/Full:    function project(Set of Variables Y)
                return this

AbstractOr:    function project(Set of Variables Y)
                E ← (variables of this relation) \ Y
(1)  if (E = ∅) return this
                else
(2)    r ← this.toPSF(E)
(3)    if r.isEmpty() return (sole instance of Empty)
        F ← (basic variables of r) ∩ E
        if (F = ∅)
(4)      throw exception "overestimating projection"
        else
(5)      r ← r.eliminateBasic(F)
(6)      return r.project(Y)

```

Figure 5.7: Pseudo-Code for Computing Projections

It remains the case of termination in line (6). Note that, when approaching line (5), *r* represents the same constraint as *this*, since line (2) may alter the representation but must preserve the represented constraint (up to equivalence), as discussed in the following subsection. By Lem. 11, line (5) turns *r* into some projection of the original relation and eliminates at least one variable. Note however that we only eliminate elements of *E* and therefore never of *Y*. Consequently, each execution of line (5) during the recursion will produce projections with less and less variables. Finally, all variables apart from the ones in *Y* will have been eliminated. This informal argumentation shows that the code in Fig. 5.7 observes indeed the specification of *project*.

The missing building block in our presented implementation of *project* is the method *toPSF*. We shall now turn to that remaining issue.

5.4.2 Establishing the Partially Solved Form

The previous paragraph made clear what the specification of *toPSF* is:

- The relation for which the method is invoked and the resulting relation must be equivalent⁵, and

⁵Following the specification of our prototype, we may actually return an overestimation of the original relation. Then, *project*, as implemented in Fig. 5.7, would also in general return only an overestimation of the actual projection.

- as many variables as possible of the provided argument set should be basic variables of the returned relation.

As to the second item, it is noteworthy that the pseudo-code given in Fig. 5.7 will work fine, as long as each invocation of `toPSF` in line (2) turns at least one variable into a basic variable.

The algorithm for making a certain set X of so-called *goal variables* basic in an arithmetic constraint, is, at least on an abstract level, rather straightforward. Note that we need to implement `toPSF` only for all terminal subclasses of `AbstractOr`, cf. Fig. 5.7. Regard the pseudo-code in Fig. 5.8.

```

Or:      function toPSF(Set of Variables X)
        result ← sole instance of Empty
        for each disjunct d of this Or
(1)      result ← result.or(d.toPSF(X ∩ vars(d)))
        return result

AbstractAnd: function toPSF(Set of Variables X)
(2)      Y ← X ∩ (non-basic variables of this conjunction)
        if Y = ∅ return this
        else
            (y, A) ← this.getBestEnrichment(Y, X)
(3)      if (previous line threw exception) return this
(4)      A' ← {a.solvedFor(y) | a ∈ A}
            if (previous line threw exception) return this
            C ← (set of atoms of this conjunction)
            if (A' = {y = t})
(5)      B ← {b.substitute(y, t) | b ∈ C \ A}
            else
                B ← C \ A
(6)      r ← A' ⋈ B
            if r.isEmpty() return (sole instance of Empty)
(7)      else return r.toPSF(X ∩ vars(r))

```

Figure 5.8: Pseudo-Code for Making a Set of Goal Variables Basic

Let us first explain the presented pseudo-code, and then talk about issues of termination and correctness with respect to the specification of the method. The section closes with some remarks concerning the two methods at atomic level, `solvedFor` and `substitute`, see lines (4) and (5), respectively.

A first investigation reveals that any call of `toPSF`, be it in line (2) of Fig. 5.7 or in lines (1) and (7) of Fig. 5.8, is parametrised with a set of variables that form a subset of the respective relation's variables.

The top implementation at `Or` treats the case of a disjunctive constraint. For that,

we need to iterate over all disjuncts d and try to accomplish the task of making as many variables as possible of $X \cap \text{vars}(d)$ basic. Afterwards, the results of those embedded invocations will be consecutively or-ed, in order to produce the final result. The bottom method is imposed on a single atomic constraint or on a conjunction of those.⁶

Line (2) identifies the interesting portion of X , that is, the variables that are not yet basic but shall be made basic. Clearly, if no such variable exists, the task of `toPSF` is performed and there is nothing else to do. Otherwise, a heuristic, encapsulated in `getBestEnrichment` is asked to provide a *best enrichment* with the following properties:

1. If there is a “good” equation in the conjunction that may be used to provide a substitution term for some goal variable $y \in Y$, then y and that equation will serve as the best enrichment.
2. Otherwise, there may exist a goal variable y that appears exclusively in a set A of inequalities, each of which can be reformulated so that y becomes basic. In that case, (y, A) is proposed as the best enrichment.
3. Both previous cases propose that best enrichment only when making y basic will not make any of $X \setminus Y$ non-basic again. For example, (5.2) illustrates that turning x into a basic variable makes the previously basic z non-basic again. This property of `getBestEnrichment` ensures the *monotonicity* of `toPSF`.
4. When no best enrichment can be proposed that satisfies the above conditions, then `getBestEnrichment` will throw an exception that is going to be caught and treated by `toPSF`.

In line (3), when the heuristic failed, we just return `this`. Note that, since Y is not empty, the returned relation is in PSF only for some proper subset of X .

Addressing 1. and 2. of the above enumeration, we clearly have to reformulate all atoms provided by A . Depending on the difficulty of that task and on how mature our implementation is, the class of atomic constraints for which this reformulation can be accomplished will be smaller or larger. We shall come back to this point below. However, in order to make the goal variable y basic, we are dependent on the success of those reformulations. Consequently, in case of failure, the method will give up and just return `this` once again.

Whereas the previous parts of the procedure apply, regardless of whether y is going to be basic according to condition 1. or 2. of Def. 22, the following `if`-statement distinguishes between the two situations: In line (5), we utilise the single member of A' , $y=t$, to replace any occurrence of y in the remainder of the conjunction by the term t . Note that, since y is basic in $y = t$, t does not mention that variable.

Approaching the `else` part, we know that only the atoms in A mention y ; cf. item 2. in the above description of our heuristic. Therefore, none of the elements of B will

⁶Our realisation of the class `And` facilitates some attributes and fast accessor methods for providing all basic and non-basic variables of a conjunction. Furthermore, given a basic variable x , the class `And` enables us to retrieve all atomic constraints of the conjunction in which x is the basic variable.

mention y .

Assuming that `solvedFor` as well as `substitute` will return equivalent relations, it is easy to verify that line (6) computes a relation that is equivalent to `this`. Moreover, only the atoms in A' mention y as the basic variable; thus, r is in PSF for some superset of $\{y\}$.

By the above argumentation, we have almost come to concluding that `toPSF` computes indeed an equivalent relation in PSF for X , unless aborted untimely due to an exception. As already pointed out, our heuristic captured by `getBestEnrichment` guarantees the monotonicity of `toPSF`. Therefore, the recursive invocation in line (7) is imposed on a relation r , which is in PSF at least for those variables of X that are also basic in `this`. Furthermore, we have just emphasised that r is also in PSF for some superset of $\{y\}$. But y is not a basic variable of `this`, and so r must have at least one more basic variable in X than `this`. This proves that `toPSF` always terminates and makes as many variable of X as possible basic. Clearly the latter statement must be seen in connection with the maturity of our heuristic and of the implementation of `solvedFor`.

Also, we have already explained that r in line (7) will always represent a constraint equivalent to the one represented by `this`. Consequently, `toPSF` always returns a constraint that is equivalent to the one to which that method was initially applied.

The substitution in line (5) has been implemented at the class `Atom` by performing the substitution in both terms and building a new atomic constraint from the resulting two terms. So, substitution should be seen mainly as an infrastructural element that needs to be provided by the term module, see Fig. 5.1. Since that module maintains symbolic representations of arithmetic terms, the naive replacement of a variable by some substitution term will very often produce large instances. Repeated substitutions are likely to worsen the situation. The below section of terms will hence address the issue of term manipulations and simplifications.

Likewise `solvedFor` can be implemented in the term module. Given an atom $f(x) \diamond t$ that needs to be solved for x , we just have to provide the implementing term $f(x)$ with the second term t and the binary relation \diamond represented by `RelationType`, cf. Fig. 5.1. The idea here is to apply the inverse of the function f to both terms, in order to make x basic. However, f need not be invertible or may be an n -ary function with $n \geq 2$ as in the case of arithmetic operators. Moreover, \diamond is allowed to be one of $\{\leq, <, \geq, >\}$ which makes the reformulation task even more complicated. Just as applying f^{-1} to the left-hand side produces a simpler term, the right-hand side will usually become more involved. Again, the ability to perform term simplifications turns out inevitable.

The equivalence given in (5.2) shows that making a certain variable basic is not at all trivial. In general, the symbolic manipulations needed in the case of arithmetic constraints, set harsh requirements for the underlying manipulator. In our implementation of a relational processor, also the reformulation presented in (5.2) can be carried out.

More generally, our prototype allows for isolating any variable that appears only once in an arithmetic atomic constraint: Let $t_x \diamond t$ be such an atom, that is, $x \in \text{vars}(t_x) \setminus \text{vars}(t)$, and t_x , written as a flat algebraic expression, mentions x exactly once. Then, our prototype can produce a relation r such that r is in DNF and in PSF for $\{x\}$, and $r \sim t_x \diamond t$. The implementation is straightforward and works, in the case of inequalities, by a case differentiation as exemplified in the following example:

Example 11: Suppose we need to turn x into a basic variable in the atom $(z+x) \cdot y < z$. Then this works by applying the appropriate inverse operations until x is isolated. When dividing by y , we need to differentiate three cases:

$$\begin{array}{ll} (z+x) \cdot y < z & | \div y \\ \sim (y = 0 \wedge 0 < z) \vee (z+x < \frac{z}{y} \wedge y > 0) \vee (z+x > \frac{z}{y} \wedge y < 0) & | - z \\ \sim (y = 0 \wedge 0 < z) \vee (x < \frac{z}{y} - z \wedge y > 0) \vee (x > \frac{z}{y} - z \wedge y < 0) & \end{array}$$

Note that this process is monotonous in that the right-hand side terms involving x become simpler and simpler.

The sketched algorithm enables our prototype to also *actively manipulate non-linear constraints*, whereas most existing constraint solvers for arithmetic constraints *postpone*, and thereby temporarily unconsider, non-linear constraints until they have sufficiently simplified.

[9] explains this postponing scheme. Any arithmetic problem is split into a linear portion L and a non-linear one, N . Then the solving algorithm considers first the linear subproblem L and solves it. In a second step, the solutions of the first step are substituted into N . The hope is here that thereby some of the previously non-linear conditions become linear and can be moved from N to L . The overall solving scheme performs several iterations of the outlined procedure, in order to finally solve all variables.

An overview over common algorithms and the main ideas concerning non-linear constraints has already been given in Subsect. 2.4.2.

5.5 Managing Arithmetic Terms and Values

5.5.1 Overview

Chapter 4 of this work developed implementations of the high-level algorithms of RCS. The presented solutions rest upon a set of services of a relational processor which are, in turn, elaborated in this chapter.

Following that top-down approach, we have now come to the connecting point between relations and terms. Lines (4) and (5) of the pseudo-code in Fig. 5.8 deploy two very basic operations implemented at **Atom**, and the above given remarks suggest that those operations can indeed be seen as issues concerning terms.

In this section, we shall not continue with the strict refinement of the two mentioned unrefined operations of Fig. 5.8. Their intention and informal specification has already been given. Instead, we will end the top-down development of pseudo-code here. This section presents an overview over arithmetic terms and a possible

class architecture, as realised in our prototype. The focus will not so much be on concrete services required by upper layers of our implementation, but on issues of representing terms and efficiently simplifying them.

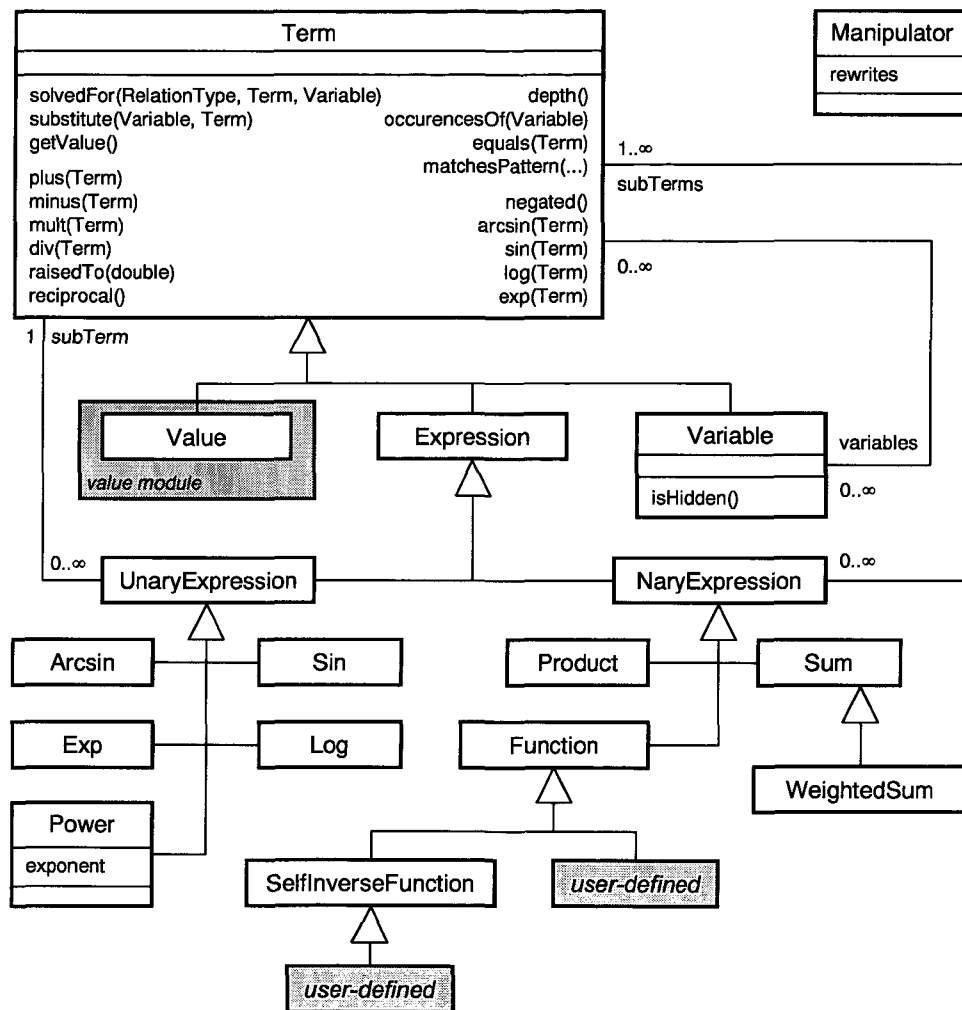


Figure 5.9: UML Diagram with the Architecture of Real-Valued Terms

Moreover, we will, for the sake of conciseness, restrict ourselves to terms with evaluations in \mathbb{R} . These terms will form a superset of arithmetic terms as defined by Def. 21 but still not make up all the objects that are supported by our current prototypic implementation. A summary of that latter class of all processable terms will be given in the final section of this chapter on constraint language and control aspects.

Figure 5.9 presents the term classes with real-valued evaluations that have actually been implemented. Note that, as already mentioned, there are in fact more classes in the term module of our prototype. Furthermore, the UML diagram depicts the most important operations and interrelations between the shown classes.

The top left class **Term** is the top-most superclass in the hierarchy. The box entails methods for arithmetic operations and some additional supportive methods. We are going to provide brief descriptions for all of them in this section. An extra paragraph is devoted to issues of automatic term manipulation and simplification. In this context, the top right class **Manipulator** will be discussed.

Each **Term** refers to a possibly empty set of variables and has the structure of a directed acyclic graph (DAG). For the moment, we may just think of that DAG as of a tree with possibly identical subtrees. The fundamental **Terms** without substructure, that is, the leaves of these trees, are **Values** and **Variables**. Again, in this section we shall exclusively focus on values in \mathbb{R} and variables x with $\text{dom}(x) \subseteq \mathbb{R}$. Apart from those terminal elements of the term hierarchy, there are n -ary expressions, i.e. terms with n subterms, $n \in \mathbb{N}_+$. In our code, we distinguish unary terms, for which $n = 1$ and n -ary terms with $n \geq 1$ but preferable $n \geq 2$; see Subsect. 5.5.2.

The most important unary terms for all our current applications, are powers of arithmetic terms, represented by instances of **Power**. Note that thereby, the exponent of a power needs to be a real number, although arithmetic terms with more involved exponents, like e.g. again arithmetic terms, can easily be defined. Other unary terms have been implemented (**Arcsin**, **Sin**, **Exp** and **Log**) in order to test whether our expressional power can quickly be extended on demand. This requirement is mainly due to so far unknown future applications of our solver, that may have to deal, for example, with trigonometric or other conditions.

As to n -ary functions, we mention the classes **Product** and **Sum** for the combination of at least two addends or factors, respectively. An important specialisation of **Sum** is **WeightedSum** which captures all sums that are moreover affine linear combinations of the form $\mu + \sum_i \lambda_i \cdot x_i$. Note that many engineering applications, though potentially non-linear, often simplify, during the course of computation, to purely linear problems. So, an effort has been made to override the more general implementation of **Sum** at **WeightedSum**, in order to treat linear combinations more efficiently.

Finally, the middle bottom cluster shows the class **Function**. That part of Fig. 5.9 is the subject of the next subsection.

5.5.2 User-Defined Functions

Our prototype allows for a relatively unproblematic incorporation of user-defined terms. From an abstract point of view, such terms are to capture a deterministic, functional relationship f , mapping a set of $n, n \in \mathbb{N}_+$, independent inputs x_i , $i \in \{1, 2, \dots, n\}$, to one resulting output $f(x_1, x_2, \dots, x_n)$. Note that here, we allow again for $n = 1$, although **Function** is derived from **NaryFunction**. That is also why we do not strictly constrain an instance of **NaryFunction** to have at least *two* subterms; cf. Subsect. 5.5.1 and the multiplicity in Fig. 5.9.

There are (at least) two good reasons which make it inevitable to provide a facility for incorporating user-defined functions:

- The functional relationship, that is, the mathematical function f , may be highly complex. Not only can the representation by other classes shown in Fig. 5.9 be difficult, but simply impossible. The evaluation of f , given all

input values, may involve numerical methods, as e.g. interpolation or numerical integration.

- f might only be given implicitly, by means of a piece of code or as a batch program or in a dynamic link library. Here, the evaluation of $f(x_1, x_2, \dots, x_n)$ will normally be accomplished by some connected but otherwise unknown module. That situation is expected to frequently arise when we try to replace older solving modules by our constraint solver. The feature of user-defined functions will often enable a gradual migration instead of a far more risky one-step changeover.

In order to incorporate a user-defined function in our prototypic implementation, the user has in practice to program a Java class that is derived from `Function`. In this class, he has to implement only four methods to make the code work.

Example 12: An example for a user-defined function can be given in the context of the so-called automated transfer vehicle (ATV, see Chap. 6). The constraint model processed by our prototype includes a propulsion component that hosts a functional dependency called *chamber pressure*. The corresponding class `ChamberPressure` is a subclass of `Function`, and may substitute the bottom right grey box in Fig. 5.9. Using `ChamberPressure`, we can represent a function f for computing a resulting pressure from two inputs. Here, f is the bi-linear interpolation with respect to a given set of reference points $(x, y, z) \in \mathbb{R}^3$, for which $z = f(x, y)$ is known to hold. In this example, the constraint $z = f(x, y)$ is actually not too hard to represent without user-defined functions. But by introducing `ChamberPressure`, the modelling of the propulsion component becomes a lot more obvious and straightforward.

Clearly, each user-defined function introduces a *directed term*, that is, a term that can only be evaluated. On the contrary, when provided with the output value, we can in general not derive information concerning the input vector. Note that this scenario resembles a simulation setup, in which we can only analyse a modelled system in a directed, one-way manner; see Subsect. 2.2.1.

The situation might however improve, when we have some more knowledge about the user-defined function, for instance when f is known to be *self inverse*. Figure 5.9 shows that further specialisation of `Function`, `SelfInverseFunction`, with the succeeding grey box as a placeholder for actual embodiments. A self inverse function f is a bijective function with $f \equiv f^{-1}$. Then, $f^2 = f \circ f$ is the identity, and $y = f(x) \sim f(y) = f^2(x) \sim x = f(y)$. Obviously, in this scenario, we can retrieve the input value x , whenever the output value y is given. So here, the user-defined function is no longer a directed term in the above sense, since there are ways to get rid of it, at least in the case of equations. For inequalities, the sketched reformulation can still be accomplished in the case of f being a monotonous function.

5.5.3 The Representation of Arithmetic Terms

Before we shall discuss the set of methods implemented for any term, as listed in Fig. 5.9, we need to understand how arithmetic terms will actually be represented

in our prototype. This also includes issues of automatic term manipulation as explicated in the subsection on *computer algebra*, see Subsect. 5.5.4.

What we are going to say in this subsection describes a sort of weak *normal form* that can gradually be made more and more restrictive by imposing additional *term rewrites*.

Note first that no existing instance of (a subclass of) **Term** may be modified after its creation. Actually, we also followed this programming paradigm in the implementation of **Relations**. We named this paradigm *instance locking*, since any instance is, once created, protected against any alteration.

Consequently, if all existing instances are locked, any desired modification when translating some input stream into the representation of a **Term**, has to take place during the creation itself. A valuable side-effect of *instance locking* is that an instance of **Term** (and likewise of **Relation**) may have more than just one *owner*, that is, it may have numerous objects pointing to it, that do not have to bother about modifications.

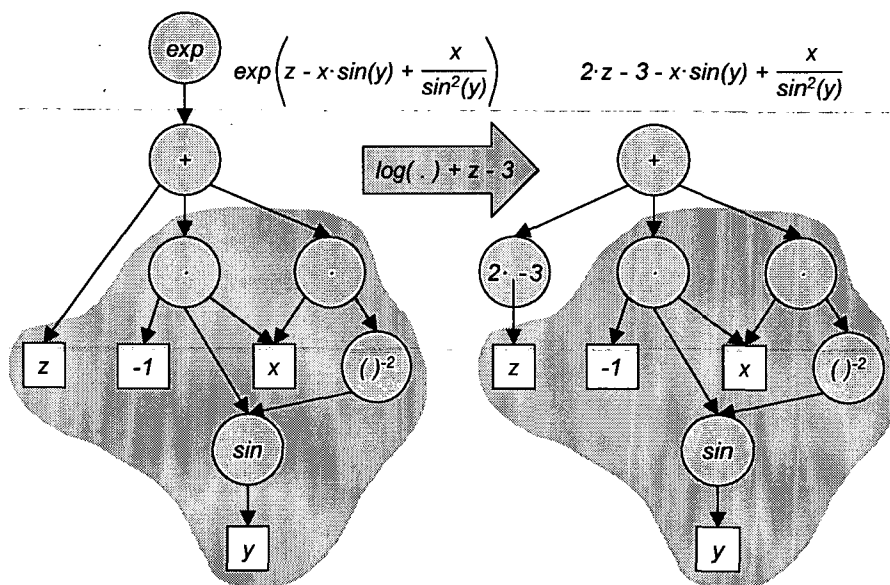


Figure 5.10: Building a Term by Maximal Reuse of Identical Subterms

In order to construct representations of more complicated arithmetic terms, we need to start with the terminal embodiments of **Term**, i.e. instances of the classes **Value** and **Variable**.

Given those, any non-terminal instance must be an n -ary expression of n subterms, $n \in \mathbb{N}_+$; cf. Fig. 5.9. The left-hand side graph of Fig. 5.10 depicts an example,

$$t \stackrel{\text{def}}{=} \exp \left(z - x \cdot \sin(y) + \frac{x}{\sin^2(y)} \right).$$

Note that the shown directed graph is connected and acyclic but not a tree. The reason for that is that there are coinciding subterms inside t , namely x and $\sin(y)$. Both have two owners, a situation that is not critical, due to *instance locking*, as explained above.

Figure 5.10 depicts the terminal **Value** and **Variable** instances as boxes and all non-terminal ones as labeled discs. A term is represented by a DAG with exactly one root, that is, a node without predecessors. Moreover, in the DAG, any succeeding node is the root of a sub-DAG representing a subterm.

Note that the subterms of a term are again instances of subclasses of **Term**. And so, *instance locking* implies, that a subterm can be reused as many times as we please. This idea is depicted in Fig. 5.10: The application of the function $u \mapsto \log(u) + z - 3$ produces the term $2 \cdot z - 3 - x \cdot \sin(y) + \frac{x}{\sin^2(y)}$, of which a representation is shown on the right-hand side. The dark grey parts can directly be reused, that is, those subterms are, in practice, going to be represented by the *same* instances.

When our prototype is asked to generate a representation \mathbf{t} of a term $t = f(t_1, t_2, \dots, t_n)$ from given representations $\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_n$ of its subterms $t_i, 1 \leq i \leq n$, it establishes a weak normal form, by observing the following rules:

1. If $n = 1$ and $t_1 = f^{-1}(s)$, for some term s , then the idea is to replace $t = f(t_1) = f(f^{-1}(s))$ simply by s . This simplification has also been utilised in Fig. 5.10, when applying \log .

But here, we must be careful when f^{-1} is defined only on a proper subset of \mathbb{R} . E.g. replacing $\sin(\arcsin(x))$ by x will abandon the implicit constraint $x \in [-1, 1]$; cf. Def. 21. So, in order not to lose information, the method that imposes the application of $f = \sin$ to $\arcsin(x)$, needs to take care of preserving the otherwise lost implicit constraint. For instance, in the context of solving the atomic constraint $\arcsin(x) = \arcsin(y)$ for the variable x , we shall obtain the equivalent constraint $x = y \wedge x \in [-1, 1]$ as opposed to the overestimation $x = y$.

2. A term $t_1 - t_2$ is replaced by $t_1 + ((-1) \cdot t_2)$ which removes the necessity of a special class for representing differences. Note that this will normally lead to numerous pointers to the instance representing -1 . Our prototype maintains three static instances representing the prominent values $-1, 0$ and 1 . Every time an instance of these is required, the implementation provides the respective static instance. This clearly eases the recognition of coinciding values, and thus the decision whether two instances represent the same arithmetic term. That latter question will also be the subject of Subsect. 5.5.4.
3. $t_1 \div t_2$ is replaced by $t_1 \cdot t_2^{-1}$. Analogous to the previous rule, this spares us an additional class for representing quotients.
4. In each **Sum**, there can be at most one linear addend which is to be found at the head of the list of subterms. Hence, all addends that are instances of **WeightedSum** will be combined to form one instance which is then sorted to the front. Figure 5.10 illustrates also that rule; regard the addends z and $z - 3$.

5. Likewise, there may be at most one factor in a **Product** that is a **Value**. Again, that value factor can be found at the head of the list of subterms. Apart from the orderings described in the previous and in this item, there is no ordering imposed on the sets of addends and factors.
6. Additional simplifications can be deployed to support the creation process for **Terms**. Obviously, by that, the imposed normal form will be altered. The below subsection on computer algebra explains how additional rewrites have been provided in our prototype. Those will e.g. force the prototype to omit a leading addend 0 in any **Sum** and a leading factor 1 in **Products**.

It should be mentioned that the normal form imposed by the above items 1.-5. is in fact rather weak, as it allows for the coexistence of very many instances that all represent the same term. For example, none of the rules will prevent us from constructing two distinct instances for the two terms $x \cdot y$ and $y \cdot x$ which a mathematician would clearly regard equal.

The purpose of a normal form in general is to map alternative mathematical representations of the *same* object to the *same* instance. In our case, a weaker form will allow for numerous mutually distinct instances that all represent the same arithmetic term. With such a form, instance creation itself becomes more straightforward and thus cheaper, but at the price of a more involved mechanism for deciding the equality of two terms. On the contrary, the latter task becomes easier when using a stronger normal form. But then the creation task itself is expected to become considerably harder.

5.5.4 Issues of Computer Algebra

With the last paragraph of the previous subsection we have come up for an interesting discussion that is closely related to *computer algebra*. Besides other subjects, as for instance symbolic differentiation, integration and the computation of symbolic determinantes, one important matter of this research field is the automated simplification of arithmetic terms, as also addressed above.

A good overview can be found in [14] which also mentions some well-established software systems for accomplishing the above tasks. Moreover, in order to explore important practical issues of software systems, exemplified for the representatives REDUCE, MACSYMA and DERIVE, the reader may also consult [52]. More such systems can easily be found via internet; follow the links [11] and [10] that also list MAPLE and MATHEMATICA, to name just a few more well-known systems.

The general goal of *computer algebra* is the development of efficient algorithms for nonnumerical mathematics. The outcome of those procedures are symbolic expressions as opposed to numerical approximations. In [63], this aspect is emphasised, and the simplification of terms is there captured by the section on *formula manipulation*.

In what follows, we stick to well-known ideas concerning the application of term simplification rules to a given representation \mathbf{t} of a mathematical term t . Each of those rules will replace some sub-DAG of a given representation by a (hopefully

simpler) alternative sub-DAG.

Figure 5.11 illustrates the general task for the left-hand side term of Fig. 5.10, after substituting $y = \frac{\pi}{2}$: The variable-free sub-DAG representing $\sin(\frac{\pi}{2})$ can be replaced by the leaf representing 1, and afterwards the entire argument of \exp by z .

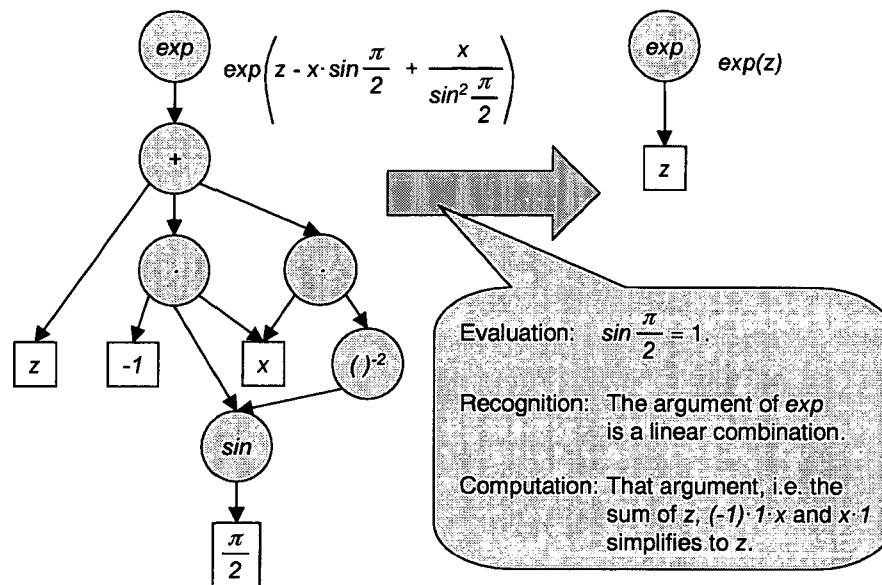


Figure 5.11: Simplification of the Term in Fig. 5.10 for $y = \frac{\pi}{2}$

Clearly, to this end, we need to know

- *how many* and *which* so-called *term rewrites* we would like to apply, and
- *whether* a sub-DAG matches a certain pattern for which one of our term rewrites applies.

As to the second item, the whole problem reveals some relationship to the field of *pattern matching*, e.g. in graphs. In this context we have implemented an appropriate method `matchesPattern`, see the following subsection.

The simplest form of pattern matching is syntactic equality. But due to the relative weakness of our normal form, our equality check for two instances `t1` and `t2` of `Term` goes beyond this default implementation. However, [63, page 304] states that it has been shown mathematically that there is no implementation which will always answer correctly whether `t1` and `t2` represent the same mathematical term.⁷

The first of the two above items, i.e. the one that deals with the set of applied rewrites, has been addressed in our prototype by the implementation of the class `Manipulator`. In it, a set of implemented rewrites is maintained; our prototype is

⁷Note also, that this fact implies, on the other hand, that there cannot be a *strongest* normal form for terms, that is, one that always maps *all* alternative mathematical representations of the *same* term to the *same* object inside an implementation.

capable of performing each of them. A user can pick the rewrites he actually wants to be performed (at any instantiation of a *Term*) from that pool of known rewrites. Hence, each rewrite can be switched on and off. Thereby, the user can adjust the symbolic computational power of our solver and accommodate to the requirements set by the respective engineering application.

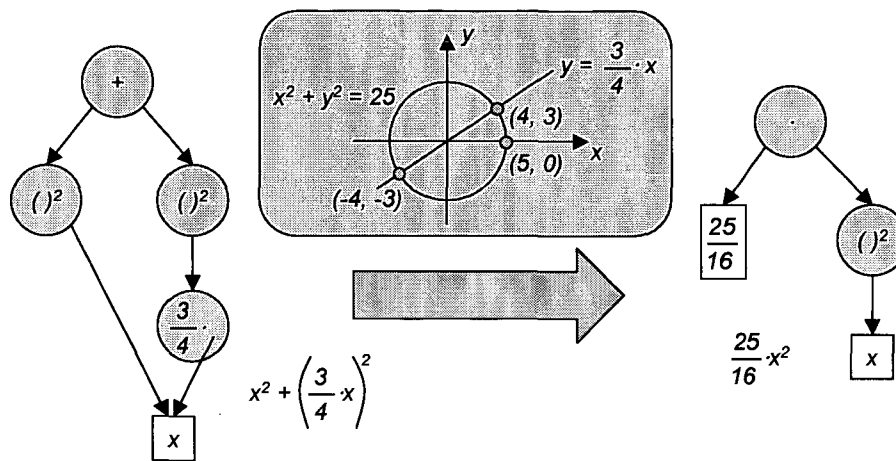


Figure 5.12: A Term Rewrite for the Reformulation of a Quadratic Term

Example 13: Suppose we would like to find all solution tuples (x, y) for which the quadratic system

$$y = \frac{3}{4} \cdot x \quad \wedge \quad x^2 + y^2 = 25$$

holds. The rectangular shape in Fig. 5.12 depicts the corresponding geometrical problem of a line with gradient $\frac{3}{4}$, intersecting a circle of radius 5 centred at the origin.

Our prototype solves the problem by substituting $\frac{3}{4} \cdot x$ for y in the quadratic circle equation which gives

$$(*) \quad x^2 + \left(\frac{3}{4} \cdot x\right)^2 = 25.$$

The left-hand side term of $(*)$ is depicted at the left of Fig. 5.12. In Subsect. 5.4.2, we stated that, in order to enable our prototype to solve $(*)$ for x , we must not have more than one occurrence of x which is however the case in $(*)$. Consequently Fig. 5.12 shows not a tree but a DAG in which x has two predecessors.

In this situation, the presented manipulation improves the situation since the resulting DAG is a tree, and our prototype is able to map $\frac{25}{16} \cdot x^2 = 25$ to the equivalent $x = -4 \vee x = 4$. So here, we need to switch on a certain set of term rewrites that accomplish the presented manipulation, in order to be able to derive the set of (x, y) -tuples, $\{(-4, -3), (4, 3)\}$.

Let us close this subsection by presenting some of the term rewrites that have been implemented in our prototypic implementation of RCS.

- Before $t_1 - t_2$ is replaced by $t_1 + (-1) \cdot t_2$, a rewrite can check whether t_2 is the **Value** 0. In this case t_1 is returned. Likewise $t - t$ may be replaced by 0.
- Similar rewrites exist for division: $t \div 1$ becomes t , $t \div (-1) = (-1) \cdot t$ and $t \div t = 1$. In the context of the latter, we must however preserve the implicit constraint $t \neq 0$.
- Several rewrites for addition can be activated: $0 + t = t + 0 = t$ and $t + t = 2 \cdot t$. Furthermore, if l_1, l_2 denote two linear terms, the addition of the **Sum** $l_1 + t$ with l_2 will be replaced by the result of $(l_1 + l_2) + t$, combining the two simpler terms first.
Whereas the previous rewrite deploys associativity, there are two rewrites for distributivity: $t + (p \cdot t)$ becomes $t \cdot (1 + p)$ and $(t \cdot p_1) + (t \cdot p_2) = t \cdot (p_1 + p_2)$, where p, p_1 and p_2 may also be more involved products.
- Besides trivial rewrites like $1 \cdot t = t$, $0 \cdot t = 0$ and $t \cdot t = t^2$, more rewrites have been implemented for collecting common factors in two products that are to be multiplied. Moreover, for a term t that is not identically 0, we may replace $x^{-1} \cdot (a \cdot x + t)$ by $a + t \cdot x^{-1}$, without discarding any implicit constraint. Note that this rewrite will decrease the number of occurrences of x by 1, without affecting any other number of occurrences.
- The obvious rewrites for powers with exponents equal to 0 or 1 are complemented by $(t^d)^e = t^{d \cdot e}$ and $(t_1 \cdot t_2 \cdot \dots)^d = t_1^d \cdot t_2^d \cdot \dots$.

Regarding once more Ex. 13, we can now name the rewrites that we need to switch on, in order to cover the manipulation shown in Fig. 5.12, i.e. $x^2 + (\frac{3}{4} \cdot x)^2 = \frac{25}{16} \cdot x^2$. First, the last of the above listed rewrites yields $x^2 + (\frac{3}{4} \cdot x)^2 = x^2 + (\frac{3}{4})^2 \cdot x^2 = x^2 + \frac{9}{16} \cdot x^2$. Next, we need to perform a distributive rewrite for isolating x^2 . This will finally rewrite $x^2 + \frac{9}{16} \cdot x^2$ as $(1 + \frac{9}{16}) \cdot x^2 = \frac{25}{16} \cdot x^2$.

The example illustrates that sometimes, distinct rewrites may apply one after another. Our rewrite scheme implemented at **Manipulator** is thus recursive. Moreover, there exists an ordering, according to which rewrites are tried. This ensures determinism in cases where more than one term rewrite is applicable.

5.5.5 Operations on Arithmetic Terms

This subsection will provide brief explanations and some additional remarks concerning the methods implemented for all **Terms**, as listed in Fig. 5.9.

We shall start with the more obvious arithmetic operations that are listed at the bottom of the box named **Term**, that is, the methods **plus**, **minus**, ..., **exp**.

Whenever one of these is applied to some instance(s) of **Value**, our implementation in the value module will return the appropriate evaluation, again as an instance of **Value**. Otherwise, a new term will be created; and during this process, our prototype

may apply several term rewrites, in order to simplify the result. This has been discussed in detail in the previous subsection and illustrated in Figs. 5.10, 5.11 and 5.12.

solvedFor and **substitute** have already been identified as the interface to higher-level algorithms as elaborated in Subsect. 5.4.2. **solvedFor** is, in that context, responsible for isolating a certain “goal” variable in an atomic constraint. This is basically done by applying inverse operators to both sides of the constraint, cf. Ex. 11. The implementation of **substitute** is straightforward, taking into account that we are to replace a variable, i.e. a leaf in a term DAG, by some other DAG. The only point worth mentioning is that, after the naive replacement, some term rewrites may apply, as exemplified in Fig. 5.11, where y has been substituted by a **Value**.

The methods **occurrencesOf**, **matchesPattern** and **equals** have to be seen in connection with the application of term rewrites.

The first method in this group simply returns, for each variable in some term DAG, how many predecessors it has. **occurrencesOf** is used by **matchesPattern**. Moreover, it is to inform **solvedFor** whether the “goal” variable has indeed only one occurrence, and whether the solver is hence able to make the reformulation; see the remarks preceeding Ex. 11 in Subsect. 5.4.2

matchesPattern is a method that answers, for a given **Term**, whether this has a certain structure, so that we can apply a given term rewrite. Clearly, this test will often also have to investigate some or even all subterms of the instance in question. Therefore, the implementation is recursive.

The recursion ends, in general, with an invocation of **equals** to check for the equality of two instances of **Term**. The default for **equals** is object identity. Additionally, for two **Sums** or **Products**, that predicate will also check the coincidence of addends or factors, respectively. This means that it will, for instance, discover that $a + b + c$ and $b + c + a$ have permuted lists of addends, and eventually consider them equal. The parametrisation of **matchesPattern** is not trivial and has been omitted in Fig. 5.9.⁸

As to the method **getValue**, suppose we are given a term t and its representaion \mathbf{t} . Then, $\mathbf{t}.\mathbf{getValue}()$ provides us with a subset of \mathbb{R} that contains all evaluations of t , in the sense of Def. 21. For example for $\sin(x)$, where $\text{dom}(x) = \mathbb{R}$, we obtain the interval $[-1, 1]$. In other words, **getValue** collects all possible values that the given term may take.

So far, we have failed to give a good motivation for the method, and the purpose of **getValue** shall become clear only in the following subsection. There, it is mainly used to simplify terms that have, due to complicated computations, become too deep, that is, for which the method **depth** returns an unacceptably large integer.

⁸Still, it should be mentioned that the method will, besides returning a boolean, provide further useful information by filling / overriding some of its arguments. That information encodes paths in the given term DAG leading to the subgraphs which may be replaced by the term rewrite in question. With that additional knowledge, the subsequent application of the rewrite itself becomes straightforward.

Let us thus, for infrastructural reasons, define the *depth* of a term DAG, as returned by **depth**: With the intuition already given, an informal definition is the longest possible path (counting edges) in a DAG from the root to a leaf. So, the *depth* of the left-hand side term in Fig. 5.10 equals 5, and is realised by the path that descends, starting at the root, always into the right-most successor.

Formally, any instance of **Value** and **Variable** is assigned a *depth* of 0. And for an expression $t = f(t_1, t_2, \dots, t_n), n \in \mathbb{N}_+$, we simply define

$$\text{depth}(t) \stackrel{\text{def}}{=} 1 + \max\{\text{depth}(t_i) \mid 1 \leq i \leq n\}.$$

5.5.6 Interval Algebra

In Subsect. 2.3.5 an important requirement for constraint solving in the context of engineering tasks was noted, that addresses uncertain or erroneous data.

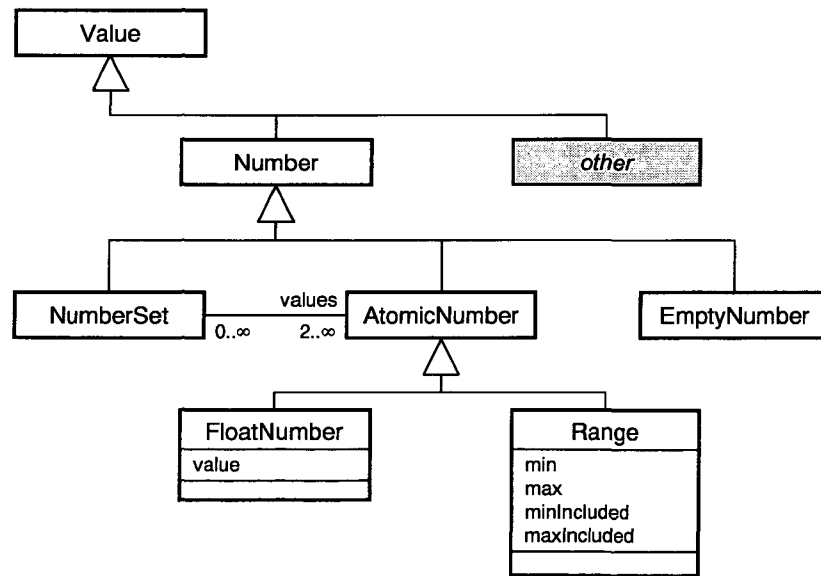


Figure 5.13: UML Diagram of Value Classes for Representing Subsets of \mathbb{R}

For the set of real numbers, \mathbb{R} , vague knowledge about coefficients or a component's parameters can be captured by means of intervals: An extra constraint may state that a variable be contained in some interval, as opposed to be fixed to a certain value $r \in \mathbb{R}$. We have also already seen an example for a constraint problem with uncertain coefficients in Subsect. 2.4.2, where the problem R_k was modified so that each resistance $R_j B_i$ lie in the interval $[100 \cdot j - \Delta, 100 \cdot j + \Delta]$, for some small $\Delta \in \mathbb{R}$.

In order to account for the requirement to be able to represent and process intervals of real numbers, our prototype facilitates a special subclass of **Value**, **Range**. That is depicted in Fig. 5.13, along with the second specialisation of **AtomicNumber**,

namely **FloatNumber**.

A **Range** has two attributes **min** and **max** for storing the boundary points of the interval, and two boolean flags for storing whether those belong to the represented subset of \mathbb{R} .⁹ On the other hand, a **FloatNumber** has one attribute **value** that represents a real number. Note however, that any **FloatNumber** represents a singleton set $\{r\} \subset \mathbb{R}$, for some $r \in \mathbb{R}$, instead of the real number r itself.

So we should note once more, that each instance of (a terminal subclass of) **Value** stands for a *subset* of \mathbb{R} , as opposed to an *element* of \mathbb{R} . Consequently, Fig. 5.13 mentions the class **EmptyNumber**, the sole instance of which represents the empty set of real numbers, $\emptyset \subset \mathbb{R}$.

Two or more **AtomicNumbers** may form an instance of **NumberSet**, which stands for the union of the sets represented by the embedded **Ranges** and **FloatNumbers**. An example is the set $(-\infty, -2) \cup \{0\} \cup (1, 7]$. As is then clearly possible, our implementation enforces the embedded subsets to be mutually disjoint and that no two of them union to a single interval. That is, an attempt to instantiate a **NumberSet** representing $(0, 1) \cup \{1\}$ will result in the instance of **Range** that captures the interval $(0, 1]$.

Fig. 5.13 suggests that our implementation supports, besides subclasses of **Number**, other classes that deal e.g. with finite domains of symbols and will become clearer in the final section of this chapter.

With what has just been noted, we actually need to revise e.g. the DAGs in Fig. 5.12, in which the reals $3/4$ and $25/16$ should now be replaced by the singleton sets $\{3/4\}$ and $\{25/16\}$, respectively. (Still, we may regard $3/4$ as an unambiguous shorthand for $\{3/4\}$.)

However, the question remains, what the semantic of the arithmetic operations outlined in the above Subsect. 5.5.5 is, e.g. when we apply **plus** to two subsets of \mathbb{R} , represented by **Numbers**. That semantic is easily (re-)established by *interval algebra*, or often also called *interval arithmetic*. The following definition resembles [72, Def. 6.2, p. 112], that defines the term *set extension* for functions and relations on \mathbb{R}^n .

Definition 23 (Interval Algebra)

Let $n \in \mathbb{N}_+$, $D_i \subseteq \mathbb{R}$, for $1 \leq i \leq n$, and $f : D_1 \times D_2 \times \cdots \times D_n \longrightarrow \mathbb{R}$ be a real-valued function that is defined on a subset of \mathbb{R}^n .

With $\mathcal{P}(M)$ (once more) denoting the powerset of M , we can then define an **extension function**, \bar{f} , for f , according to

$$\begin{aligned} \bar{f} : \mathcal{P}(D_1) \times \mathcal{P}(D_2) \times \cdots \times \mathcal{P}(D_n) &\longrightarrow \mathcal{P}(\mathbb{R}), \\ (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n) &\longmapsto \{f(x_1, x_2, \dots, x_n) \mid \\ &\quad \forall i \in \{1, 2, \dots, n\} \ x_i \in \bar{x}_i\}. \end{aligned}$$

The extension functions for the basic arithmetic operators $+, -, \cdot : \mathbb{R}^2 \longrightarrow \mathbb{R}$ and $\div : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \longrightarrow \mathbb{R}$, induce the so-called **interval algebra**, for combining two intervals in \mathbb{R} .

⁹Via exceptional assignments to **min** and **max**, respectively, **Range** also covers representations of intervals that are not bounded below or above or that coincide with $(-\infty, \infty) = \mathbb{R}$.

We may hence invoke `plus`, as implemented by our prototype, for the two `Numbers` that stand for $[1, 2] \cup \{3\}$ and $(4, 5]$, to obtain a representation of

$$[1, 2] \cup \{3\} + (4, 5] = (1 + 4, 2 + 5] \cup (3 + 4, 3 + 5] = (5, 7] \cup (7, 8] = (5, 8].^{10}$$

Note that the invocation of an arithmetic operation for two `FloatNumbers` will return the appropriate `FloatNumber`, and recovers thus the *ordinary* arithmetic operation. In other words, we are able to recover the original function f from \bar{f} .

In Def. 21, we defined the usual shorthand for arithmetic constraints, and clarified thus, for example, the meaning of the constraint $3 \cdot x + y = 5$, by mapping it back to Def. 2.

Now, we can go even one step further and extend this notation in a natural way, by allowing for subsets of \mathbb{R} instead of real numbers:¹¹

Definition 24 (Extended Arithmetic Constraint)

Let c denote an arbitrary and-or-junction of arithmetic atomic constraints. Suppose c mentions the finite set of real numbers $\{q_1, q_2, \dots, q_n\}, n \in \mathbb{N}_+$. By replacing each q_i in c by some subset $\bar{q}_i \subseteq \mathbb{R}$, we obtain an **extended arithmetic constraint** \bar{c} , defined in the following way: Choose for each q_i a new variable x_i with $\text{dom}(x_i) \stackrel{\text{def}}{=} \bar{q}_i$. Then replace in c each q_i by x_i . With \hat{c} denoting the result of all those replacements, we let

$$\bar{c} \stackrel{\text{def}}{=} \pi_{\text{vars}(c)}(\hat{c}).$$

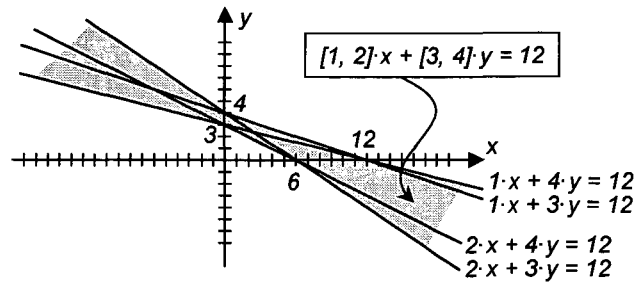


Figure 5.14: An Extended Arithmetic Constraint that is a Linear Band

Example 14: Figure 5.14 depicts an example of an extended arithmetic constraint that arises by replacing the real coefficients 1 and 4 in $1 \cdot x + 4 \cdot y = 12$ by the intervals $[1, 2]$ and $[3, 4]$. Note that the set of (x, y) -tuples allowed by the constraint, form no longer a convex set as usual for linear constraints. Moreover, writing down an equivalent *ordinary* arithmetic constraint with the solution area as given in grey,

¹⁰Obviously, $+$ is monotonous in both arguments which yields the demonstrated method for computing the sum of two `Numbers`. There are similar well-known calculation rules for $-$, \cdot and \div ; see e.g. [40]

¹¹Note again the similarity to the notion of *natural interval extension* in the framework of Numerica, see [72, Def. 6.5, p. 113].

will be a rather involved and-or-junction of the four presented helper constraints and the case-differentiating constraints $x \leq 0, x \geq 0, y \leq 0$ and $y \geq 0$.

After that little excursus, let us return to our implementation of RCS. We shall now exemplify a typical problem arising from interval algebra. After that, it will be shown how that problem has been addressed in our prototype.

Example 15: Figure 5.15 shows an aggregation tree that solves the linear constraint problem

$$\{x = z, \quad x = y, \quad y = [0, 2], \quad z = 2 - y\}.$$

Note that we have four constraints for three variables, all of which have to equal 1. Indeed, the problem would be solved in a correct manner, if the uncertainty constraint $y = [0, 2]$ was discarded.

Consider first the white nodes and the white annotation to the root node. The latter contains the forward join at the root node, as it results from preceeding computations. The reason for the overestimation is the aggregation of $x = y \wedge y = [0, 2]$ with $z = 2 - y$. Here, following the algorithm in Figs. 5.7 and 5.8, y must be made basic in order to be eliminated. in Figs. 5.7 and 5.8.

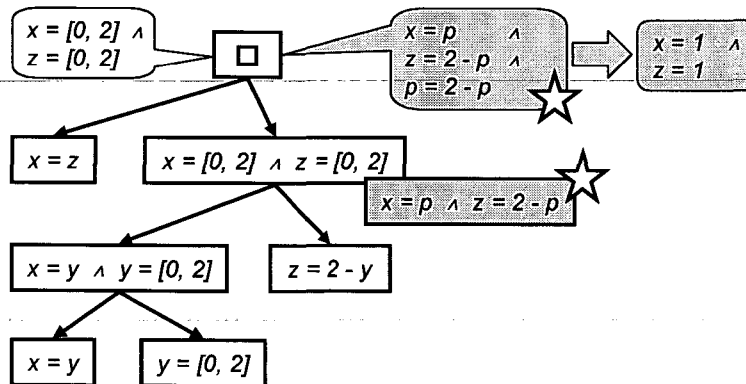


Figure 5.15: A Typical Problem with Interval Algebra

There exists one atom in solved form for y , namely $y = [0, 2]$. Thus, y is going to be substituted by $[0, 2]$, and the correlation $z = 2 - x$, implied by $z = 2 - y \wedge x = y$, will be lost.

The problem can easily be fixed by introducing a parameter p with a domain of $[0, 2]$. The aggregation in question will be replaced by the grey node, this time preserving the constraint $z = 2 - x$. Note that this node's forward relation is just a rewritten version of $x = y \wedge z = 2 - y$, where y has been replaced by a *hidden* variable p of which no class outside the term module will take notice.

At the root node, p will be found to be 1, and thus x and z , too; see grey annotation.

How are we going to represent a hidden variable in our prototype? As suggested by Fig. 5.9, we may instantiate a `Variable` for which the method `isHidden` yields

true. Those instances will serve as parameters in the sense of the above example.

The problem we encountered in Ex. 15 can even be further simplified. When two intervals $I = [1, 2]$ and $J = [3, 5]$ are added, and J is later to be subtracted from the result, we obtain the naive evaluation

$$(I + J) - J = ([1, 2] + [3, 5]) - [3, 5] = [4, 7] - [3, 5] = [4 - 5, 7 - 3] = [-1, 4] \supset I,$$

whereas a symbolic computation, e.g. aided by hidden variables, yields again I . So, with naive interval algebra, we lose information, and our framework tends to produce unreasonable overestimations of the exact constraints.

Replacing uncertain coefficients represented by `Range` or `NumberSet`, by hidden `Variables` is therefore our means to increase computational accuracy, in the sense that overestimations are, as far as possible, avoided.

There is however a sweeping drawback: When a term mentions a hidden variable, it will be treated as any non-`Value` term, i.e. it is subject to term rewrites. Also, for large problems with many uncertain coefficients where we can only seldom apply the activated term rewrites, new instances of `Term` tend to become very *deep*, that is, have a large depth as returned by the method `depth` introduced in the previous subsection. This will inevitably have a negative impact on runtimes.

Consequently, our implementation enables the user to set a *maximal term depth*, m : Whenever a term is about to be instantiated for which this threshold is exceeded, then the respective DAG will be simplified. To this end, we identify all sub-DAGs that are rooted in a distance of m from the DAG's root, and replace them by their evaluations obtained with `getValue`; see again the previous subsection.

The adjustable maximal term depth is an important facility of our prototype that enables the user to trade computational accuracy for runtime.

The trade-off can be studied with the help of Fig. 5.16. All of a, b, c, d are hidden variables. The top left DAG represents the term $a + (b/a)$ and has a depth of 3: The fat path is the longest one with 3 edges. The top right DAG will be generated from the left one, when we enforce a maximal term depth of 2; c denotes a new hidden variable with the domain $1 \div \text{dom}(a)$. We have obviously already lost information, for the DAG has now become a tree.

The left column shows the effect of adding the middle tree of depth 2, that represents $-(b/a)$. On the right-hand side, the same computation is shown for a maximal depth of 2. On the left, term rewrites may simplify the temporary instance to the final result a , which is correct. Contrariwise, on the right, the temporary result of `sum` is again a DAG with depth 3; note the fat path. Due to term rewrites, b may - as on the left-hand side - be factored out, still leaving our prototype with a DAG of depth 3. Decrementing the depth is then done by introducing the new hidden variable d with the domain $\text{dom}(c) - (1/\text{dom}(a))$. Note that the right computation flow is not aware of $d \equiv 0$, and by decreasing the depth, we have sacrificed computational accuracy and potentially produced overestimating constraints.

On the other hand, in order to deduce the result a in the left column, we need to apply more rewrites than on the right, which shows that here, the computation will in general be more expensive.

We have elaborated the idea of hidden variables, in order to increase the computational accuracy of relational aggregation and to overcome the natural shortcomings of interval algebra. Those hidden variables fit obviously nicely in the concept of term DAGs and their manipulation via term rewrites.

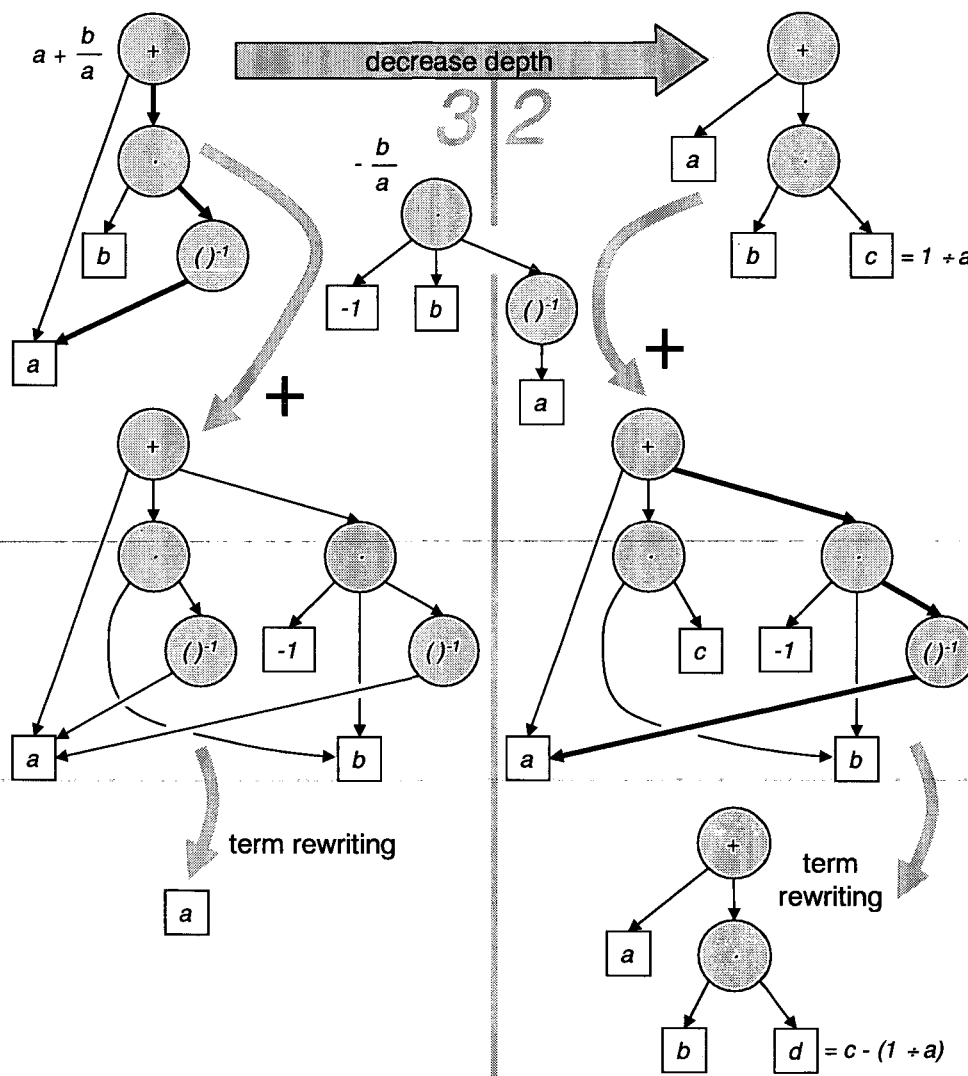


Figure 5.16: The Relationship between Term Depth and Overestimation

Let us consider hidden terms, that is, DAGs with leaves consisting exclusively of **Values** and hidden **Variables**: When the forward phase reaches the last aggregation, i.e. the top-most node in an aggregation tree, we no longer need hidden terms and they can hence be evaluated using `getValue`. Therefore, even when the user did not impose a maximal term depth, hidden terms will eventually need to be reduced to **Values**.

Let us thus finish our investigations on interval algebra by going further into the following question: Is there a sufficient condition that guarantees the naive evaluation of a hidden term (by means of interval algebra) to produce the tightest **Value** and not just some overestimation?

The next lemma shows that the answer is “yes”:

Lemma 12

Let $t(x_1, x_2, \dots, x_n)$ be a rational term¹² in the variables $x_i, 1 \leq i \leq n \in \mathbb{N}_+$, where each $\text{dom}(x_i)$ is a bounded, closed interval in \mathbb{R} . Suppose furthermore that the DAG for t is a tree, i.e. each x_i appears exactly once in t .

Then, t takes its extremal values at some corners of the cuboid $K \stackrel{\text{def}}{=} \text{dom}(x_1) \times \text{dom}(x_2) \times \dots \times \text{dom}(x_n)$. Furthermore, naive interval algebra will return the set of all values that t takes on K , and no more.

The assumption concerning the structure of $\text{dom}(x_i)$ is made only for the sake of simplicity. It is fairly easy to convince oneself that the lemma can be “lifted” to arbitrary bounded subsets $\text{dom}(x_i) \subset \mathbb{R}$ which can be represented by an instance of **FloatNumber**, **Range** or **NumberSet**.

It is also easy to give examples of rational terms in which at least one hidden variable appears more than once, and for which the lemma’s conclusion does not hold:

Example 16: Considering again the term $a + (b/a)$ of Fig. 5.16, we observe for $a, b > 0$:

$$a + \frac{b}{a} = \left(\sqrt{a} - \sqrt{\frac{b}{a}} \right)^2 + 2 \cdot \sqrt{b} \geq 2 \cdot \sqrt{b}.$$

Also, the estimate is sharp, that is, we obtain equality for $a = \sqrt{b}$. Thus, for $b > 0$ fixed, $a + (b/a)$ takes its minimum at $a = \sqrt{b}$. Consequently, the minimum is in general not taken in the corner of $\text{dom}(a) \times \text{dom}(b)$.

Clearly, the lemma’s precondition is rather restrictive and will in general not apply for practical applications. But its proof uses a monotonicity argument, and we certainly have cases, in which all our hidden terms are known to be monotonous in each argument: For resistive networks with uncertain (Ohmic) resistances r_1, r_2, \dots, r_n , $n \in \mathbb{N}_+$, as e.g. derived from R_k in Subsect. 2.4.2, it can be shown that all unknowns, that is, all currents and voltages, are monotonous in each resistance r_i . That means that, if we fix all but one uncertain resistance r_k , then all currents as well as voltages will monotonously change provided r_k is monotonously changed.

So here, each hidden term $t(r_1, r_2, \dots, r_n)$ will, due to this strong monotonicity, obviously also take its minimum m and maximum M in some corners of the respective domain cuboid $K = \text{dom}(r_1) \times \text{dom}(r_2) \times \dots \times \text{dom}(r_n)$.

In order to find m and M , we just have to identify the right corners of K and evaluate t there. This identification can be efficiently done in time $O(n)$:

¹²This is to be understood as fixed by Def. 21 and includes thus that t be well-defined.

1. Start at an arbitrary corner of K and evaluate t there. (Note that this evaluation, as well as all following ones, are evaluations of t for n instances of `FloatNumber`.)
2. For $1 \leq i \leq n$ do: Move along the edge of K that corresponds to r_i , to the neighbouring corner of K and evaluate t there. This changes exactly the i^{th} argument `FloatNumber` for which t is evaluated. Afterwards, we will know whether t increases or decreases, and thus what kind of monotonicity we have along the r_i -edge of K .
3. After the loop, we have identified, for each i , that end of the r_i -edge, at which t becomes smaller and that at which it becomes larger. In other words, we can directly move to the two corners at which t takes the values m and M .

The outlined algorithm has been implemented in our prototype. It will be used for the evaluation of hidden terms, whenever the user sets a special static *monotonicity flag* situated in the class `AggController`, see Fig. 4.1.

We have already mentioned [40] which deals with interval constraint reasoning and addresses the inaccuracy of naive interval algebra. [40, Sect. 4] also points out the drawback of multiple occurrences of interval parameters in arithmetic terms; cf. precondition of Lem. 12.

[60] emphasises that, in general, we need to complement symbolic solving techniques - as in our prototype - with numerical interval narrowing techniques. The latter have in our implementation so far been omitted, accepting overestimating constraints in more involved applications. However, [65] illustrates how far we can get with purely symbolic techniques in the case of linear constraint problems arising from electrical applications. Starting from the general *Gauss elimination method*; see also [6, pp. 148-159]; it moves to more sophisticated methods that can be applied when the coefficient matrix is sparsely filled with non-zeros. Note that again, this relates to the *low density assumption* which we deployed especially in Chap. 4, to obtain moderate complexity figures.

In [66], Smit takes up the property of sparseness of the coefficient matrix for resistive network problems. This publication elaborates a way to obtain procedures for efficiently computing symbolic terms for unknowns, that identify common symbolic subterms.

Other very useful literature pointers to thorough introductions to the matters of interval computations and finding extremal values of functions (on bounded n -dimensional cuboids), are [51] and [61], respectively.

5.6 Constraint Language and Control Aspects

In this section, we shall give an overview over the constraint language supported by our prototypic implementation of RCS. Before this issue is addressed in full detail, our facilities for controlling the prototype will be outlined. Those enable the user to choose particular settings so as to trade, e.g., runtime for accuracy, that is, the degree of overestimation.

5.6.1 Control Facilities

We give a complete list of all adjustable control parameters implemented so far. All those static parameters reside in the class `AggController`; see Fig. 4.1; and will be initialised with default settings, at any start-up of the system. There are methods that enable the user to alter each parameter on-the-fly. The parts of our implementation that need to pay attention to the settings, will retrieve the respective parameter(s) via accessor methods, directly from `AggController`.

- **Digits:** Any computer provides only finitely many so-called machine numbers for the approximation of reals. The complex issue of *floating point arithmetic* has been paid attention to ever since processes have been automatised using scientific calculators. The prominent *box-consistency* approach has been realised in many solvers, see e.g. [72].

On the one hand, the finite pool of machine numbers enforces a certain classification of \mathbb{R} . On the other hand, the finite mantissa of machine numbers results in rounding errors: A computation might yield a machine number that is not the best approximation of the exact result.

Note that this phenomenon is likely to introduce conflicts where there are in fact none: Suppose t_1 and t_2 are alternative representations of the same mathematical term that evaluate, due to rounding errors, to two distinct machine numbers m_1, m_2 . Then the consistent constraint problem $\{x = t_1, x = t_2\}$ is going to be judged inconsistent because $m_1 \neq m_2$.

Our prototype is to pronounce inconsistency only when the problem has indeed no solution. Therefore, we need to implement a notion of tolerance that regards m_1 and m_2 as equal, as long as they differ by less than some pre-defined, small $\epsilon > 0$.

Our control parameter *digits* maintains an integer $d \geq 1$, and 10^{-d} will basically be used as that ϵ for all comparisons of machine numbers m_1 and m_2 . Obviously, d is the number of leading digits that need to coincide in the mantissas of m_1 and m_2 .

Clearly, any ϵ may eventually be exceeded by an unfortunate chain of rounding errors. And so, *digits* is in fact an application-specific setting.

- **Inequalities:** A user of our prototype is given different ways to represent an inequality $t_1 \prec t_2$, where t_1, t_2 are arbitrary real-valued terms and $\prec \in \{<, \leq, >, \geq\}$. He may use the natural one-to-one representation that maintains pointers to the two terms and the static instance of `RelationType` that stands for the binary relation \prec ; see Fig. 5.1.

Secondly, he can enforce the translation into an equation with a *slack parameter*: For example, in the case of $t_1 < t_2$, we know that $t_2 - t_1$ must be positive and can hence write equivalently $t_2 - t_1 = s$, where s is going to be a hidden `Variable` with the domain $\text{dom}(s) = (0, \infty)$. Compared to the first, natural representation, this way of writing will eliminate all inequalities. Consequently, our prototype will never have to deploy Fourier's algorithm but pure Gauss-like substitutions instead. Knowing that Fourier's algorithm is likely to introduce vast numbers of (redundant) inequalities during each elimination

step, that approach seems appealing at first sight. However, the drawback is that we are likely to obtain lots of constraints with hidden variables only, due to our substitution scheme. Those are however only poorly treated by our prototype, and often immediately discarded, when they do not immediately provide a tight restriction for some visible variable.

A third possibility is to represent the above inequality $t_1 < t_2$ by the extended arithmetic constraint $t_2 - t_1 = (0, \infty)$, i.e. even omitting the slack parameter. The intention here is to enforce arithmetic terms with a comparatively small depth to speed up the solving process. Clearly, this comes at the cost of a reduced accuracy, that is, we are in general going to get looser restrictions for the unknowns.

- **Disequations:** There exists a static instance of `RelationType` that captures the binary relation \neq . Therefore, a user may choose the natural representation of a disequation.

As in the previous item, we may - for real-valued terms t_1, t_2 - represent $t_1 \neq t_2$ by the equivalent equation $t_2 - t_1 = s$, where this time $\text{dom}(s) = (-\infty, 0) \cup (0, \infty)$ is represented by a `NumberSet`. Again, this will remove all disequations from a given problem, but introduce the outlined difficulties with hidden constraints, i.e. constraints over hidden variables. Note that the slack parameters' domains cover \mathbb{R} almost entirely, and so these hidden constraints will only seldom provide a tight restriction for some term. As a result, our prototype will almost always simply discard the hidden constraints, taking poor accuracy into account.

As above, $t_1 \neq t_2$ may, for real-valued terms, be represented as the extended arithmetic equation $t_2 - t_1 = (-\infty, 0) \cup (0, \infty)$. The consequences are similar to those described above.

- **Reuse of AggForest:** In Subsect. 3.4.2, we developed methods for deploying the reuse of previous computations. However, along large sequences of similar constraint problems, the repaired aggregation trees tend to become unbalanced or otherwise disadvantageous for any further reuse. In that case, it is often more advisable, to abandon the previous aggregation forest and analyse the next context from scratch.

Whereas the default setting for reuse is unconditioned reuse, all implemented reuse facilities can just as well be turned off. A third, heuristic setting captures a conditioned reuse: Depending on the *quality* of the underlying aggregation forest, as computed in the previous context, reuse of reusable subtrees will automatically be turned on or off.

Such *quality* parameters are, e.g., the variance of root-leaf path lengths and the complexity of forward relations of non-leaf nodes. That complexity, in turn, is a measurement which combines the number of disjuncts and the average number of conjuncts per disjunct of a given constraint in DNF.

- **Term Depth:** The previous Sect. 5.5 discussed term DAGs, as used as the representation of arithmetic terms in our prototypic implementation of RCS. Also, Fig. 5.16 illustrates the relationship between term depth and computa-

tional accuracy: The higher the maximal term depth is, the more accurate will the computations be. That is, fewer relations are going to be approximated by overestimations. The user can directly set that maximal term depth. By choosing any negative integer as maximal depth, he allows for arbitrarily deep term DAGs.

- **Separation of Values:** This control option may have considerable effect in the context of problems that involve a large number of sets in \mathbb{R} which can be represented by **NumberSets**. As we have seen, those sets are unions of singleton sets $\{r\}$, where $r \in \mathbb{R}$, and of real intervals.

Here, the user can choose to allow for the instantiation of appropriate instances of **NumberSet**, or suppress that instantiation in favour of disjunctive constraints.

To make that clear, consider the constraint that forces x to take its values in $\{3\} \cup (4, 5]$. Then, this may be represented by the extended arithmetic constraint $x = \{3\} \cup (4, 5]$ or by the disjunction $x = 3 \vee x = (4, 5]$.¹³ In the former case, we allow for instances of **NumberSet**, in the latter we do not.

Due to our Gauss-like substitution scheme, the former way of writing will in general imply less accurate results. Suppose for example the computation of the join $x^2 = 9 \bowtie x \cdot y \geq 0$ when allowing for **NumberSets**:

$$\begin{aligned} x^2 = 9 \bowtie x \cdot y \geq 0 &\sim x = \{-3, 3\} \bowtie x \cdot y \geq 0 \\ &\rightarrow x = \{-3, 3\} \bowtie \{-3, 3\} \cdot y \geq 0 \\ &\sim x = \{-3, 3\} \bowtie (-3 \cdot y \geq 0 \vee 3 \cdot y \geq 0) \\ &\sim x = \{-3, 3\} \bowtie (y \leq 0 \vee y \geq 0). \end{aligned}$$

Note that the implication is not an equivalence, since the surplus solutions $(x, y) = (3, -r)$ and $(-3, r)$, for $r > 0$, are introduced. However, this is what happens inside our prototype which just follows its substitution scheme.

When suppressing instances of **NumberSet**, we obtain - for this example - the correct join:

$$\begin{aligned} x^2 = 9 \bowtie x \cdot y \geq 0 &\sim (x = -3 \vee x = 3) \bowtie x \cdot y \geq 0 \\ &\sim (x = -3 \wedge x \cdot y \geq 0) \vee (x = 3 \wedge x \cdot y \geq 0) \\ &\sim (x = -3 \wedge -3 \cdot y \geq 0) \vee (x = 3 \wedge 3 \cdot y \geq 0) \\ &\sim (x = -3 \wedge y \leq 0) \vee (x = 3 \wedge y \geq 0). \end{aligned}$$

Generally speaking, when allowing for **NumberSets**, the prototype tends to lose correlations between certain variables and introduce the corresponding symmetric solution tuples, as exemplified. By setting this control parameter to suppress **NumberSets**, that source of inaccuracy can be removed. The price is the introduction of additional disjunctions and thereby more involved joins and projections.

The presented six control options can be changed on-the-fly. Thus, alternative representations of **Values**, **Terms** and **Relations** may coexist and can be combined

¹³There are, of course, more possible representations involving hidden variables.

without trouble.

We shall now come to a concise overview over the constraint language “understood” by our prototype.

5.6.2 The Constraint Language of our Prototypic Implementation of RCS

In what follows, almost all language features of the current version of our implementation are covered. Some features have been omitted for the sake of simplicity. The corresponding implementational parts can be considered to be in an advanced state of development.

There are also other language ingredients that focus on additional aspects of our prototype which have only little to do with what has been formalised and explained in previous chapters and sections. Therefore we shall also not address those features here.

Our implementation of RCS has been complemented with a parser for strings of a special-purpose instance of the *extensible markup language (XML)*; for an introduction see e.g. [33] or [74]. The strings represent values, variables, terms, relations and other RCS-related objects. Their correct interpretation is due to a so-called *document type definition (DTD)*. The DTD we are using has been developed in the context of the development of RCS.

Using that XML interface, entire constraint problems, aggregation strategies and context spaces can easily be imported by our prototype. We remark that our implementation also provides, for each relevant object, a method that returns an XML string representation which observes the underlying DTD. Thereby, the prototype is fit not only for importing but also for exporting XML representations of RCS-relevant objects.

We shall neither describe here the structure of the XML files, nor the DTD. For exhaustive examples of XML strings, see App. B. Instead, an informal description of the language will be given that allows for an immediate understanding of which objects can be fed into and processed by our prototype.

Values & Variables

The following enumeration presents all values that the prototype can handle. Actually, *not values are handled but sets of values*. Note that this is a consequence of our concept of extended (arithmetic) constraints, as defined by Def. 24, which is based upon extension functions, see Def. 23. We shall see below that a similar lift from values to sets of values can be accomplished for constraints involving finite domain variables.

Let, in what follows, \mathbb{M} denote the set of machine numbers in \mathbb{R} facilitated by the underlying hard- and software.

Likewise, be \mathbb{S} the set of all symbols. Those are going to be used as the elements of

finite sets of mutually distinct discrete values, as e.g. in the set $\{red, blue, green\}$ of Ex. 2.

1. **float:** We can generate a representation of the singleton set

$$\{m\}, \quad \text{where } m \in \mathbb{M}.$$

2. **interval:** For any two machine numbers $m_1, m_2 \in \mathbb{M}$ with $m_1 < m_2$, we can express the machine number intervals

$$\begin{aligned} (m_1, m_2)_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 < m < m_2\}, \\ [m_1, m_2)_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 \leq m < m_2\}, \\ (m_1, m_2]_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 < m \leq m_2\}, \\ [m_1, m_2]_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 \leq m \leq m_2\}, \\ (-\infty, m_2)_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m < m_2\}, \\ (-\infty, m_2]_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m \leq m_2\}, \\ (m_1, \infty)_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 < m\}, \\ [m_1, \infty)_{\mathbb{M}} &\stackrel{def}{=} \{m \in \mathbb{M} \mid m_1 \leq m\}. \end{aligned}$$

3. **reals:** The set of all machine numbers

$$\mathbb{M},$$

i.e., an approximation of \mathbb{R} can be represented.

4. **numberSet:** For any finite set N of *floats* $f_a, 1 \leq a \leq k, k \in \mathbb{N}$, and *intervals* $i_b, 1 \leq b \leq l, l \in \mathbb{N}$, with $k + l \geq 2$, our prototype is able to provide a representation of $\bigcup N$, that is,

$$\bigcup_{1 \leq a \leq k} f_a \cup \bigcup_{1 \leq b \leq l} i_b.$$

That representation is minimised so that it maintain an ordered list of mutually disjoint *floats* and *intervals* such that no two list entries union to an interval.

5. **symbols:** The set of all symbols

$$\mathbb{S}$$

can be expressed.

6. **symbolSet:** We can represent any non-empty finite subset of \mathbb{S} ,

$$T \subset \mathbb{S}, \quad |T| \in \mathbb{N}_+.$$

7. **emptyValue:** There are representations of

$$\begin{aligned} \emptyset &\subset \mathbb{M}, \quad \text{and} \\ \emptyset &\subset \mathbb{S}. \end{aligned}$$

Given one of the above sets of values V , we may generate a variable

$$x, \quad \text{dom}(x) = V.$$

Obviously, it will in this context only make sense to choose a value V that is not an *emptyValue*. Otherwise, any constraint problem that mentions the variable will automatically be inconsistent.

Terms

The class of more involved representable terms, i.e., those built from the above listed values and variables, are presented below.

Note that all apart from the last one (*function*) represent extension functions in the sense of Def. 23. However, also in the case of *function*, we shall speak of an extension function: Indeed, Def. 23 can be altered appropriately to apply also for a function f that does not evaluate to a real number and that is not defined on a subset of some $\mathbb{R}^d, d \in \mathbb{N}_+$.

In order to recursively cover all representable terms, let $t_i, 1 \leq i \leq n, n \in \mathbb{N} \setminus \{0, 1\}$, denote a set of already given representable terms for which `getValue` yields a subset of \mathbb{M} .

1. **sin, arcsin, exp, log:** We can express

$$\begin{array}{l} \sin(t_1), \\ \arcsin(t_1), \\ \exp(t_1) \quad \text{and} \\ \log(t_1). \end{array}$$

Those represent the extension functions of `sin`, `arcsin`, `exp` and `log`, respectively.¹⁴

2. **power:** For a given machine number $m \in \mathbb{M}$, the expression

$$\text{power}(t_1, m)$$

can be represented which is the extension function of $f(x) = x^m$, imposed for the argument t_1 .

3. **sqr:** We may generate the square root of a term,

$$\text{sqr}(t_1) \stackrel{\text{def}}{=} \text{power}(t_1, 0.5).$$

4. **plus, mult, minus, div:** n -ary sum and product, as well as the binary expressions difference and quotient can be expressed,

$$\begin{array}{l} \sum_{1 \leq i \leq n} t_i, \\ \prod_{1 \leq i \leq n} t_i, \\ t_1 - t_2 \quad \text{and} \\ t_1 \div t_2. \end{array}$$

5. **function:** Suppose, there exists a subclass of `Function` that implements a user-defined function $f : D_1 \times D_2 \times \dots \times D_n \longrightarrow D$ (see Subsect. 5.5.3), where $n \in \mathbb{N}_+$ and the D_i 's and D are representable subsets either of \mathbb{M} or of \mathbb{S} . Furthermore, be for each $i \in \{1, 2, \dots, n\}$, s_i a representable term for which `getValue` produces a set of values that intersects with D_i .

Then we can obtain a representation for

$$f(s_1, s_2, \dots, s_n).$$

¹⁴We are thus overloading the function symbols here.

Recall that `sin`, `arcsin`, `exp` and `log` have been implemented as a few first examples for unary terms, in order to assess the effort for extending the expressional power of our prototype.

Relations

Most language facilities for constraints in our prototype are due to logical and-or-combinations and prominent binary relations for arithmetic terms.

1. **empty, full:** There are expressions for the unsatisfiable and the nonrestrictive constraint,

$$\emptyset, \quad \text{and} \\ \square.$$

2. **and, or:** For representable relations $r_i, 1 \leq i \leq n, n \in \mathbb{N} \setminus \{0, 1\}$, we can generate representations of the logical combinations

$$\bigwedge_{1 \leq i \leq n} r_i, \quad \text{and} \\ \bigvee_{1 \leq i \leq n} r_i.$$

3. **linear:** Let $x_i, 1 \leq i \leq n, n \in \mathbb{N}_+$, be variables with $\text{dom}(x_i)$ being representable by the above subsets of \mathbb{M} . Moreover, be $\lambda_i, 0 \leq i \leq n$, representable sets of machine numbers. Then, the extended linear arithmetic constraint

$$\sum_{1 \leq i \leq n} \lambda_i \cdot x_i \diamond \lambda_0$$

can be represented, where $\diamond \in \{=, \neq, \leq, <, \geq, >\}$.

4. **eq, neq:** Suppose we are given two arbitrary representable terms t_1, t_2 . We need to assume that **getValue** returns for both terms either a subset of \mathbb{M} or a subset of \mathbb{S} . Then there are representations for the equation and the disequation,

$$t_1 = t_2, \quad \text{and} \\ t_1 \neq t_2.$$

5. **lt, leq:** Let t_1, t_2 be as in the previous item. Only this time, we demand that **getValue** return in both cases a subset of \mathbb{M} . Then the inequalities

$$t_1 < t_2, \quad \text{and} \\ t_1 \leq t_2.$$

can be expressed.

6. **implies:** Let r_1, r_2 be two representable relations. Be furthermore r_1 an assignment, that is, an equation relating a variable and a set of values. Then the implication

$$r_1 \implies r_2 \stackrel{\text{def}}{=} (\neg r_1) \vee r_2$$

can be expressed.

Summary

Before we go on with miscellaneous language features of our prototype, we shall briefly summarise the set of constraints that can effectively be represented. For that, we simply expand the recursive definition patterns given in the above paragraphs on representable terms and relations.

- **extended arithmetic terms:** There are ways to represent arbitrary arithmetic terms involving the operations $+$, $-$, \cdot , \div , \sin , \arcsin , \exp , \log and powers with machine number exponents. The leaves of those term DAGs are variables and values for which `getValue` returns a set of values in \mathbb{M} .
- **extended arithmetic constraints:** We may express equations, disequations and inequalities of arbitrary extended arithmetic terms.
- **equations and disequations over finite-domain variables:** For a finite domain variable x , we can formulate equations and disequations that relate x to some finite set of symbols V ; $x = V$ and $x \neq V$.
- **and-or-junctions:** Arbitrary and-or-junctions of equations, disequations and inequalities can be expressed.
- $\emptyset, \square, \implies$: There are representations for the unsatisfiable and the nonrestrictive constraint, as well as for implications in which the left-hand constraint is an assignment.

Note that the meaning of an extended constraint, according to Def. 24, implies sometimes rather subtle sets of solution tuples; cf. Fig. 5.14.

For disequations, the usage of value sets with more than one element will often reduce the constraint to \square . This is illustrated by the following example:

Example 17: Be x a variable with $\text{dom}(x) = \{\text{red}, \text{blue}, \text{green}\} \subset \mathbb{S}$. Then, what is the restriction for x imposed by the disequation $x \neq \{\text{red}, \text{blue}\}$?

By an appropriate finite-domain version of Def. 24, that constraint equals

$$\pi_{\{x\}}("x \neq y"), \text{ where } \text{dom}(y) = \{\text{red}, \text{blue}\}.$$

Note that $x \neq y$ allows for any possible value that can be assigned to x , simply because $\text{dom}(y)$ has more than one element. Consequently, the projection does not at all restrict x . Therefore, the initial constraint $x \neq \{\text{red}, \text{blue}\}$ does not restrict x to a proper subset of its domain, and is hence equivalent to \square .

This may seem a bit odd at first sight, since we may have expected $x \neq \{\text{red}, \text{blue}\}$ to be equivalent to $x = \text{green}$, but it is not as the thorough interpretation shows.

Miscellaneous Elements

Our DTD has been designed also for the expression of further RCS-related objects. Most of those need to reference previously defined objects. The reference works by name, as any object defined in an XML file can be named in its respective defining XML phrase.

- **relationTemplate:** Sometimes, we happen to have great numbers of similar constraints, e.g. many Kirchhoff nodes in an electric circuit. In that setting, it is advantageous to formulate a *relation template*. The corresponding XML string resembles a prototypic Kirchhoff node and has a list of template variables. Each actual Kirchhoff node can then be modelled by a short XML string that references the template and provides the actual variables that are to replace the template variables.
- **oneOf:** The DTD provides a definition pattern for **OneOfs**. This expects a list of names of **Relations** that are to be the alternatives of the **OneOf**.
- **projection:** We can instantiate **Projections** from XML strings that reference a **Relation** or a **OneOf**, and the list of variables onto which to project.
- **contextSpace, cluster:** Going back to Fig. 4.1, it becomes now obvious, how XML strings may look like that represent context spaces and clusters.
- **leafNode, aggNode, aggForest:** Likewise, entire aggregation forests may be built up from XML strings. For the nodes, we need to provide all relevant attribute entries, as e.g. the forward relation. An aggregation tree is then just a reference to a root node; a forest is built from a list of nodes.

For examples of XML strings defining several RCS-related objects, consult App. B.

We shall close this section by pointing out that our prototype's constraint language can relatively easy be extended. This is, of course, to make it applicable to an ever greater class of constraints. For a detailed illustration, the reader is pointed to App. C. There, we present a recent development of our prototype that prepares it for *bus communication* problems.

Chapter 6

Experimental Results

In this chapter, a large set of various experimental results derived from our prototypic Java implementation are presented. Those support the claim made before, that the theoretical framework of RCS may be implemented so that the requirements collected in Chap. 2 can efficiently be addressed. A first set of experiments illustrates these services by means of a simple showcase example from electrics. Further examples focus on particular aspects such as non-linear constraints, scalability, explanations, reuse and accuracy.

Unless stated otherwise, all the following experiments have been undertaken on a Pentium III, 500 MHz PC with 640 MByte RAM.

The Java code is JDK version 1.4.1. Running RCS was accomplished by invoking the Java engine with an appropriate *.jar file. Before, all other applications had been closed, and the computer had been restarted in Windows 2000.

6.1 Proving the Capabilities of the Prototype

6.1.1 A Showcase Example Covering all Services of the Prototype

In order to clarify once again the set of constraint solving services provided by RCS, we shall start with a simple example from electrics and go through a typical sequence of experiments.

That small circuit is depicted in Fig. 6.1; App. B.5 provides the corresponding XML constraint definitions. The entire constraint problem consists of 8 wire constraints (each being a conjunction of two simple atomic conditions), 6 more involved component constraints (representing S, B, R, D, NSRC and NGND) and 4 value settings (as shown in the top half of Fig. 6.1).

We shall now deploy our prototype to solve the constraint problem given by the electric circuit in Fig. 6.1. Each of the following steps has been snapshot, and the resulting images can be found in App. D.1 together with more detailed descriptions.

Solving the Problem

After loading the XML-formulation of the circuit, we may let the prototype solve the problem, that is, determine consistency or inconsistency and - in the former case -

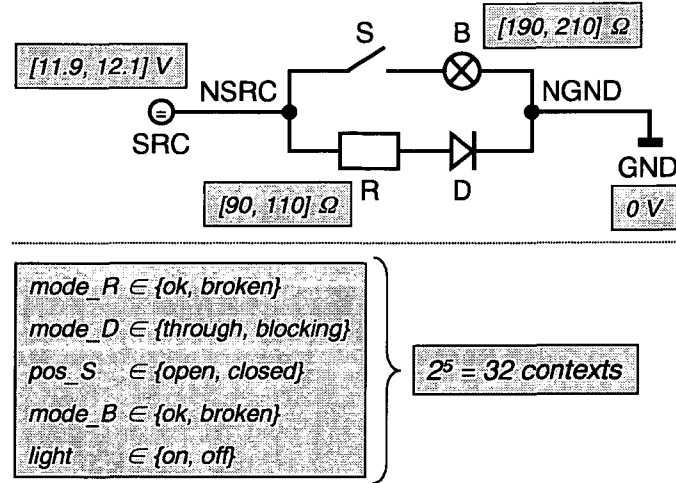


Figure 6.1: An Electric Circuit with 32 Contexts

find tightest restrictions for all variables. For our example, the prototype announces consistency with the following restrictions:

$c1_B$	$\in \{0\} \cup [0.05666667, 0.06368421]$
$c1_D$	$\in \{0\} \cup [0.10818182, 0.13444445]$
$c1_NGND$	$\in [-0.19812866, -0.16484849] \cup [-0.13444445, -0.10818182] \cup \dots$ $\dots [-0.06368421, -0.05666667] \cup \{0\}$
$c1_NSRC$	$\in \{0\} \cup [0.05666667, 0.06368421] \cup [0.10818182, 0.13444445] \cup \dots$ $\dots [0.16484849, 0.19812866]$
$c1_R$	$\in \{0\} \cup [0.10818182, 0.13444445]$
$c1_S$	$\in \{0\} \cup [0.05666667, 0.06368421]$
$c2_B$	$\in [-0.06368421, -0.05666667] \cup \{0\}$
$c2_D$	$\in [-0.13444445, -0.10818182] \cup \{0\}$
$c2_NGND$	$\in \{0\} \cup [0.05666667, 0.06368421]$
$c2_NSRC$	$\in [-0.06368421, -0.05666667] \cup \{0\}$
$c2_R$	$\in [-0.13444445, -0.10818182] \cup \{0\}$
$c2_S$	$\in [-0.06368421, -0.05666667] \cup \{0\}$
$c3_NGND$	$\in \{0\} \cup [0.10818182, 0.13444445]$
$c3_NSRC$	$\in [-0.13444445, -0.10818182] \cup \{0\}$
c_GND	$\in \{0\} \cup [0.05666667, 0.06368421] \cup [0.10818182, 0.13444445] \cup \dots$ $\dots [0.16484849, 0.19812866]$
c_SRC	$\in [-0.19812866, -0.16484849] \cup [-0.13444445, -0.10818182] \cup \dots$ $\dots [-0.06368421, -0.05666667] \cup \{0\}$
$light$	$\in \{on, off\}$
$mode_B$	$\in \{ok, broken\}$
$mode_D$	$\in \{through, blocking\}$
$mode_R$	$\in \{ok, broken\}$
pos_S	$\in \{open, closed\}$

r_B	\in	$[190, 210]$
r_R	\in	$[90, 110]$
$v1_B$	\in	$(-\infty, +\infty)$
$v1_D$	\in	$(-\infty, 0]$
$v1_{NGND}$	\in	$\{0\}$
$v1_{NSRC}$	\in	$[11.9, 12.1]$
$v1_R$	\in	$[11.9, 12.1]$
$v1_S$	\in	$[11.9, 12.1]$
$v2_B$	\in	$\{0\}$
$v2_D$	\in	$\{0\}$
$v2_{NGND}$	\in	$\{0\}$
$v2_{NSRC}$	\in	$[11.9, 12.1]$
$v2_R$	\in	$(-\infty, 0]$
$v2_S$	\in	$(-\infty, +\infty)$
$v3_{NGND}$	\in	$\{0\}$
$v3_{NSRC}$	\in	$[11.9, 12.1]$
v_{GND}	\in	$\{0\}$
v_{SRC}	\in	$[11.9, 12.1]$

Those results are found within 411 milliseconds. It is easy to check that they provide indeed tightest domain restrictions.

Altering the Problem

With our prototype, we can benefit from powerful reuse facilities, as studied in Chap. 3, whenever we like to solve a problem similar to the preceeding one. To illustrate that, we deploy our prototype as it is after solving the above initial problem. In our implementation, we can now easily activate an additional constraint *bulbIsLit* that captures the condition *light = on* and which had so far been deactivated and thus ignored. By adding that constraint, we simulate an information feed to the solver, e.g. the observation that the bulb is lit.

Having made that activation, we enforce the prototype to solve the new, more restrictive problem. The runtime drops significantly - somewhere below 100 milliseconds - since subtrees of the old aggregation tree can be reused. As expected, the findings for all variables have now also become more restrictive:

$c1_B$	\in	$[0.05666667, 0.06368421]$
$c1_D$	\in	$\{0\} \cup [0.10818182, 0.13444445]$
$c1_{NGND}$	\in	$[-0.19812866, -0.16484849] \cup [-0.06368421, -0.05666667]$
$c1_{NSRC}$	\in	$[0.05666667, 0.06368421] \cup [0.16484849, 0.19812866]$
$c1_R$	\in	$\{0\} \cup [0.10818182, 0.13444445]$
$c1_S$	\in	$[0.05666667, 0.06368421]$
$c2_B$	\in	$[-0.06368421, -0.05666667]$
$c2_D$	\in	$[-0.13444445, -0.10818182] \cup \{0\}$
$c2_{NGND}$	\in	$[0.05666667, 0.06368421]$
$c2_{NSRC}$	\in	$[-0.06368421, -0.05666667]$
$c2_R$	\in	$[-0.13444445, -0.10818182] \cup \{0\}$

$c2_S$	\in	$[-0.06368421, -0.05666667]$
$c3_NGND$	\in	$\{0\} \cup [0.10818182, 0.13444445]$
$c3_NSRC$	\in	$[-0.13444445, -0.10818182] \cup \{0\}$
c_GND	\in	$[0.05666667, 0.06368421] \cup [0.16484849, 0.19812866]$
c_SRC	\in	$[-0.19812866, -0.16484849] \cup [-0.06368421, -0.05666667]$
$light$	\in	$\{on\}$
$mode_B$	\in	$\{ok\}$
$mode_D$	\in	$\{through, blocking\}$
$mode_R$	\in	$\{ok, broken\}$
pos_S	\in	$\{closed\}$
r_B	\in	$[190, 210]$
r_R	\in	$[90, 110]$
$v1_B$	\in	$[11.9, 12.1]$
$v1_D$	\in	$(-\infty, 0]$
$v1_NGND$	\in	$\{0\}$
$v1_NSRC$	\in	$[11.9, 12.1]$
$v1_R$	\in	$[11.9, 12.1]$
$v1_S$	\in	$[11.9, 12.1]$
$v2_B$	\in	$\{0\}$
$v2_D$	\in	$\{0\}$
$v2_NGND$	\in	$\{0\}$
$v2_NSRC$	\in	$[11.9, 12.1]$
$v2_R$	\in	$(-\infty, 0]$
$v2_S$	\in	$[11.9, 12.1]$
$v3_NGND$	\in	$\{0\}$
$v3_NSRC$	\in	$[11.9, 12.1]$
v_GND	\in	$\{0\}$
v_SRC	\in	$[11.9, 12.1]$

The given solutions are still tightest restrictions.

Obtaining an Explanation

One of the changes with respect to the initial problem is that the restriction for pos_S has shrunk to the sole value *closed*. Though intuitively clear, we can let our prototype provide us with a minimal explanation for that finding. An appropriate function of the prototype produces a window with such a minimal explanation within 10 milliseconds; cf. Fig. D.5 in the appendix.

This explanation displays an aggregation tree which proves $pos_S = closed$. It has the leaf relations S , B , $wireStoB$ and $bulbIsLit$ that stand for the switch and bulb constraints, the intermediate wire constraint and the newly introduced condition concerning the bulb. A quick investigation with the help of Fig. 6.1 tells us that no constraint can be suspended from the explanation which is therefore indeed minimal.

Solving all 32 Contexts

With only few enhancements concerning the XML formulation of the circuit in Fig. 6.1, we can make our prototype recognise a context space with 5 `OneOfs` that reflect the possible settings for the five discrete variables. We shall not give the details of these simple enhancements here; they basically deal with the specification of the five `OneOfs`.

Once the context space has been recognised, a menu item allows us to solve all 32 contexts at once. Besides performing the consistency check for each context, this process will also collect all solutions for all variables, in any of the 12 consistent contexts.¹ Note that the problems will be solved one after another, always by reusing subtrees of the respective preceeding problem.

In our example, all findings produced by the prototype are correct, and again no tightest restriction is an overestimation. Although the 32 contexts are not solved in an optimised ordering, the entire process takes only around half a second.

6.1.2 Solving Some Non-Linear Problems

In Sect. 5.5, we have discussed how RCS can be made fit for solving also non-linear problems; the management and manipulation of arithmetic terms by the help of term rewrites are here the appropriate measure. In order to also illustrate those facilities, we shall have a brief look at some simple non-linear constraint problems.

First, our prototype is used to solve the problem in Ex. 13, where we intersect a circle and a straight line. Indeed, our implementation returns the two solutions $(x, y) \in \{(4, 3), (-4, -3)\}$. Moreover, it also provides - on demand - a window that lists all applied term rewrites, which are in our case:

$$\begin{aligned} t \cdot t &= t^2, \\ 0 + t &= t, \\ (a \cdot b \cdot \dots)^e &= a^e \cdot b^e \cdot \dots, \text{ and} \\ t + (s \cdot t) &= t \cdot (1 + s). \end{aligned}$$

Note that the deployed rewrites are exactly the ones that have been discussed in the context of Fig. 5.12.

A second bunch of experiments has been carried out using *quadratic electric resistors*. This is a straightforward approach to increase the algebraic degree of well-known and well-studied resistive networks. Figure 6.2 shows the quadratic constraint for such a resistor; here the “consumed” voltage is not proportional to the electric current but to its second power. The figure also presents the two basic circuits that may be combined to build arbitrary series-parallel-decomposable quadratic resistive networks, as e.g. the bottom right one.

Our prototype manages to solve all variables in the two basic quadratic constraint

¹Why are there 12 consistent contexts? Going back to Fig. 6.1, note that there are four consistent scenarios for the upper branch: $(pos_S, mode_B, light)$ is in $\{(open, ok, off), (open, broken, off), ((closed, ok, on)), ((closed, broken, off))\}$. And in the lower branch the plausible settings are $(mode_R, mode_D) \in \{(broken, through), (broken, blocking), (ok, through)\}$. The upper four and lower three cases then freely combine to 12 consistent scenarios.

problems. Although most of the rewrites are fairly simple, the complete list already becomes somewhat longish:

$$\begin{aligned}
 t \cdot t &= t^2, \\
 t \div 1 &= t, \\
 0 + t &= t, \\
 t - 0 &= t, \\
 t \div (-1) &= (-1) \cdot t, \\
 1 \cdot t &= t, \\
 (a \cdot b \cdot \dots)^e &= a^e \cdot b^e \cdot \dots, \\
 \text{value1} \cdot (\text{value2} \cdot t) &= (\text{value1} \cdot \text{value2}) \cdot t, \\
 0 \cdot t &= 0, \\
 (t^e)^d &= t^{e \cdot d}, \\
 t - t &= 0, \text{ and} \\
 t^1 &= t.
 \end{aligned}$$

(See also App. D.2, where a screenshot with all applied rewrites in the case of the parallel circuit can be found.)

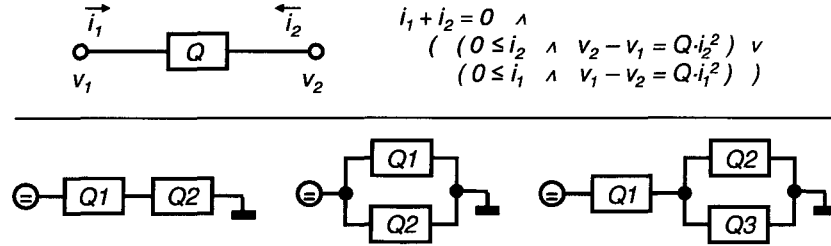


Figure 6.2: Increasing the Algebraic Degree - Quadratic Resistors

We mention that we encounter problems when trying to solve the bottom right circuit. But this can be fixed by implementing additional term rewrites. An interesting question which we did not investigate further, is to find a complete set of rewrites that suffices to solve all series-parallel-decomposable circuits with quadratic resistors.

6.1.3 Trading Runtime for Accuracy

In Subsect. 5.5.6 we have presented Lem. 12. It states that naive interval calculus will provide us with exact solution intervals whenever our term DAGs are trees.

The same subsection outlines an algorithm for utilising monotonous dependencies among the variables in a given constraint problem. Basically, the algorithm traverses the corners of the cube spanned by the input intervals, in order to find the two corners for which the respective term attains its minimal and maximal value. Let

us therefore call it the *traversing algorithm*, for later reference.

Note that, due to shortcomings in the implemented term simplification methods, RCS will often arrive at a DAG (which is not a tree) that could actually be replaced by a tree, only RCS does not notice. In that case, naive interval calculus will in general *not* yield narrowest interval solutions.

However, the proof of Lem. 12 shows that the traversing algorithm will still work and provide us with tightest bounds for all variables.

Already for a problem as simple as the bridge circuit R_1 , RCS encounters DAGs (which are not trees) that could actually be replaced by trees. As already pointed out, the use of naive interval calculus will then no longer yield narrowest interval solutions. But the user - who is aware of the monotonous character of the functional dependencies - can tell RCS to switch from naive interval calculus to the traversing algorithm which produces exact bounds for all unknowns. This extra setting can - as all other settings - be given via the RCS control pannel.

Appendix D.3 presents screenshots of RCS after solving R_1 , first by naive calculus and then by the traversing algorithm which assumes monotonous dependencies and produces indeed the exact solution intervals.

As expected, the higher accuracy is reflected in a longer runtime for solving the problem: In the given example the overall runtime is approximately four times as long as in the case of naive interval calculus.

6.2 The Runtime Behaviour for Selected Problem Families

In this section we shall take a look at some experimental results derived for three families of scalable constraint problems. These are the already introduced circuit families $(R_k)_{k \in \mathbb{N}_+}$, $(D_k)_{k \in \mathbb{N}_+}$, and a family of four spacecraft propulsion systems. The latter have been taken from a diagnosis application in the context of an *automated transfer vehicle* (ATV) that has actually been used in space. More details concerning the ATV shall be given in Subsect. 6.2.3 where we also present the respective experimental results.

Whereas the first two families must clearly be seen as academic examples, the third is a real-world application that is to support the claim that RCS embodies indeed a practicable approach to constraint solving for model-based engineering applications.

6.2.1 The Family $(R_k)_{k \in \mathbb{N}_+}$

For the family $(R_k)_{k \in \mathbb{N}_+}$, we investigated the runtime behaviour for $k \in \{10, 20, \dots, 100\}$. More concretely, forward times and backward times have been recorded, together with the portion of forward time that was spent on the built-in on-the-fly aggregation strategy; cf. Subsect. 3.5.1.

According to our discussion in the Sects. 4.2 and 4.3, the overall computation time can be expected to be quadratic in the number of boxes, that is, $O(k^2)$. The main precondition for that result to hold is our *low density assumption*. Moreover, when the time spent on strategic matters can be neglected, the order should decrease to

some $O(k)$. Therefore, the runtime figures should in that case present a linear curve. Figure 6.3 shows the three curves for strategic and non-strategic forward phase as well as for the backward phase. Note that the curves have been depicted in a stack, i.e., on top of each other. Consequently, the upper boundary can be read as the total computation time for each problem instance. We find an “almost” linear curve, as forecasted by our theoretical discussion. But why is the curve not linear?

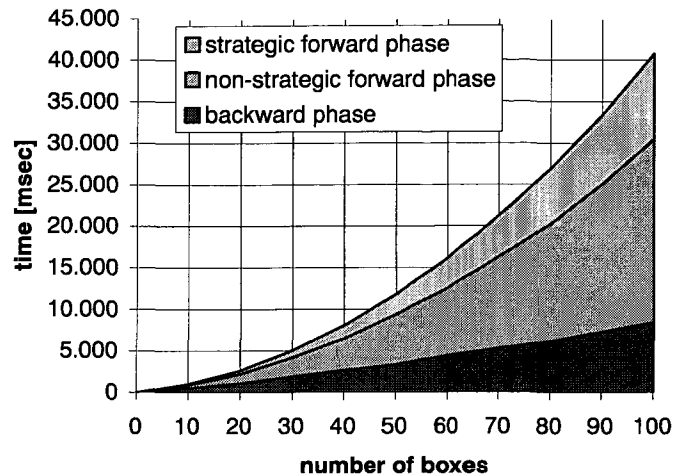


Figure 6.3: Runtimes for the Family $(R_k)_{k \in \{10, 20, \dots, 100\}}$

Obviously and seen in relation to the other two curves, the portion of time spent on strategic matters during the forward phase can *not* be neglected. This violates one of the preconditions that would guarantee a linear runtime. The other reason why we actually witness a non-linear curve is that the second pre-condition does not hold either: Figure 6.4 shows that the *low density assumption* is violated. On average, we have almost one more variable in our non-leaf relations than in the leaf nodes.

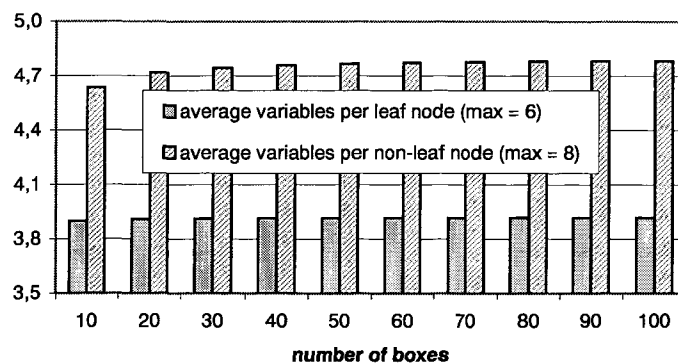


Figure 6.4: Scope Sizes for the Analysis of $(R_k)_{k \in \{10, 20, \dots, 100\}}$

Appendix E provides some additional information concerning our experiments, especially about the structure of leaf and non-leaf relations. As an example, Fig. E.1 proves that non-leaf relations are more complex than the initial leaf relations of the problem R_k : We have - on average - half an atom more in the disjunctive normal form. This illustrates the violation of another notion that we have been emphasising especially in Ex. 3: Obviously, the result of aggregating two relations does not remain as simple as each of the two partaking relations, but tends to become more complex as we move up in the aggregation tree.

6.2.2 The Family $(D_k)_{k \in \mathbb{N}_+}$

The picture looks much different for the family $(D_k)_{k \in \mathbb{N}_+}$; see Fig. 6.5. This time, one obtains exponentially growing time consumptions for both the non-strategic forward and the backward phase. The strategic time portion has been omitted since it contributes only to a neglectable degree. Furthermore, this time the analysis was run for $k \in \{1, 2, \dots, 10\}$.

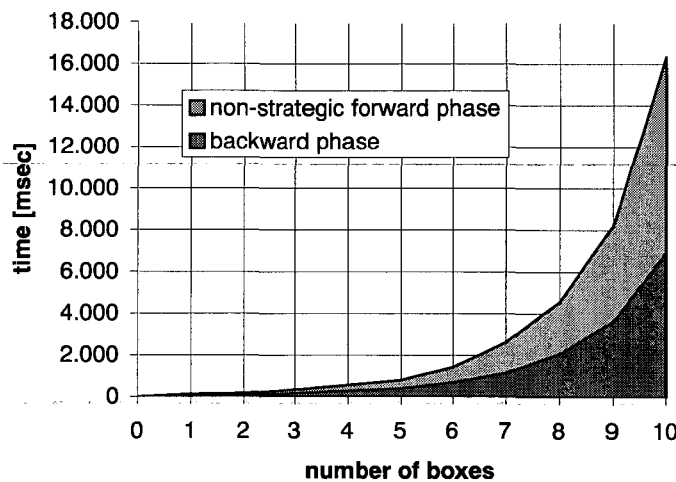


Figure 6.5: Runtimes for the Family $(D_k)_{k \in \{1,2,\dots,10\}}$

The reasons for the different behaviour become obvious by taking a look at the scope sizes and some structural measurement.

Again, we find approximately one more variable in non-leaf nodes, compared to the leaves; see Fig. 6.6. This figure also shows that we sometimes even have 10 variables in a non-leaf relation, whereas the maximal number for leaves is 6. We conclude that the *low density* assumption is far from being satisfied. Consequently, we must not expect a linear runtime.

The more influential factor for the given exponential behaviour lies however in the complexity of the relations. In the previous family, there was a constant added complexity of only about half an atom; see again Fig. E.1. This time, we obtain an ever rising average number of atoms in non-leaf nodes as we move from smaller constraint problems to bigger ones; see Fig. E.2. For $k = 10$, there are more than 18

atoms in the average non-leaf node, as opposed to less than 3 for leaves. Moreover, Figure E.3 shows that the average number of atoms does no longer provide an appropriate measure since the maximal number of atoms in a non-leaf node of the computed aggregation tree even exceeds 1400.

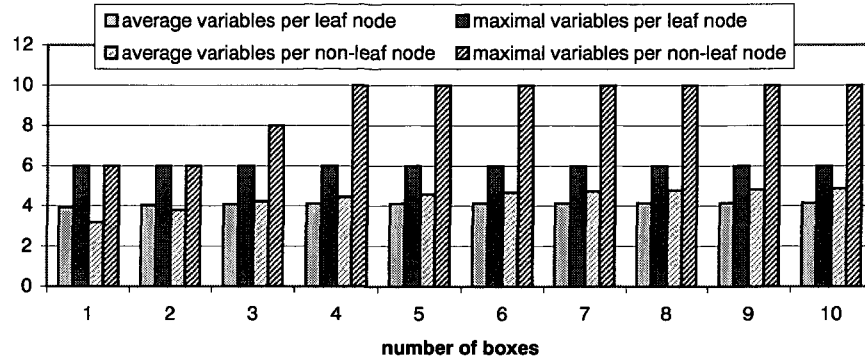


Figure 6.6: Scope Sizes for the Analysis of $(D_k)_{k \in \{1,2,\dots,10\}}$

It is in the nature of the problem family $(D_k)_{k \in \mathbb{N}_+}$ that there are many potential cases for the current flow through the entire circuit. (The reader should remember that each diode accounts for two potential cases, so that we end up with 2^d theoretical possibilities for d diodes.) At computation time, this is reflected by a large number of disjunctive relations in the aggregation tree. Figure E.4 presents the average number of disjuncts per aggregation node. It is easy to see that the average node in the tree becomes more and more complex as we move to bigger instances. Also here, there exist very “bad” nodes with even more than 120 disjuncts in the case of $k = 10$; see Fig. E.5.

6.2.3 ATV: A Real-World Application

As already mentioned above, ATV stands for *automated transfer vehicle*. This third problem family in the current Sect. 6.2 consists of four spacecraft propulsion systems, the largest of which shows reasonable resemblance to real-world propulsion systems, e.g., to the one that had actually been used in the ATV.

The smaller three problems have been derived from the fourth, by removing redundant paths for the provision of oxidiser and fuel. There exist also redundant pipes for the helium that pressurises the propellant tanks; see Fig. 6.7.

Apart from the shown components, the system comprises furthermore sensors for pressure, temperature and valve positions. The pressure regulators are modelled using piecewise linear constraints. The engine, in which the oxidiser and fuel mix, react and produce heat and thrust, is represented by a procedurally defined relation that uses characteristic lines to compute the chamber pressure and temperature as functions of fuel and oxidiser pressure. In effect, these functions are bilinear interpolations based on a set of pre-defined pairs of input vectors and output values.

In order to get trustworthy measurements, we randomly generated - for each of

the four propulsion systems - 1000 constraint problem instances. This was done as follows.

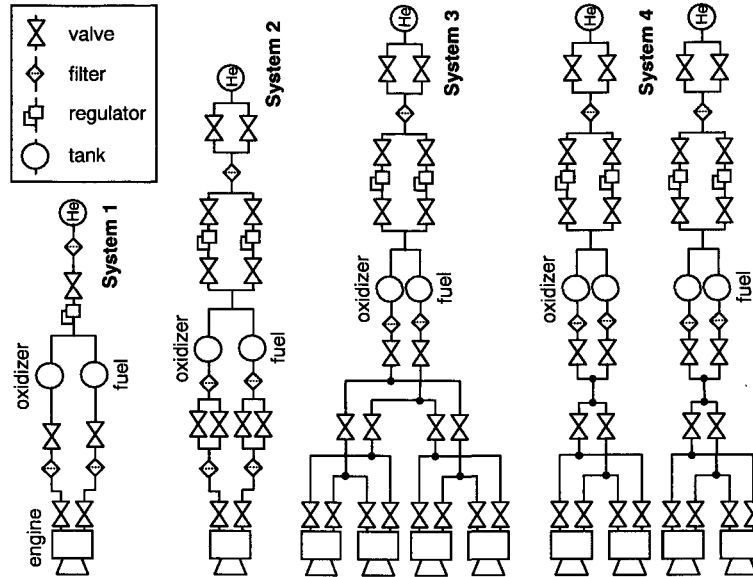


Figure 6.7: Four Propulsion Systems with Increasing Redundancy

The default setting for each of the four systems was the one in which all components behave normally and all valves are open. (It should be mentioned that this scenario is not a realistic one, as in practice not all redundant paths will simultaneously allow for a flow of helium, fuel or oxidiser, respectively.) Let us call that context C_0 . Then, we fixed a set of m valves for which RCS was allowed to modify control settings (*closed* or *open*) and behavioural modes (*normal*, *stuck at closed*, *stuck at open*). Thus, we spanned a space of 6^m contexts.

The first of the 1000 contexts was C_0 . Next, given C_0 , the prototype randomly chose 2 out of the m valves and randomly altered their states within the set of the 6 possible states, to arrive at C_1 . By repeating this pattern 999 times, a sequence of 1000 contexts was generated on the fly. Note that thereby any two neighbouring contexts in the sequence differ by exactly two relations, which made it possible to utilise the same sequence for a reuse-based, incremental analysis; see Subsect. 6.4.2.² The results shown in Fig. 6.8 are therefore average measurements over the 1000 contexts. The table shows again that the portion of time spent on strategic matters is not to be underestimated. The upper curve in Fig. 6.16 depicts the total runtimes. There, each x-coordinate of a black square equals the number of relations in the constraint problem. This number can also be read off the left-most column of the table in Fig. 6.8: It must equal the number of aggregations per context plus one. The curve in Fig. 6.16 suggests an “almost” linear order, as discussed in Sects. 4.2

²Also, the given procedure does not ensure that all of the 1000 contexts be mutually different. However, this does not play a role in the context of the experiments.

Non-Incremental Analysis (Contexts: 1000)					* Average Measurements			
	forward phase				backward phase			total
	aggs/cbxt	msec/agg*	strategy time*	msec*	joins/cbxt	projs/cbxt	msec*	msec
System1	121	0,58	30,65%	100,51	263	262	85,36	185,87
System2	287	0,70	41,05%	338,66	632	631	229,15	567,81
System3	394	0,92	40,51%	611,83	868	867	340,48	952,30
System4	597	1,10	44,93%	1196,65	1315	1314	540,52	1737,17

Figure 6.8: Runtimes for Non-Incrementally Solving the ATV Systems

and 4.3. And indeed, the *low density assumption* seems to hold: Figure E.6 shows all structural data recorded during the experiment. According to that data, non-leaf and leaf relations are comparable in both the number of variables and their complexity. We have only about 0.3 to 0.8 more variables in non-leaf nodes. And the numbers of atoms, conjuncts and disjuncts differ also only very little.

6.3 Minimal Conflicts & Explanations vs. Conventional Constraint Suspension

Before we move on to issues of performance and in order to first illustrate the explanatory facilities of RCS we shall mention that RCS is capable of retrieving the two minimal conflicts for the 3-queens problem; cf. Ex. 6. The interested reader can find some remarks in App. D.4 which also shows a screenshot of RCS after it has successfully dealt with the explanation task.

Apart from the small example of 3-queens, we will in this section only focus on finding *one* minimal conflict or explanation, respectively. In fact it is the case for all investigated problem instances that there is just one minimal conflict or explanation.

To clarify the below experimental results, it is necessary to recall the algorithms developed in Subsects. 4.4.2 and 4.4.3.

Starting with one minimal explanation, let us once more take a look at Fig. 4.8. The basic idea is here to alter an existing aggregation tree, so that the method for retrieving one minimal conflict can be deployed. The most important step in that alteration is captured in line (7). In our implementation, we do however *not* alter the respective aggregation node but instantiate a new one. Consequently, we shall below report the number of newly-created aggregation nodes instead of altered nodes.

Moreover, also the actual realisation of Fig. 4.7's pseudo-code involves the creation of new aggregation nodes. Where do new nodes come from when computing one minimal conflict?

First, we recall an obvious code improvement already introduced in Subsect. 4.4.2: We may implement the method `getOneConflict` so that it not only return a set of relations S , but also their combined join $s = \bowtie S$. Line (6) is then to be replaced by `return (S1 \cup S2, s1 \bowtie s2)`. Now, one can even go a step further: The join

$s1 \bowtie s2$ may be replaced by its projection onto the set of variables of the given aggregation node.³ The consequent next measurement is to return not just the aggregation of $s1$ and $s2$ but to return a new aggregation node with $s1$ and $s2$ as successors. Thereby, we obtain in the end not only the minimal set S of conflicting constraints but also an aggregation tree, the leaves of which form the set S . Moreover, the root of that tree is the empty relation \emptyset ; and we can regard the tree as a proof for the fact $\bowtie S \sim \emptyset$. This is what has been done in our prototypic Java implementation. Consequently, our algorithm for computing one minimal conflict creates a new aggregation node whenever line (4) of the pseudo-code of Fig. 4.7 is approached. Note that this is the case if and only if the respective minimal conflict spreads over both subtrees of the given aggregation node.

Apart from the number of newly-created aggregation nodes, we are going to report the number of visited nodes. This includes leaf nodes as well as aggregation nodes, and gives an impression for how focussed the computation was.

In Subsects. 4.4.2 and 4.4.3, we have argued that - provided the *low density assumption* holds - the worst case cost for computing one minimal conflict or explanation is linear in the size of the original constraint problem instance. In our experiments, we have not witnessed such a worst case; partly because the aggregation trees were rather balanced and because of only small minimal conflicts and explanations compared to the total number of relations. In fact, our experiments have not been designed to investigate the claim of linear runtimes.

Instead, we are going to compare our new explanation algorithms with *constraint suspension*. Constraint suspension is a popular algorithm, e.g. for minimising conflicting sets of constraints. The discussion in Sect. 2.4 makes clear that explanatory facilities are also very rare in existing solvers. Therefore, constraint suspension is often even the only way to obtain minimal conflicts. This is the more true as explanatory services tend to be only an afterthought in many constraint solver implementations.

Chapter 3 shows that relational aggregation is a powerful framework in which explanatory services can naturally and straightforwardly be addressed. In what follows it becomes clear that aggregation trees allow for a *focussed* search for minimal conflicts and explanations, whereas constraint suspension is an unfocussed and far less efficient method.

The testbed uses the same implementation for *join* and *project* for both our new algorithms and constraint suspension. Therefore, the obtained results are very explicit about the intrinsic characteristics of the high-level algorithms themselves.

6.3.1 Constraint Suspension

The new algorithms for computing minimal conflicts and explanations are based on existing aggregation trees. In other words, the tree guides the algorithm's search.

³It is easy to see that this alteration does not affect the correctness of the algorithm; just as Subsect. 4.4.2 suggests an additional first line of code that replaces the relation c by an appropriate projection $c.\text{project}(X12)$.

It is quite the opposite for constraint suspension, which shall briefly be explained before we come to the experimental results.

Suppose, we are looking for a minimal conflict. Then the idea is to temporarily suspend a candidate relation r and check whether the remaining set of relations is still contradictory. If so, the suspension becomes permanent. Otherwise, the relation r is known to contribute to the minimal conflict. It will hence be re-introduced and marked as non-suspendable. The iterative process terminates with a minimal conflict set when all remaining relations have been marked as non-suspendable.

```

Engine:  function getOneConflict()
    F ← AggForest computed by forward phase
    n ← root node in F with F.forwardRelation.isEmpty()
    conflict ← ∅
(1)  suspension( $\Lambda(n)$ , conflict)
(2)  return conflict
end

Engine:  function getOneExplanation(Variable x)
    L ← set of leaves in AggForest computed during forward phase
    Vx ← solutions.entryFor(x)
    sx ← Vx.complement()
(3)  if sx.isEmpty() throw exception 'nothing to explain'
    conflict ← {sx}
(4)  suspension(L, conflict)
(5)  return conflict \ {sx}
end

Engine:  procedure suspension(Sets of LeafNodes S, C)
(6)  if (S = ∅) then return
    r ← random member of S
(7)  S ← S \ {r}
    forward(S ∪ C)
    if (state = inconsistent)
        F ← AggForest computed by forward phase
        n ← root node in F with F.forwardRelation.isEmpty()
(8)  S ←  $\Lambda(n) \setminus C$ 
    else
(9)  C ← C ∪ {r}
(10) suspension(S, C)
end

```

Figure 6.9: Pseudo-Code for Explanatory Services Using Constraint Suspension

Of course, the check for consistency (that determines whether a suspension becomes permanent) can be accomplished by building an aggregation forest, i.e. by the for-

ward phase of the relational framework.

Figure 6.9 gives, for completeness sake, pseudo-code formulations of suspension-based methods for obtaining minimal conflicts and explanations. The core method at the bottom, **suspension**, expects two sets of leaf nodes; S , the set of *suspendable* relations and C , the set of relations that must belong to the sought-for *conflict*. The invariant of this method is that the union of both sets is contradictory;

$$(\bowtie S) \bowtie (\bowtie C) \sim \emptyset.$$

It is easy to verify that this condition holds for the invocations in both line (1) and (4). In line (1) we know that $\Lambda(n)$ is inconsistent; in (4) we just take all leaves together with the complement sx of the finding Vx for variable x .⁴

In contrast to line (2), we need in line (5) to remove sx from the returned set. (Note that **suspension** only adds new members to C and never removes one; thus sx will still be a member of the set **conflict** in line (5).)

In order to prove that the above invariant also holds at every recursion in line (10), regard the bottom method: r is being suspended from S in line (7). Line (9) clearly establishes the invariant. In the other case - the case of inconsistency - we know that the leaves "beneath" n form an inconsistent set of constraints. Moreover, $\Lambda(n)$ must even subsume C , as each member c of C has been added because suspending c led to a consistent set of constraints (see the **else** branch). Therefore, line (8) actually says $\Lambda(n) = S \cup C$. This proves the validity of the invariant in line (10) also for the case that the **if** branch had been processed.

As to termination, the bottom method must clearly eventually end in line (6) as S shrinks monotonously; see line (7).

Note that the method would also work if the **if** branch would not contain any statement. But the given pseudo-code realises a somewhat more efficient version of - an otherwise completely unfocussed - constraint suspension: Whenever the forward phase proves inconsistency by building an aggregation forest with *more than one tree*, any leaf that does not belong to $\Lambda(n)$ can obviously immediately be suspended, too.

6.3.2 Explaining Zero Bridge Currents in Special Instances of the $(R_k)_{k \in \mathbb{N}_+}$ Family

For our first experiment, let us regard once again the family of bridge circuits $(R_k)_{k \in \mathbb{N}_+}$; see also the top part of Fig. 2.7. Only this time we replace the resistance for R_4 by 600 Ω . Thereby, the following equation is established:

$$\frac{R_1}{R_2} = \frac{R_3}{R_4}.$$

⁴Clearly, **getOneConflict** assumes the inconsistent **Engine** state; and **getOneExplanation** the consistent one.

If the condition for line (3) is satisfied, then this means that Vx represents the full relation \square . I.e., there is no restriction for x and hence nothing to explain; cf. line (3) in Fig. 4.8.

It is well-known that in this case, the *bridge current*, i.e. the current through R_5 diminishes.⁵ Interestingly, as the argumentation suggests, the fact that the bridge current is zero is independent from the remaining circuit: We are obviously able to prove that fact only by regarding the respective box of five resistors. Consequently, a proof consists exactly of 24 relations: the once that capture the 5 resistors, the 5 corresponding resistance settings, the 4 Kirchhoff nodes and the 10 in-box wires. Note that this result is not affected by the number of boxes k .

In our experiment, both constraint suspension and our new explanation algorithm yield the outlined smallest explanation of 24 relations. The variable for which the solution was to be explained is the bridge current in the middle box, that is, in the box B_m , where m is the rounded-down half of k .

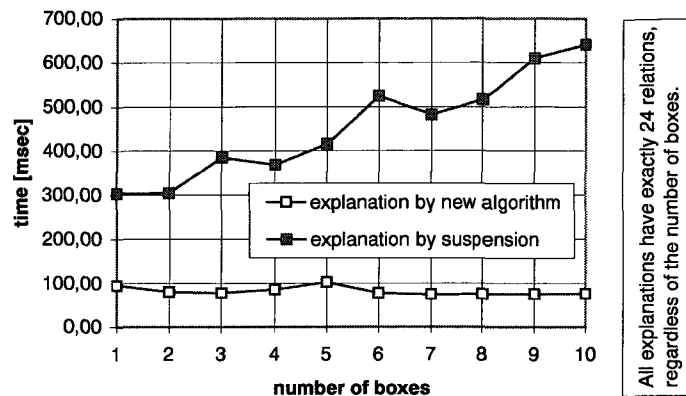


Figure 6.10: Explaining Zero Bridge Currents

Figure 6.10 presents the runtime results. It illustrates that our new algorithm consumes a portion of time that relates to the explanation size⁶ rather than to the number of constraints in the context. This latter causality holds indeed for constraint suspension: When there are more constraints (to choose from for random suspension), then the time for finding all nonsuspendable constraints will increase accordingly. We can study this increase in Fig. 6.10. The curve's somewhat unsteady gradient is a consequence of the small number of repetitions for each invocation of suspension-based `getOneExplanation`. (Each explanation has been computed 10 times; the given time consumptions are average measurements.)

As mentioned above, we have also recorded the number of visited nodes in the pre-computed aggregation tree, and the number of newly-created nodes; see Fig. 6.11. For our new algorithm, the former number is almost constant. The number of new

⁵An intuitive argumentation runs as follows: The current that runs into N_2 equals the current that leaves N_3 . Moreover, since the two ratios are - due to the above condition - equal, that current splits in N_3 just in the same way as it splits in N_2 . Therefore, the currents through R_1 and R_3 (and analogously through R_2 and R_4) must be equal. Considering the condition for the Kirchhoff node N_1 (or N_4), we conclude that the current through R_5 must be zero.

⁶To be more exact, the explanation time based on Th. 4 relates to the size of that aggregation subtree which captures the aggregation of the respective box B_m .

nodes hardly exceeds 100, whereas with suspension more than 400 nodes are being instantiated. Again, for suspension the number rises as the number of boxes k increases.

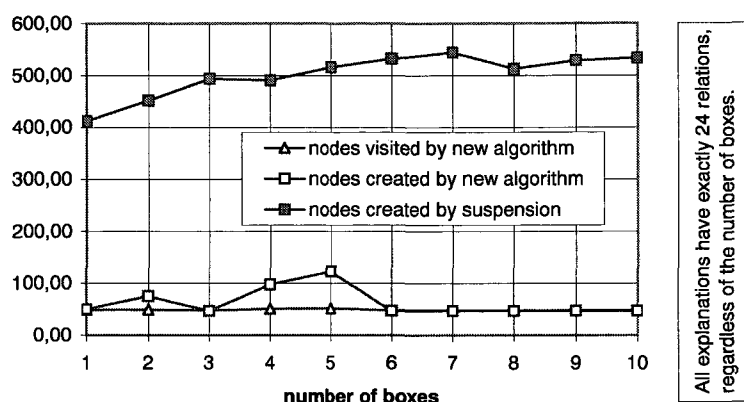


Figure 6.11: Node Statistics for the Explanation of the Zero Bridge Current

6.3.3 Conflicts and Explanations for the ATV Family

The results of Subsect. 6.3.2 are not due to specifics of the bridge circuit family. Rather, we find the same characteristics for appropriate instances in the context of the four ATV propulsion systems as shown in Fig. 6.7.

In order to apply our new algorithms for computing minimal conflicts and explanations, as well as the suspension versions shown in Fig. 6.9, we constructed four consistent problem instances. More concretely, for each of the four systems, certain valve settings were chosen that imply both the oxydizer and the fuel pressure to be non-zero at one of the engines. In the constraint model, this implication is - roughly speaking - captured in a variable `bothPressuresNonZero`. That is the variable for which we imposed our new explanation algorithm, as well as the suspension-based one. Both algorithms found the only existing minimal explanation with a size as presented in the "explanation" column of Fig. 6.13. That column also gives the total numbers of the four underlying consistent ATV constraint problems.

Conflicts (Data for One Inconsistent Context)					* Average Measurements of 100 Runs	
	conflict	new algorithm			suspension	
	size; total relations	created agg. nodes	visited nodes	msec*	created agg. nodes*	msec*
System1	14; 101	13	41	27,44	318,83	191,69
System2	19; 235	18	63	26,34	590,00	420,18
System3	31; 327	30	79	152,62	923,18	1.013,64
System4	31; 491	30	77	148,11	904,53	1.020,98

Figure 6.12: Conflict Computation for the Four ATV Systems

By adding an additional constraint which negates the afore mentioned implication, we arrive at four inconsistent constraint problems. Those are the instances for which the algorithms for finding one smallest conflict were invoked. Again, there is only one minimal conflict, for each of the four scenarios. Note that the conflict size must be one more than that of the corresponding explanation. This can also be seen in the "conflict" column of Fig. 6.12, which also reports total problem sizes.

Explanations (Data for One Consistent Context)				* Average Measurements of 100 Runs		
	explanation	new algorithm			suspension	
	size; total relations	created agg. nodes	visited nodes	msec*	created agg. nodes*	msec*
System1	13; 100	108	37	19,69	324,77	199,96
System2	18; 234	246	59	30,60	605,20	486,67
System3	30; 326	257	78	215,98	923,23	958,92
System4	30; 490	288	76	214,42	837,48	1.338,53

Figure 6.13: Explanations for the Four ATV Systems

This time, we recorded 100 runs; the Figs. 6.12 and 6.13 present the resulting average measurements. Again, as was the trend also in Fig. 6.10, the new algorithms outperform the suspension-based algorithms by a runtime factor of about 5 to 10. Moreover, the number of created aggregation nodes is about 3 times as high in the case of suspension-based explaining and even 30 times as high for suspension-based conflict retrieval. (Recalling again the discussion in the beginning of this Sect. 6.3, it is clear that our new algorithm for explaining needs to create some more aggregation nodes than the one for obtaining a minimal conflict. This explains the remarkable shift from 30 to 3.)

We shall once more point out that our new algorithms are guided by the given aggregation trees whereas suspension realises a blind search (if we forget about the enhancement in our suspension routine in Fig. 6.9 which also utilises the structure of aggregation forests). Due to their mode of functioning, our new algorithms produce runtimes which relate to subtree sizes (if not even conflict and explanation sizes themselves), as opposed to the total problem size. This latter order is however appropriate and logical for suspension-based algorithms.

6.4 The Effect of Reuse along Similar Contexts

Appendix D.5 illustrates with two screenshots the facilities of our prototype to solve entire context spaces instead of just a single constraint problem. For smaller context spaces, this is certainly a practicable way to obtain solutions for all variables in each of the specified contexts. In the case of larger spaces, it is certainly better to redirect the results into output files.

This section starts with the results for the 1-bit full adder, as introduced in App. B.1 and the small electric circuit depicted in Fig. 6.1 that gives rise to a space of 32 contexts. Clearly, for small context spaces the effect of reuse will not be too dramatic. This is due to the fact that the gained runtime is almost entirely consumed by certain one-time effects.

Therefore, our third example - again undertaken in the context of the ATV propulsion system - shall make the benefits of reuse very clear. Here, we have run the implementation with 1000 contexts, and one-time effects should thus no longer play a role.

6.4.1 Two Small Context Spaces

Figure 6.14 presents the runtime results for analysing all 16 contexts of the 1-bit full adder. (Again, the reader is pointed to App. B.1, where a model of the problem is illustrated. Figures D.10 and D.11 depict screenshots of our prototype before and after tackling the given context space.)

Likewise, Fig. 6.15 shows similar results for the space of size 32, spanned by the electric circuit of Fig. 6.1.

Runtimes in Milliseconds	Average Measurements of 10 Runs		
	16 contexts by reuse		16 contexts without reuse
	first context	remaining 15 contexts	
forward time, contains:	35,1	260,7	502,9
- built-in strategy	3,0	25,1	91,1
- non-strategic time	32,1	235,6	411,8
backward time	10,1	50,0	79,1
total time	45,2	310,7	582,0

Figure 6.14: The Effect of Reuse for the 1-Bit Full Adder

Runtimes in Milliseconds	Average Measurements of 10 Runs		
	32 contexts by reuse		32 contexts without reuse
	first context	remaining 31 contexts	
forward time, contains:	20,0	110,1	302,2
- built-in strategy	2,0	18,0	58,0
- non-strategic time	18,0	92,1	244,2
backward time	61,2	246,3	333,5
total time	81,2	356,4	635,7

Figure 6.15: The Effect of Reuse for the Example Circuit in Fig. 6.1

We see that already for as few as 16 contexts there is a saving in runtime when we deploy the reuse facilities of RCS: The total computation time for all 16 contexts is $45,2 + 310,7 = 355,9$ milliseconds versus 582 milliseconds in the case of solving all 16 contexts without reuse. In the second example, the corresponding measurements are 437,6 versus 635,7 milliseconds.

In order to get an even better understanding of the quality of our reuse facilities, we shall now turn to much larger context spaces.

6.4.2 1000 Contexts: The ATV Family

In Subsect. 6.2.3 it has been explained - for each of the four ATV systems - how a sequence of 1000 contexts can be generated in an on-the-fly fashion, so that any two neighbouring contexts differ by exactly two relations. Actually, the results in Fig. 6.8 and those given here have been obtained by running two distinct instances of *Engine* in parallel on the produced constraint problems. Of these two instances one did not and one did deploy reuse. It is therefore legitimate to directly compare the results of both experiments, in order to emphasise the effects of reuse.⁷

Incremental Analysis (Context Switches: 999, Altered Relations: 2)					* Average Measurements			
	forward phase				backward phase			total
	aggs/cbt*	msec/agg*	strategy time*	msec*	joins/cbt	projs/cbt	msec*	msec
System1	15,82	0,71	15,93%	13,31	263	262	85,28	98,59
System2	21,39	1,32	9,59%	31,12	632	631	240,18	271,30
System3	20,63	1,92	8,98%	43,48	868	867	347,16	390,64
System4	22,83	2,58	5,73%	62,36	1315	1314	537,49	599,85

Figure 6.16: Runtimes for Incrementally Solving the ATV Systems

The incremental runtime results are presented in Fig. 6.16. They prove that incremental processing drastically decreases the effort spent on the forward phase. Moreover, the number of forward operations relates to the theoretical result of $O(\log(n))$. Also, the portion of time spent on strategic matters drops significantly from over 40% to less than 10% in the incremental case. This is not surprising, as repairing single aggregation paths does not leave RCS much of a choice.

On the other hand, each single aggregation seems to become harder. This is indicated by the rise in time per aggregation: Whereas this parameter approaches one millisecond for system 4 in the non-incremental case, we witness a rise over 2.5 milliseconds in the incremental case. Obviously, maximal reuse and simplicity of aggregation may happen to be contrary goals.

The runtimes covering the backward phase do not differ. Note that this is as expected since we have used the same method in both experiments for solving all variables.

Figure 6.17 presents a direct comparison of non-incrementally and incrementally solving the four ATV systems. Here, we see the average runtimes per context. (In the case of a reuse-based, that is, incremental analysis, this figure comprises also the initial non-incremental context.) Obviously, incremental analysis saves almost two thirds of the non-incremental runtime.

Again, as has also been the case for the bridge circuit family, $(R_k)_{k \in \mathbb{N}_+}$, the presented graphs suggest an “almost linear” runtime, at least for the three larger systems. In

⁷In the incremental case, i.e. the one in which we deployed the reuse facilities, we implemented a trigger for abandoning the current aggregation tree and compute the next context from scratch. This trigger fired whenever the current aggregation tree had become very unbalanced. (Note that repairing existing aggregation trees over and over again is indeed very likely to make them extremely unbalanced and hence unsuitable for further computations.) The presented experimental results are nonetheless valid, as the trigger fired less than 10 times in 1000 contexts.

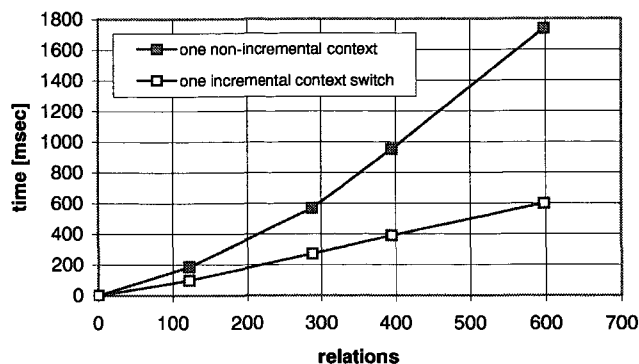


Figure 6.17: Comparison of Incremental and Non-Incremental ATV Runtimes

order to deal some more with this aspect, the reader is pointed to the structural data provided by Fig. E.6. There it can be seen that the number of variables per relation does indeed not depend on whether the relation is a leaf of the aggregation tree, or not. Similarly, other features, like the number of atoms, conjuncts or disjuncts, are almost identical for leaf and non-leaf relations. The only remarkable difference is the balancedness of the aggregation trees: As expected, in the non-incremental case trees tend to be more balanced. On the contrary, if subsequent contexts are solved by reuse, then longest path and average path differ - at least for systems 3 and 4 - by more than 20 nodes.

For completeness sake, we shall also give runtimes for the 4 ATV systems that have been obtained with the Model-based Diagnosis System (MDS), which we already mentioned in Subsect. 2.2.2. It is the only system that was used to directly compete with our prototype since it also implements reuse facilities (there based on an ATMS; see Subsect. 2.2.2).

*msec Measurements on a 733 MHz PC with 1024 Mbyte RAM				
	System 1	System 2	System 3	System 4
one non-incremental context*	≈ 0	840	7.900	2.720
one incremental context switch*	≈ 0	730	3.740	2.650

Figure 6.18: ATV Runtimes Acquired by MDS

The runtimes presented in Fig. 6.18 have been measured on a faster computer with more RAM. Still, MDS was outperformed by our prototypic implementation of RCS. Apart from the peculiar difference for system 3, the reuse facilities implemented in MDS do not seem to provide much benefit in the case of the ATV problem instances. For system 4, MDS takes - despite the better hardware - more than 4 times as long as our prototype when deploying reuse.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The overall goal of this dissertation is to contribute to and support the further automation of standard and non-standard engineering tasks.

As available hardware and software become more and more powerful, we simultaneously witness a tremendous increase in information that may or may not be relevant for the different phases in a product's life cycle. Besides the relevance and quality of available data, its degree of formalisation decides whether it can be directly deployed in order to reveal hidden dependencies or infer further inexplicit, yet valuable knowledge.

In this context, *constraints* can be regarded as a highly formalised type of knowledge and therefore almost immediately deployable. This work does not deal with the problem of condensing informal data to formal knowledge or even constraints. Instead, we considered the automation of engineering tasks of *model-based system synthesis and analysis*, by means of *available* constraints. In this approach, the constraints capture the alternative physical behaviours of the component models which are used to assemble entire system models. Moreover, the resulting mathematical system descriptions can be used only for the analysis of *static* system characteristics; *dynamic* system behaviour has not been addressed in this work. The reader should note that this would involve much more complicated mathematics; we just mention differential equations which are usually used to capture dynamic system behaviour.

In Sect. 2.3 the common engineering-specific requirements for constraint solving techniques have been collected. The remainder of Chap. 2 makes clear that existing constraint solvers are mainly not suited for model-based engineering applications, as for instance diagnosis. One reason for that to be mentioned is the lack of reuse facilities in most existing solvers. But this is essential in large diagnosis applications where we need to solve thousands of similar constraint problem instances in an acceptable response time. Furthermore, in many solver implementations, the provision of minimal conflicts and explanations is not addressed or has only been an afterthought. But in real-world applications with hundreds and thousands of constraints, engineers will need to be provided with small or even minimal subsets

of constraints to be able to roll back to a consistent system design, or to understand a variable's solution.

Considering constraint conferences and workshops of the last few years, it is legitimate to remark that this insight has presently led to an increased interest for explanation services in the constraint computation community; cf. e.g. [3], [26], [50]. Traditionally, explanation services have been implemented either using *truth maintenance systems* - see [69] and also [16], [27] - or by propagating labeled values; [44]. This work has presented a novel approach to explanation services. This approach utilises existing aggregation trees, and in the case of linear aggregation trees, that is, deep and extremely unbalanced trees, we can recognise similarities to the method presented in [49]. Therefore, the approach presented here is more general and applicable to a greater class of problems.

However, the main reason why many existing solvers fail when confronted with engineering problems is the heterogeneity of the constraints. This is because these solvers have been tailor-made to tackle homogeneous problems, that is, problems which are built of only one type of constraints, e.g. linear ones over real-valued variables or constraints over finite-domain variables. But in many engineering applications, we are faced with heterogeneous problems that are mixed of all sensible types of constraints.

Consequently, the solution proposed in Chap. 3 had to take a very generic view on constraints: Constraints are just sets of assignments, or - equivalently - sets of solution tuples. The advantage of such an approach is that the basic operators for manipulating and combining constraints, *join* and *project*, can also be concisely be defined.¹ Based upon that very simple abstraction of constraints and the two operators *join* and *project* together with their combination *aggregate*, Chap. 3 developed a concise framework for

- deciding the consistency of a given set of constraints,
- solving, in case of consistency, all variables and
- providing one or all minimal explanations for a given variable's solutions,
- producing one or all minimal conflicts, in case of inconsistency,
- facilitating an efficient reuse when switching from one problem instance to a (similar) next one and
- enabling the import and utilisation of aggregation strategies, as e.g. obtained from decomposition tools.

Chapters 4 and 5 present an object-oriented implementation of the developed and proven results. These chapters with their class diagrams and method descriptions

¹The attentive reader will immediately recognise parallels to operators of the same name and semantics in database theory. Furthermore, the RCS-related task of determining a good aggregation strategy has much in common with *query optimisation* in database theory.

can be seen as a guideline for an actual implementation of RCS, e.g. in the programming language Java.

Whereas there are no theoretical limitations to the framework of RCS as elaborated in Chap. 3, limitations do become obvious in the context of its practical realisation in the Chaps. 4 and 5. Subsection 4.1.1 discusses this issue and shows that an implementation of RCS can always be only an approximation that will produce overestimations of the actual results. It shall however be pointed out that one may implement additional subclasses of the class `Relation` in order to push the implementation further towards “smaller” overestimations; see Th. 6. Another means to this end is the implementation of additional term rewrites; see Sect. 5.5.

Also, an implementation could include, for each method that returns an instance of `Relation`, a `boolean` attribute like `isOverestimation`. With that it would be possible to distinguish exact results from overestimations. In the current version of our prototypic implementation of RCS this has however not been done.

During the course of this dissertation and as its practical part, an implementation of RCS has been carried out mainly by the author and aided by his colleagues at a DaimlerChrysler research group. The resulting code has been tested in a few first promising engineering applications. The respective results, together with results obtained for some more academic problem families have been presented and discussed in the previous Chap. 6. Those results *prove the practicability of the presented aggregation-based constraint solver RCS*.

In the following, we shall give a more concrete list of this dissertation’s achievements and major contributions.

7.2 Major Contributions

Let us repeat the major contributionss of this dissertation, as already listed in the introduction.

- **Requirements Analysis:** A thorough requirements analysis for constraint solving in engineering applications were the starting point for this dissertation. Apart from the heterogenity of constraints in engineering, aspects of reuse along large sequences of similar constraint problem instances and explanation facilities constitute the main difference to already available constraint solvers.
- **Concise Theoretical Framework:** Based upon viewing constraints as sets of assignments, we elaborated a thorough and concise theoretical framework, the relational constraint solver RCS. The core idea - besides the afore mentioned abstraction of constraints - is to use only two operators for the manipulation and combination of constraints, *join* and *project*. With these it became possible to efficiently analyse constraint problems by building so-called *aggregation trees*. Their utilisation addresses conventional tasks, such as deciding consistency, inevitable tasks, such as reuse along problem sequences, and newly

emerged tasks, such as explaining.

In this context, the work at hand also solved the following technical problems.

- **Proof - One Minimal Conflict:** A detailed proof of a theorem concerning the computation of one minimal conflict, which had been hypothesised in the course of joint work, could be given.
 - **All Minimal Conflicts:** Likewise, a theorem concerning the provision of all minimal conflicts based on aggregation trees, and its proof were given for the first time.
 - **Explanations:** With a small modification of the conflict-related theorems, algorithms for finding one and all minimal explanations, respectively, could be presented. This dissertation also contains proofs for all hypotheses made in that context.
 - **Relationship to Decomposition Techniques:** Section 3.5 establishes the link between aggregation strategies and well-known decomposition techniques. Subsection 3.5.3 shows how *hypertree decompositions*, obtained from *constraint hypergraphs*, may be used to generate aggregation strategies that can be deployed by RCS to build aggregation trees and thereby to solve a given constraint problem.
 - **(Partially) Solved Form:** Subsection 5.2.1 introduces the concept of a *disjunctive normal form*, DNF, for relations. We argued that subsequent join and projection steps can be supported by an appropriate representation of a relation in DNF. In Subsect. 5.3, the *(partially) solved form*, (P)SF, was introduced as an advantageous DNF variant: By establishing the (P)SF, the elimination of so-called basic variables becomes a trivial task. Furthermore, finding solutions for basic variables during the backward phase is simple.
- Whereas most normal forms for representing constraints are for good reasons limited to conjunctive linear constraints, cf. e.g. [43], the (P)SF is applicable to heterogeneous constraints. Thanks to implemented methods of term manipulation, our prototype is also able to actively reformulate non-linear constraints, as opposed to passive postponing schemes, as e.g. presented in [9].
- **Implementation:** All algorithms, which result from the presented and verified theorems, have been implemented in Java and proven to be practicable in some first engineering applications.
 - **Evaluation:** Chapter 6 of this dissertation presents a showcase example that illustrates the different services of our prototypic implementation. Furthermore, several screenshots have been added, in order to provide the reader with a better understanding. The most important part of Chap. 6 provides run-times as well as structural data for the analysis of a few different problems and problem families.
- The very general *low density assumption* for model-based engineering applications, i.e. the fact that each variable appears in only few constraints, could be

shown to hold not only initially but also after several aggregations. (For that result, consider again the ATV family and see Subsect. 6.4.2.) Based on that assumption - and as forecasted in Chaps. 4 and 5 - we witnessed an almost linear dependency between a problem's size and the time consumed to solve it. However, when an initially low density grows during the forward phase, we must not expect a linear runtime behaviour, as the example of the diode circuit family $(D_k)_{k \in \mathbb{N}_+}$ shows; see Subsect. 6.2.2.

7.3 Future Work

We shall close this dissertation thesis by discussing the current status of our prototypic implementation, its shortcomings and the resulting issues that are to be investigated and dealt with in the future.

The work at hand makes a clear separation between the theoretical framework of RCS and its practical realisation by means of an actual Java implementation. The former part as well as the practical implementation as they are now are the starting point for numerous open research topics.

Concerning the theoretical part, it should be clear that - once we know which pair of relations to pick for the next aggregation - it is implicitly clear how to build aggregation trees. Likewise, all other presented algorithms which utilise and deploy existing aggregation trees, will automatically work. So, the foremost question from a theoretical point of view is not so much *how* to perform an aggregation but rather to determine *which* pair of relations to aggregate next. Note that this question has to some extent already been discussed in Sect. 3.5.

On the one hand, we have at our disposal local aggregation strategies, as the on-the-fly strategies in Subsect. 3.5.1. On the other hand, we may use known (hypertree) decomposition techniques to obtain global aggregation strategies, before the actual constraint problem is going to be solved.

An interesting problem in practice is then to decide whether precomputing a global strategy will lead to better aggregation trees², or whether we will do with a local strategy for the problem at hand. Note that such decision making will not only need to consider relations and their scopes, but also the types of relations, their mathematical structure and their constrainedness. With these attributes we will clearly leave the simplistic picture in which all constraints look alike and are just relations, i.e. sets of assignments. Instead, certain characteristics of the constraints will suddenly play a role.

Furthermore, the decision whether a global strategy will yield better results than a local one, must include problem specifics - a phenomenon that is very common for any sort of automated problem solving, and more generally in the field of *artificial intelligence*.

Going back again to the relational engine; see Chap. 4; we shall once again point out that all algorithms implemented there only assume the existence of instances of

²“Better” means here aggregation trees for which the forward relations are better approximations of the actual results, i.e. for which the forward relations are *smaller* overestimations; cf. Th. 6.

Relation and very few methods; cf. the class **Relation** in the context of Fig. 4.1. This offers a natural interface for the incorporation of existing high-performance modules which implement *join* and *project* for certain, well-studied classes of constraints. An example here is $\text{CLP}(\mathbb{R})$ which may be used to incorporate a very efficient realisation of linear constraints (including normal forms for conjunctive linear relations). Likewise, many existing finite-domain solvers that have evolved over the past two decades are candidates to be included to make our prototype fit for finite-domain (sub-)problems. For such problems, the current implementation is not very well conditioned.

The benefit of such incorporations would be that all high-level algorithms implemented at **Engine** would just as well work and e.g. provide minimal conflicts and explanations, even though the incorporated special modules do not address such issues. Clearly, the work to be done here is to implement and ensure the interface methods *join* and *project* on the side of the incorporated special modules. We need to investigate to what extent this would be possible without sacrificing the advantages of those modules.

Moreover, with the incorporation of existing high-performance modules, we encounter other, new problems. E.g. a disjunctive normal form as used in our implementation is not efficient for finite domain or boolean constraints. Here, decision diagrams, see [7], or other normal forms, see e.g. [13], and finite domain automata, see [73], offer much more efficient realisations.

To handle performance problems that are due to the usage of the disjunctive normal form, we may also introduce a new control option that allows for discarding certain disjuncts in a given disjunction, in order to manage the complexity of an otherwise unhandable problem. In diagnosis, this approach has already found some realisations, cf. minimal cardinality diagnosis in [12] and [67].

Another problem with our prototype occurs when it is fed with too many inequalities. The issue here is due to the well-known problem of redundant inequalities introduced by Fourier's algorithm. A lot of research has been done on that topic, but so far our current prototype version does only check for the simplest redundant inequalities, even though good methods are available.

We have pointed out that, in the prototype, we have the possibility to represent inequalities as equations with a slack variable; see Subsect. 5.6.1. But that will only shift the problem from redundant inequalities to the insufficient handling of slack variables: For example, whenever one tries to instantiate a constraint that only relates slack variables, our implementation will immediately replace the constraint by either \emptyset or \square . This is because to our class **Engine**, slack variables appear to be invisible. Therefore, representing inequalities by equations with slack variables is a potential source of highly overestimating results.

Similarly, when there are too many disequations in a problem, then the results produced by our prototype tend to be rough overestimations. The reason for that is that our solving schema is based on Gauss-like variable substitution or Fourier elimination. However, with disequations, both techniques become inapplicable.

Often disequations exclude just one value from a variable's domain. Hence, disequa-

tions are rather weak constraints, and our prototype often simply discards them.

In our implementation, finding a good substitute for a variable has to do with the symbolic manipulation of constraints and eventually of (algebraic) terms. One may increase the accuracy of the implementation by implementing additional term rewrites. However, this will also have drawbacks, as additional rewrites consume additional time which is spent to determine whether a given rewrite is applicable. Another interesting question here is how to find a smallest set of term rewrites that will solve a given family of constraint problems without any overestimation. In Subsect. 6.1.2 we briefly sketched a model for a quadratic resistor. So far, we have not been able to determine a smallest set of term rewrites so that any resistive network which contains only quadratic resistors, can be solved without overestimation.

Certainly, apart from finite-domain problems, integer problems play a major role in planning scenarios in engineering. Job scheduling and resource allocation problems are typically modelled using integer variables. With our current implementation, we cannot constrain a variable to take only integer values. Consequently, constraint problems of the outlined type cannot be analysed.

A further interesting question is due to floating point arithmetic and rounding errors. Although our prototype allows to constrain a real-valued variable to lie within a real interval, any computation that initially starts with real numbers will stick to numbers and never instantiate an interval. As a consequence, elaborate computations will eventually produce erroneous data. What is worse, two values that must actually coincide may turn out distinct and thus introduce a conflict where there is none. Note that this would contradict the result of Th. 6, part 2.

In the prototype, this problem has been fixed as already explained in Subsect. 5.6.1.: Whenever two numbers are compared, then they are regarded distinct only when they differ by more than some small $\epsilon > 0$. The problem with this procedure is however that we can in general not know which ϵ to choose: If it is too small, we may still introduce a false conflict; too big an ϵ will identify numbers that may actually be distinct. In the latter case the prototype will tend to produce unnecessary overestimations.

An item of future work here would be to utilise or reimplement some prominent *box consistency* approach, see e.g. [72].

Most practical problems occur when the relations become more complex, e.g. non-linear. Our solution schema is based on symbolic variable elimination. But often, there is simply no good substitute for the variable that we would like to eliminate next. A simple example is the constraint $x = \sin(x)$ with the unique solution $x = 0$. We will hardly be able to make that subtle simplification without the application of numerical methods. Such a method could for instance search all zeros of the function $f(x) = x - \sin(x)$, e.g. by deploying Newton's method.

The given equation is only a very simple example for a cyclic dependency among

a set of unknowns. Typically, there are some initial bounding boxes for these unknowns. Iteration can then help to shrink the bounding boxes and thereby narrow the solution space. Note that this is - besides symbolic elimination techniques - a second approach to finding solutions for a set of unknowns. For many engineering tasks, this alternative path is inevitable. Consequently, we need to incorporate iteration and other numerical techniques into our implementation, and maybe even in the theoretical framework of RCS.

The final item in the list of future work has been brought to our attention by another prominent engineering task. For many products engineers need to ensure a steady gap between a given pair of touching parts. A well-known example is the gap between a passenger car's door and the frame defined by the car body. Obviously, the requirement of a steady gap is not necessarily due to a product's functionality, but may also be due to a product's aesthetic parameters.

The crux with steady gaps is that all of a product's parts, e.g. the car's body and doors, will inherit from the manufacture a certain deviation from its set dimensions. Moreover, the car's assembly will introduce even more deviations. Mathematically, this can be captured by means of a probability distribution for each value, i.e. for each geometrical dimension of the parts and the relevant manufacture tools. Then, computing the door gap is basically done by combining all initial distributions to a final distribution.

Currently, this is typically done by Monte-Carlo-like simulations. Here, one does not even need to know the type of initial distributions, e.g. Gauss distribution. Obviously, a simulation could often be replaced by a well-founded computation if we knew the initial distributions and how they combine. For instance, a linear combination of Gauss-distributed values is again Gauss-distributed.

Thus, a constraint solver could *compute* distributions of interest, as opposed to naively *simulate* the physical assembly. Also, one could mathematically transpose the assembly: It would become possible to identify those parts and manufacture tools that have the greatest impact on the final distribution, i.e. on how the door gap will look. Currently, this information is often not available which makes it hard to optimise the production lines. It is very costly to decrease a part's production tolerance. Therefore it would be helpful to know *which* parts to improve first.

For our prototype, this complex problem sets a new requirement and the starting point for future work. One possibility would be to enable our implementation to attach probability distributions to certain input values and to infer the resulting distribution whenever two such values are combined.

Appendix A

Proofs

A.1 Lemma 1

\Rightarrow

Picking a tuple $(v_1, v_2, \dots, v_n) \in \text{dom}(x_1) \times \text{dom}(x_2) \times \dots \times \text{dom}(x_n)$, the task is to show that there is an assignment $\alpha^* \in \mathcal{A}$ that maps x_1, x_2, \dots, x_n to that tuple. c is nonrestrictive and therefore there must exist some $\alpha_0 \in \mathcal{A}$, by Def. 2. α_0 can gradually be altered to arrive at α^* :

For $i \in \{1, 2, \dots, n\}$ we can define $\alpha_i \in \mathcal{A}$ in the following way: Set $\alpha_i(x_j) \stackrel{\text{def}}{=} \alpha_{i-1}(x_j)$ for all $j \in \{1, 2, \dots, n\}, j \neq i$; and $\alpha_i(x_i) \stackrel{\text{def}}{=} v_i$. Since $\alpha_{i-1} \in \mathcal{A}$ by construction, and since x_i is a free variable of c , α_i must also lie in \mathcal{A} .

Eventually, $\alpha_n \equiv \alpha^*$, and $\alpha^*(x_k) = v_k$ for all $k \in \{1, 2, \dots, n\}$.

\Leftarrow

The given condition clearly implies the one given in Def. 2, stating that x_i is a free variable of c , for each $i \in \{1, 2, \dots, n\}$. Hence c is nonrestrictive by definition. ■

A.2 Lemma 2

commutativity

In the case that a or b , or both are trivial constraints, the condition is directly ensured by Def. 4. If both are non-trivial, commutativity follows from the interchangeability of X, Y and \mathcal{A}, \mathcal{B} , respectively, in the resulting non-trivial constraint.

associativity

Again, if at least one of a, b, c is a trivial constraint, both terms $a \bowtie (b \bowtie c)$ and $(a \bowtie b) \bowtie c$ can be evaluated by the computation rules provided by Def. 4, and are found to be equal for any constellation.

Let us compute $a \bowtie (b \bowtie c)$ for the non-trivial constraints $a = (X, \mathcal{A}), b = (Y, \mathcal{B})$,

and $c = (Z, \mathcal{C})$:

$$\begin{aligned}
& a \bowtie (b \bowtie c) \\
&= (X, \mathcal{A}) \bowtie (Y \cup Z, \{ \beta : Y \cup Z \rightarrow \bigcup_{v \in Y \cup Z} \text{dom}(v) : \beta|_Y \in \mathcal{B} \wedge \beta|_Z \in \mathcal{C} \}) \\
&= (X \cup Y \cup Z, \{ \gamma : X \cup Y \cup Z \rightarrow \bigcup_{v \in X \cup Y \cup Z} \text{dom}(v) : \\
&\quad \gamma|_X \in \mathcal{A} \wedge \gamma|_{Y \cup Z} \in \mathcal{M} \}), \text{ where } \mathcal{M} \text{ denotes the set of } \beta' \text{'s,} \\
&\quad \text{in the previous line,} \\
&= (X \cup Y \cup Z, \{ \gamma : X \cup Y \cup Z \rightarrow \bigcup_{v \in X \cup Y \cup Z} \text{dom}(v) : \\
&\quad \gamma|_X \in \mathcal{A} \wedge \gamma|_Y \in \mathcal{B} \wedge \gamma|_Z \in \mathcal{C} \}).
\end{aligned}$$

We obtain thus a term symmetric in (X, \mathcal{A}) , (Y, \mathcal{B}) and (Z, \mathcal{C}) . Similarly, evaluation of $(a \bowtie b) \bowtie c$ yields the same result, completing the proof. \blacksquare

A.3 Definition 6

well-defined

Checking whether the definition is coherent is done by verifying that $\emptyset \sim \square$, and $\square \sim \emptyset$ produce each the same results for both last lines of the definition. This is the case since \emptyset is unsatisfiable and restrictive, and \square is satisfiable and nonrestrictive. One obtains $\emptyset \not\sim \square$ and $\square \not\sim \emptyset$.

equivalence relation

reflexivity: Considering the definition, obviously $c \sim c$ for any constraint c .

symmetry: This is also obvious from the definition.

transitivity: Let a, b, c be three arbitrary constraints such that $a \sim b$ and $b \sim c$.

If all three are non-trivial constraints, transitivity follows from (3.3) and the transitivity of \equiv (as defined before the definition). Suppose now, at least one of a, b, c is either \emptyset or \square . The following complete case differentiation shows that \sim is transitive:

1. $b = (Y, \mathcal{B})$ is non-trivial.

(a) $a \equiv \emptyset$:

$a \sim b$, hence b is unsatisfiable, by definition, i.e., $\mathcal{B} = \emptyset$ by Def. 2. If $c = (Z, \mathcal{C})$ is also non-trivial, then $b \sim c$ and (3.3) force \mathcal{C} to be empty, too. In case, c is a trivial constraint, then $b \sim c$ enforces $c \equiv \emptyset$. In either case, c must be unsatisfiable, implying $a \sim c$.

(b) $c \equiv \emptyset$:

Symmetric to previous case.

(c) $a \equiv \square$:

This time, $a \sim b$ implies that b is nonrestrictive. Then $b \sim c$ and (3.3) yield for a non-trivial constraint $c = (Z, \mathcal{C})$ together with Lem. 1, that \mathcal{C} must contain all possible assignments to Z . Likewise, for a trivial constraint c , $c \equiv \square$ can be deduced. In any case, c must be nonrestrictive, and thus $a \sim c$.

(d) $c \equiv \square$:

Symmetric to previous case.

2. $b \equiv \emptyset$:

$a \sim b$ implies that either a is trivial with $a \equiv \emptyset$, or a has no assignment. This means that a is unsatisfiable. The same is true for c . Def. 6 gives then $a \sim c$.

3. $b \equiv \square$:

Here, $a \sim b$ enforces that either $a \equiv \square$, or a encodes all possible assignments; see Lem. 1. Again, deriving the same property for c , the conclusion $a \sim c$ follows.

This shows that \sim defines indeed an equivalence relation. ■

A.4 Lemma 3

Let (Y, \mathcal{B}) denote the result of $\bowtie C$. There is nothing to show for $|C| = 1$.

\Rightarrow

Pick some $c = (X, \mathcal{A}) \in C$, and write $(Z, \mathcal{C}) \stackrel{\text{def}}{=} \bowtie (C \setminus \{c\})$. Then

$$\begin{aligned} (Y, \mathcal{B}) &= (X, \mathcal{A}) \bowtie (Z, \mathcal{C}), \text{ by Lem. 2,} \\ &= (X \cup Z, \{\beta : X \cup Z \longrightarrow \bigcup_{v \in X \cup Z} \text{dom}(v) : \beta|_X \in \mathcal{A} \wedge \beta|_Z \in \mathcal{C}\}). \end{aligned}$$

Therefore, $\alpha \in \Sigma(C) \iff \alpha \in \mathcal{B} \implies \alpha|_X \in \mathcal{A}$.

\Leftarrow (induction over the size of C)

Let again $c = (X, \mathcal{A}) \in C$ be any of the given non-trivial constraints, and assume, for the " \Leftarrow "-part of the proof, that $\alpha|_X \in \mathcal{A}$. Furthermore, by writing again $(Z, \mathcal{C}) = \bowtie (C \setminus \{c\})$, induction provides the condition $\alpha|_Z \in \Sigma(C \setminus \{c\})$, i.e. $\alpha|_Z \in \mathcal{C}$. Combining both containment conditions, α must be "among the β 's" in the above chain of equations. It follows that $\alpha \in \mathcal{B}$. Thus, α is an assignment of $\bowtie C$, i.e. $\alpha \in \Sigma(C)$. ■

A.5 Lemma 4

All three hypotheses can easily be verified in the case where c , and one of c_1, c_2 are trivial constraints. For this, just recall the computation rules provided by Defs. 4, 5, and 6. The only exception is (3.8) in the case that $c_1 \equiv \square$. But then this assertion becomes (3.7) which is going to be shown first.

So, in what follows we can safely assume that all constraints be non-trivial.

(3.7)

Let $c = (Y, \mathcal{B})$. Then

$$\begin{aligned} (Y, \mathcal{B}) \sim \emptyset &\iff \mathcal{B} = \emptyset \\ &\stackrel{*}{\iff} \{\alpha : X \longrightarrow \bigcup_{x \in X} \text{dom}(x) : \exists \beta \in \mathcal{B} \beta|_X \equiv \alpha\} = \emptyset \\ &\iff \pi_X(c) \text{ unsatisfiable} \\ &\iff \pi_X(c) \sim \emptyset. \end{aligned}$$

In order to prove the other direction at (*), it is more obvious to show

$$\mathcal{B} \neq \emptyset \implies \left\{ \alpha : X \longrightarrow \bigcup_{x \in X} \text{dom}(x) : \exists \beta \in \mathcal{B} \beta|_X \equiv \alpha \right\} \neq \emptyset.$$

To do that consider that, for any given $\beta \in \mathcal{B}$, an $\alpha : X \rightarrow \bigcup_{x \in X} \text{dom}(x)$ can be constructed such that $\beta|_X \equiv \alpha$, just by setting $\alpha := \beta|_X$.

(3.8)

Let $c_i = (X_i, \mathcal{A}_i)$, $i \in \{1, 2\}$, and α be an assignment belonging to $c_1 \bowtie c_2$. Then, by Def. 4,

$$\alpha : X_1 \cup X_2 \rightarrow \bigcup_{x \in X_1 \cup X_2} \text{dom}(x) \quad \alpha|_{X_1} \in \mathcal{A}_1, \alpha|_{X_2} \in \mathcal{A}_2.$$

Define $\beta := \alpha|_{X_1 \cup X_2}$. Then $\beta|_{X_1} \in \mathcal{A}_1$. Also, since $X \subseteq X_2$, $\beta|_X$ is an assignment of $\pi_X(c_2)$. Hence β must be an assignment of $c_1 \bowtie \pi_X(c_2)$.

For the converse, let α belong to the set of assignments of $c_1 \bowtie \pi_X(c_2)$, assigning values to $X_1 \cup X$. Then, again by Defs. 4 and 5,

$$\alpha|_{X_1} \in \mathcal{A}_1 \wedge \exists (\beta : X_2 \rightarrow \bigcup_{x \in X_2} \text{dom}(x)) \in \mathcal{A}_2 \quad \alpha|_X \equiv \beta|_X.$$

Now, defining the assignment β' to $X_1 \cup X_2$ by setting $\beta'|_{X_2} \stackrel{\text{def}}{=} \beta$ and $\beta'|_{X_1} \stackrel{\text{def}}{=} \alpha|_{X_1}$, is coherent, because $\beta|_X \equiv \alpha|_X$ and $X \supseteq X_1 \cap X_2$. Furthermore, β' satisfies

$$\beta'|_{X_1} \equiv \alpha|_{X_1} \in \mathcal{A}_1 \wedge \beta'|_{X_2} \equiv \beta \in \mathcal{A}_2.$$

Therefore, β' is an assignment of $c_1 \bowtie c_2$.

Both previous arguments show that there is an assignment of $c_1 \bowtie c_2$ if and only if there is one of $c_1 \bowtie \pi_X(c_2)$, establishing the assertion.

(3.9)

Obviously, the sets of variables coincide on both sides (Y_2).

Let $\alpha : Y_2 \rightarrow \bigcup_{y \in Y_2} \text{dom}(y)$ first be an assignment of the left-hand side constraint. Then Defs. 4 and 5 imply

$$\exists \beta : X_1 \cup X_2 \rightarrow \bigcup_{x \in X_1 \cup X_2} \text{dom}(x) \quad \beta|_{X_1} \in \mathcal{A}_1 \wedge \beta|_{X_2} \in \mathcal{A}_2 \wedge \beta|_{Y_2} \equiv \alpha.$$

Therefore $\alpha \equiv \beta|_{Y_2}$ is clearly an assignment of $\pi_{Y_2}(c_2)$. Also, due to $Y_1 \subseteq Y_2$, α and β agree on Y_1 , and so $\alpha|_{Y_1}$ must be an assignment of $\pi_{Y_1}(c_1 \bowtie c_2)$. Summarising, α belongs to the set of assignments of the right-hand side constraint.

If α is in the right-hand side,

$$\begin{aligned} & \exists \beta : X_1 \cup X_2 \rightarrow \bigcup_{x \in X_1 \cup X_2} \text{dom}(x) \quad \beta|_{X_1} \in \mathcal{A}_1 \wedge \beta|_{X_2} \in \mathcal{A}_2 \wedge \beta|_{Y_1} \equiv \alpha|_{Y_1} \\ & \wedge \exists (\gamma : X_2 \rightarrow \bigcup_{x \in X_2} \text{dom}(x)) \in \mathcal{A}_2 \quad \gamma|_{Y_2} \equiv \alpha. \end{aligned}$$

Especially, $\beta|_{Y_1} \equiv \alpha|_{Y_1} \equiv \gamma|_{Y_1}$, since $Y_1 \subseteq Y_2$. Since $Y_1 \supseteq X_1 \cap X_2$, this means that $\beta' : X_1 \cup X_2 \rightarrow \bigcup_{x \in X_1 \cup X_2} \text{dom}(x)$, defined by $\beta'|_{X_1} \stackrel{\text{def}}{=} \beta|_{X_1}$ and $\beta'|_{X_2} \stackrel{\text{def}}{=} \gamma$ is coherent. By construction, $\beta'|_{X_i} \in \mathcal{A}_i$, for $i \in \{1, 2\}$, and moreover $\beta'|_{Y_2} \equiv \gamma|_{Y_2} \equiv \alpha$, because $Y_2 \subseteq X_2$. This proves that α is an assignment of $\pi_{Y_2}(c_1 \bowtie c_2)$. ■

A.6 Lemma 5

existence part of statement 1.

Fix a variable x mentioned by some $c \in C$, i.e. $x \in c$. Let the path to c from the root ρ of Δ be denoted by

$$c \equiv c_1 \longleftarrow c_2 \longleftarrow \dots \longleftarrow c_n \equiv \rho.$$

Then obviously $x \notin \text{vars}(c_n) \wedge x \in \text{vars}(c_1)$. Therefore the index $k_0 \stackrel{\text{def}}{=} \min\{k \in \{2, 3, \dots, n\} : x \notin \text{vars}(c_k)\}$ is well-defined. With d denoting the second successor of c_{k_0} , this gives $x \in (\text{vars}(c_{k_0-1}) \cup \text{vars}(d)) \setminus \text{vars}(c_{k_0})$. Hence c_{k_0} eliminates x , in the sense of 1. of Lem. 5, proving the existence of such a node.

statement 2.

c_{k_0} is the aggregate of c_{k_0-1} and d . And so, by condition 2. of Def. 10,

$$x \notin \text{int}(c_{k_0-1} \bowtie d, C \setminus \Lambda(c_{k_0})).$$

This implies that, for all $c \in C \setminus \Lambda(c_{k_0})$, $x \notin \text{vars}(c)$. Clearly, this is equivalent to statement 2.

uniqueness part of statement 1.

Suppose there were two distinct nodes $\epsilon_i(x)$, $i \in \{1, 2\}$, eliminating x . Then 2. implies the condition

$$\Lambda(\epsilon_1(x)) \cap \Lambda(\epsilon_2(x)) \neq \emptyset.$$

But in a tree, this means that, without loss of generality, $\Lambda(\epsilon_1(x)) \subseteq \Lambda(\epsilon_2(x))$; and because of distinctness of the two nodes even a proper subset relation, \subset .

Now the defining property of $\epsilon_1(x)$ ensures that x appear only "below" $\epsilon_1(x)$. But as $\epsilon_2(x)$ is "above" $\epsilon_1(x)$, this means that none of the successors of $\epsilon_2(x)$ must mention x which clearly contradicts the fact that $\epsilon_2(x)$ eliminates x . Hence, there cannot be more than one node in Δ , eliminating x .

statement 3.

This is a simple consequence of the property of $\epsilon(x)$: Starting at each leaf $c \in C$ for which $x \in \text{vars}(c)$, x has to appear in any node on the path that leads to c from $\epsilon(x)$. Thus, all nodes mentioning x must form a subtree rooted at $\epsilon(x)$. Only, that node does clearly not mention x , by its definition.

statement 4.

Let x be any variable belonging to the left-hand side intersection. Then $\epsilon(x)$ cannot lie "below" r , that is, $\epsilon(x)$ must lie somewhere on the path to r from the root of Δ . The previous statement 3. ensures then that x appears also in both r_1 and r_2 . This establishes the claimed set inclusion.

statements 5.(a) and 5.(b) (by bottom-up induction on Δ)

Suppose first that r is an arbitrary leaf of Δ . Then 5.(a) simplifies to an instantiation of (3.2). Moreover, in that case, there is nothing to show for 5.(b)

For the induction proof, assume now that r is any non-leaf node. The plan is to show, for the node r , 5.(b) first. This bit of the proof will assume the validity of 5.(a) for both r_1 and r_2 . The induction step concludes by proving 5.(a) for r . As

soon as this is done, the induction principle ensures the validity of both 5.(a) and 5.(b) on the entire tree Δ .

So given $r_i \equiv \pi_{vars(r_i)}(\bowtie \Lambda(r_i))$, $i \in \{1, 2\}$, 5.(b) will be shown to hold for r . Obviously, variable sets coincide on both sides of the equation; hence the following argument will just consider assignments.

Write $s_i \stackrel{def}{=} \bowtie \Lambda(r_i)$, $X_i \stackrel{def}{=} vars(r_i)$, $i \in \{1, 2\}$, and let $\alpha : X_1 \cup X_2 \rightarrow \bigcup_{x \in X_1 \cup X_2} dom(x)$ first be any assignment of

$$r_1 \bowtie r_2 \equiv \pi_{X_1}(s_1) \bowtie \pi_{X_2}(s_2).$$

Then, there are assignments β_i , $i \in \{1, 2\}$ of s_i such that $\beta_i|_{X_i} \equiv \alpha|_{X_i}$, and the β_i must agree on $X_1 \cap X_2$. Statement 4. ensures that the β_i must hence agree on $vars(s_1) \cap vars(s_2)$. Therefore, defining β on the set of variables $vars(s_1) \cup vars(s_2)$, according to $\beta|_{vars(s_i)} \equiv \beta_i$, is coherent.

By construction, β is an assignment of $s_1 \bowtie s_2 = \bowtie \Lambda(r)$. Moreover, for $i \in \{1, 2\}$, $\beta|_{X_i} \equiv \beta_i|_{X_i} \equiv \alpha|_{X_i}$, since $X_i \subseteq vars(s_i)$. So, β and α agree on $X_1 \cup X_2$. Summarising, this proves that α is an assignment of $\pi_{X_1 \cup X_2}(\bowtie \Lambda(r))$.

Let, for the converse, now α be an assignment of $\pi_{X_1 \cup X_2}(\bowtie \Lambda(r))$.

Clearly, by decreasing the set of joined constraints, α must also be an assignment of $\pi_{X_1 \cup X_2}(\bowtie \Lambda(r_i))$, $i \in \{1, 2\}$. Decreasing now the set of variables, one finds that $\alpha|_{X_i}$ is an assignment of $\pi_{X_i}(\bowtie \Lambda(r_i))$, for $i \in \{1, 2\}$. Hence α has been found to be an assignment of $r_1 \bowtie r_2$.

This concludes the induction step concerning 5.(b).

Given the validity of 5.(b) for the non-leaf node r , it is easy to derive that of 5.(a): Since r is the aggregate of r_1 and r_2 , applying the projection onto $vars(r)$ on both sides of 5.(b) immediately yields 5.(a).

statement 6

Write $c_1 \stackrel{def}{=} \bowtie M$ and $c_2 \stackrel{def}{=} r_1 \bowtie r_2$, then the goal is to apply (3.8). Choose X according to Def. 10, 2., i.e. such that $r = agg_X(r_1, r_2)$. Then obviously, $X \supseteq (int(r_1, M) \cup int(r_2, M))$, due to $M \subseteq C \setminus \Lambda(r)$. And Def. 9 implies $X \supseteq vars(c_1) \cap (vars(r_1) \cup vars(r_2)) = vars(c_1) \cap vars(c_2)$. Thus, the precondition of (3.8) holds indeed, and the assertion follows.

statement 7

Let $M' \in \tau(M)$ be arbitrary such that $\bowtie M' \sim \emptyset$ holds. Let

$$M'' \stackrel{def}{=} \bigcup_{m \in M} \Lambda(m)$$

denote the element of $\tau(M)$ with the most nodes. Then $M'' \supseteq M'$, and therefore $\bowtie M'' \sim \emptyset$. But now we can repeatedly apply statement 6 until all leaves in M'' have been replaced by the appropriate subtree root in M . This proves the condition $\bowtie M \sim \emptyset$. ■

A.7 Lemma 6

statement 1 (by top-down induction)

For $r = \rho$, the only possible set M is, by Def. 14, the empty set of nodes of Δ . But due

to inconsistency of C , Th. 1 provides $\rho \equiv \emptyset$, and therefore $\bowtie(M \cup \{r\}) = \bowtie(\{\emptyset\}) \equiv \emptyset$, i.e. the assertion.

Let now r be any node with successors r_1, r_2 , and let us assume the assertion be satisfied for r . Let also $M_i \in \text{con}_\downarrow(r_i)$ be arbitrary for $i \in \{1, 2\}$.

If $\bowtie(M_i \cup \{r_i\}) \sim \emptyset$ (in order to consider the first case of Def. 14), then the assertion holds obviously also for r_i .

If $M_i = M \cup \{r_{3-i}\}$, where $M \in \text{con}_\downarrow(r)$, then $\bowtie(M \cup \{r\}) \sim \emptyset$ by induction. According to Lem. 5, 6., this is equivalent to $\bowtie(M \cup \{r_1, r_2\}) \sim \emptyset$, i.e. again $\bowtie(M_i \cup \{r_i\}) \sim \emptyset$, because $\{r_1, r_2\} = \{r_i, r_{3-i}\}$ for both $i = 1$ and $i = 2$.

statement 2 (by bottom-up induction)

Consider first any leaf $\lambda \in \Lambda(\rho)$. Here, the assertion follows immediately from Def. 14 and statement 1.

Assume now the assertion holds for both successors r_1, r_2 of a non-leaf node r . Let $M \in \text{con}_\uparrow(r)$ be arbitrary.

If $M \in \text{con}_\uparrow(r_i)$ for $i = 1$ or $i = 2$ (see Def. 14, then $\bowtie M \sim \emptyset$ by induction. If, otherwise, M is composed of some M_1 and M_2 as elaborated in Def. 14, then $\bowtie M \sim \emptyset$ is explicitly guaranteed, and there is nothing to show either.

statement 3

This property follows immediately from Def. 14: When defining $\text{con}_\downarrow(r)$, only the brother node r' of r will be inserted into the respective sets of nodes. (r, r' are brothers if they have the same predecessor.) Moreover, no successor of r itself will be inserted.

statement 4 (by bottom-up induction)

Consider first any leaf λ . By Def. 14, $M \in \text{con}_\uparrow(\lambda)$ implies that $\lambda \in M$, and hence $M \cap \Lambda(\lambda) = \{\lambda\} \neq \emptyset$. The first implication becomes trivial for leaves since $V(\Delta(\lambda)) = \Lambda(\lambda)$. The second implication is an obvious consequence of Def. 14 and statement 3.

Let, as above, r be any non-leaf node with successors r_1, r_2 ; $M \in \text{con}_\uparrow(r)$ be arbitrary, and assume the validity of the assertion for both r_1 and r_2 .

Suppose first the case that $M \in \text{con}_\uparrow(r_i)$, for either $i = 1$ or $i = 2$, and $r_{3-i} \notin M$. $M \in \text{con}_\uparrow(r_i)$ and the induction assumption imply that $M \cap V(\Delta(r))$ contains only leaves of $\Delta(r_i)$ and maybe r_{3-i} . But the latter node is known not to lie in M . Thus, $M \cap V(\Delta(r))$ consists exclusively of leaves of $\Delta(r_i)$.

For any $n \in M \setminus V(\Delta(r))$ the stated property follows from $M \in \text{con}_\uparrow(r_i)$ and the fact that the property in question is assumed to hold already for r_i .

The remaining case is the one in which M is composed of some M_1, M_2 as stated in Def. 14. Here, the argument is similar since the structural properties hold for M_1, M_2 by assumption. r_1 may belong to M_2 (from which it is removed in M) but not to M_1 . Likewise, r_2 may belong to M_1 (from which it is removed in M) but not to M_2 . Summarising, $M \cap V(\Delta(r))$ must be a subset of $\Lambda(r)$. For $n \in M \setminus V(\Delta(r))$, the assertion follows again from what we already know to hold for M_1 and M_2 .

statement 5

This is the application of statement 4 to the node $r = \rho(\Delta)$.

statement 6 (by top-down induction on Δ)

For ρ , the assertion follows easily, since the only possible choice $M = \emptyset$ gives $\tau(M \cup \{\rho\}) = \tau(\{\rho\})$ which contains all non-empty sets of leaves of $\Delta(\rho)$ and hence each minimal conflict of C .

Let now r be any non-leaf node, for which the assertion may be assumed by induction, with successors r_1, r_2 . Note that there is something to show only if K is a minimal conflict that meets $\Lambda(r_i)$, for $i = 1$ or $i = 2$ or both. Either way, we have $K \cap \Lambda(r) \neq \emptyset$, and so, by induction assumption, there must exist $M \in \text{con}_\downarrow(r)$ such that $K \in \tau(M \cup \{r\})$.

So let us first consider the case that $K \cap \Lambda(r_i) \neq \emptyset$, for $i = 1$ or $i = 2$ but not for both. Then we need to find $M_i \in \text{con}_\downarrow(r_i)$ such that $K \in \tau(M_i \cup \{r_i\})$. It is clear that actually $K \in \tau(M \cup \{r_i\})$ since K does not contain leaves of $\Delta(r_{3-i})$. By Def. 13, $\bowtie K \sim \emptyset$, and thus application of Lem. 5, 7., yields $\bowtie(M \cup \{r_i\}) \sim \emptyset$. Now, this property together with $M \in \text{con}_\downarrow(r)$ show that $M \in \text{con}_\downarrow(r_i)$, by Def. 14. Hence $M_i \stackrel{\text{def}}{=} M$ is the set we needed to find.

In the remaining case, K contains leaves of both $\Delta(r_1)$ and $\Delta(r_2)$, i.e. $K \in \tau(M \cup \{r_1, r_2\})$. Defining $M_i \stackrel{\text{def}}{=} M \cup \{r_{3-i}\}$, $i \in \{1, 2\}$, produces elements of $\text{con}_\downarrow(r_i)$, by Def. 14. Moreover, $K \in \tau(M_i \cup \{r_i\})$ for both $i = 1$ and $i = 2$. This shows the assertion for both nodes r_1, r_2 .

statement 7 (by bottom-up induction on Δ)

The induction begins with the case of a leaf λ , and a minimal conflict K that contains λ . Statement 6 implies the existence of $M \in \text{con}_\downarrow(\lambda)$ such that $K \in \tau(M \cup \{\lambda\})$. But then $M \cup \{\lambda\} \in \text{con}_\uparrow(\lambda)$ by Def. 14, and we are done.

Let again r be any non-leaf node with successors r_1, r_2 , for which the assertion can be assumed to hold by induction. And be K a minimal conflict that meets $\Lambda(r)$. Consider first the case in which $K \cap \Lambda(r_i) \neq \emptyset$ but $K \cap \Lambda(r_{3-i}) = \emptyset$, for either $i = 1$ or $i = 2$. By assumption, there must exist $M_i \in \text{con}_\uparrow(r_i)$ with $K \in \tau(M_i)$. Also, M_i does not contain r_{3-i} for otherwise K would have to contain a leaf of $\Delta(r_{3-i})$. Definition 14 implies then that $M_i \in \text{con}_\uparrow(r)$.

It remains the case in which K contains leaves of both $\Delta(r_1)$ and $\Delta(r_2)$. Then, by induction, there exist sets $M_i \in \text{con}_\uparrow(r_i)$, $i \in \{1, 2\}$, such that $K \in \tau(M_i)$. Moreover, we must have $r_1 \in M_2$ and $r_2 \in M_1$. Define $M \stackrel{\text{def}}{=} (M_1 \setminus \{r_2\}) \cup (M_2 \setminus \{r_1\})$; then we are done if we can show $\bowtie M \sim \emptyset$, cf. Def. 14.

Due to $K \in \tau(M_1) \cap \tau(M_2)$, and the structures of M_1, M_2 implied by statement 4, we obtain $M_1 \setminus V(\Delta(r)) = M_2 \setminus V(\Delta(r))$ and both $M_1 \cap V(\Delta(r))$ and $M_2 \cap V(\Delta(r))$ consist exclusively of leaves. Together, these observations give $K \in \tau(M)$. But Def. 13 guarantees $\bowtie K \sim \emptyset$ and the application of Lem. 5, 7., turns this into $\bowtie M \sim \emptyset$ which was to be shown. ■

A.8 Lemma 7

statement 1

All equivalences given in statement 1 are immediate consequences of Def. 15.

statement 2

Again, this is easily verified by taking a look at Def. 15 and the definition of $E_l(\rho)$. Note that $\{\rho\}$ is indeed a minimal conflict of $V(\Delta)$, since $\rho \equiv \emptyset$.

statement 3(a) (by bottom-up induction on Δ)

The assumption and statement 1 guarantee, for all parts of the proof concerning statement 3, that $\widehat{con}_l(s) \neq \star$ and $\widehat{con}_r(s) \neq \star$.

For any leaf λ , Def. 15 says that $\widehat{con}_r(\lambda) = \widehat{con}_l(\lambda) \cup \{\lambda\}$. Therefore, the assertion follows.

Let now r be any non-leaf node with successors r_1, r_2 for which the assertion may be assumed, by induction. By Def. 15, there are two distinct cases:

case 1: $\widehat{con}_l(r_i) = \widehat{con}_l(r) \wedge \widehat{con}_l(r_{3-i}) = \star$, for either $i = 1$ or $i = 2$. Then, by mapping \star 's according to statement 1, $\widehat{con}_r(r) = \widehat{con}_r(r_i)$. By induction, we may assume $\widehat{con}_l(r_i) \setminus V(\Delta(r_i)) = \widehat{con}_r(r_i) \setminus V(\Delta(r_i))$. In this equation, we may replace $V(\Delta(r_i))$ by the superset $V(\Delta(r))$. Hence, after furthermore omitting the index i , due to known equalities, we arrive at the assertion stated for the node r .

case 2: $\widehat{con}_l(r_1) = \widehat{con}_l(r) \cup \{r_2\} \wedge \widehat{con}_l(r_2) = \widehat{con}_r(r_1) \setminus \{r_2\}$. In this case, Def. 15 implies $\widehat{con}_r(r) = \widehat{con}_r(r_2)$. Let us refer to those three conditions by (x), (y) and (z), respectively. Then, we obtain

$$\begin{aligned}
 \widehat{con}_r(r) \setminus V(\Delta(r)) &= \widehat{con}_r(r_2) \setminus V(\Delta(r)), && \text{by (z),} \\
 &= \widehat{con}_l(r_2) \setminus V(\Delta(r)), && \text{by assumption, and} \\
 &= && V(\Delta(r_2)) \subseteq V(\Delta(r)), \\
 &= \widehat{con}_r(r_1) \setminus V(\Delta(r)), && \text{due to (y),} \\
 &= \widehat{con}_l(r_1) \setminus V(\Delta(r)), && \text{again by assumption, and} \\
 &= && V(\Delta(r_1)) \subseteq V(\Delta(r)), \\
 &= \widehat{con}_l(r) \setminus V(\Delta(r)), && \text{by (x).}
 \end{aligned}$$

statement 3(b) (by top-down induction on Δ)

The assertion is true for the root ρ , due to Def. 15.

Let again r, r_1, r_2 be as above. We need to show 3(b) for both r_1, r_2 assuming that $\widehat{con}_l(r) \neq \star$ and $\widehat{con}_l(r) \cap V(\Delta(r)) = \emptyset$.

We must either have $\widehat{con}_l(r_1) = \widehat{con}_l(r)$ or $\widehat{con}_l(r_1) = \widehat{con}_l(r) \cup \{r_2\}$, so in either case $\widehat{con}_l(r_1) \cap V(\Delta(r)) \subseteq \{r_2\}$. Therefore $\widehat{con}_l(r_1) \cap V(\Delta(r_1)) = \emptyset$, and thus the condition follows for r_1 .

The argument is a bit more complicated for r_2 .

case 1: $\widehat{con}_l(r_2) = \widehat{con}_l(r)$. Then $\widehat{con}_l(r_2) \cap V(\Delta(r_2)) \subseteq \widehat{con}_l(r) \cap V(\Delta(r)) = \emptyset$, and we are done.

case 2: $\widehat{con}_l(r_2) = \widehat{con}_r(r_1) \setminus \{r_2\}$. This is only then the case if $\widehat{con}_l(r_1) = \widehat{con}_l(r) \cup \{r_2\}$; see Def. 15. Since $\widehat{con}_l(r)$ does not meet $V(\Delta(r))$, we know thus that the only node of $\widehat{con}_l(r_1)$ in $V(\Delta(r))$ is r_2 . But then, by 3(a), $(\widehat{con}_r(r_1) \setminus V(\Delta(r_1))) \cap V(\Delta(r)) = \{r_2\}$. This means that $\widehat{con}_l(r_2) = \widehat{con}_r(r_1) \setminus \{r_2\}$ must not mention a node of $\Delta(r_2)$.

statement 3(c) (by bottom-up induction on Δ)

For any leaf λ , we have $V(\Delta(\lambda)) = \Lambda(\lambda) (= \{\lambda\})$, and there is nothing to show.

Let now r, r_1, r_2 be as above. Then we have again two cases, according to Def. 15.

case 1: $\widehat{\text{con}}_{\uparrow}(r) = \widehat{\text{con}}_{\uparrow}(r_i) \wedge \widehat{\text{con}}_{\uparrow}(r_{3-i}) = \star$. Here, recalling the proof of 3(a), we must have $\widehat{\text{con}}_{\downarrow}(r) = \widehat{\text{con}}_{\downarrow}(r_i)$. Suppose there were a node $n \in \widehat{\text{con}}_{\uparrow}(r)$ that also lies in $V(\Delta(r_{3-i})) \cup \{r\}$. Then $n \in \widehat{\text{con}}_{\uparrow}(r_i)$; and 3(a) implies thus $n \in \widehat{\text{con}}_{\downarrow}(r_i) = \widehat{\text{con}}_{\downarrow}(r)$. But this clearly contradicts 3(b). Hence, such an n cannot exist, and $\widehat{\text{con}}_{\uparrow}(r) \cap V(\Delta(r)) \subseteq V(\Delta(r_i))$. According to $\widehat{\text{con}}_{\uparrow}(r) = \widehat{\text{con}}_{\uparrow}(r_i)$ and the assumption for r_i , that intersection can only contain leaves in $\Lambda(r_i) \subseteq \Lambda(r)$; and we are done.

case 2: $\widehat{\text{con}}_{\uparrow}(r) = \widehat{\text{con}}_{\uparrow}(r_2)$, and $\widehat{\text{con}}_{\uparrow}(r_i) \neq \star$, for $i \in \{1, 2\}$. Then, Def. 15 dictates $\widehat{\text{con}}_{\downarrow}(r_1) = \widehat{\text{con}}_{\downarrow}(r) \cup \{r_2\}$ and $\widehat{\text{con}}_{\downarrow}(r_2) = \widehat{\text{con}}_{\uparrow}(r_1) \setminus \{r_2\}$.

Now, what are the nodes in $\widehat{\text{con}}_{\uparrow}(r) \cap V(\Delta(r))$?

Taking $\widehat{\text{con}}_{\downarrow}(r_1) = \widehat{\text{con}}_{\downarrow}(r) \cup \{r_2\}$ together with 3(b) applied to r , we obtain $\widehat{\text{con}}_{\downarrow}(r_1) \cap V(\Delta(r)) = \{r_2\}$. Consequently, 3(a) and 3(c) applied to r provide us with the inclusion $\widehat{\text{con}}_{\uparrow}(r_1) \cap V(\Delta(r)) \subseteq \Lambda(r_1) \cup \{r_2\}$. Therefore $\widehat{\text{con}}_{\downarrow}(r_2) = \widehat{\text{con}}_{\uparrow}(r_1) \setminus \{r_2\}$ consists - inside $V(\Delta(r))$ - exclusively of leaves in $\Lambda(r_1)$. 3(a) and 3(c), this time applied to r_2 , yield now $\widehat{\text{con}}_{\uparrow}(r_2) \cap V(\Delta(r)) \subseteq \Lambda(r)$. But this provides us with the claim, because $\widehat{\text{con}}_{\uparrow}(r) = \widehat{\text{con}}_{\uparrow}(r_2)$.

statement 4

Suppose $\widehat{\text{con}}_{\downarrow}(r_1) \neq \star \wedge E_{\downarrow}(r)$. According to Def. 15, there are the two following cases:

case 1: $\widehat{\text{con}}_{\downarrow}(r_1) = \widehat{\text{con}}_{\downarrow}(r) \wedge \boxtimes(\widehat{\text{con}}_{\downarrow}(r) \cup \{r_1\}) \sim \emptyset$. Here, we need to show that no constraint in $\widehat{\text{con}}_{\downarrow}(r_1) \cup \{r_1\}$ can be suspended. Due to $E_{\downarrow}(r)$, we know that $\boxtimes \widehat{\text{con}}_{\downarrow}(r_1) \not\sim \emptyset$, and so r_1 cannot be suspended.

The following argument proves that neither may $s \in \widehat{\text{con}}_{\downarrow}(r_1)$ be suspended:

$$\begin{aligned} E_{\downarrow}(r) &\implies \boxtimes((\widehat{\text{con}}_{\downarrow}(r) \setminus \{s\}) \cup \{r\}) \not\sim \emptyset \\ &\implies \boxtimes((\widehat{\text{con}}_{\downarrow}(r) \setminus \{s\}) \cup \{r_1, r_2\}) \not\sim \emptyset, \quad \text{by 3(b) and Lem. 5, 6,} \\ &\implies \boxtimes((\widehat{\text{con}}_{\downarrow}(r_1) \setminus \{s\}) \cup \{r_1\}) \not\sim \emptyset, \quad \text{by suspending } r_2. \end{aligned}$$

case 2: $\widehat{\text{con}}_{\downarrow}(r_1) = \widehat{\text{con}}_{\downarrow}(r) \cup \{r_2\} \wedge \boxtimes(\widehat{\text{con}}_{\downarrow}(r) \cup \{r_i\}) \not\sim \emptyset$, for $i \in \{1, 2\}$. This time

$$\begin{aligned} E_{\downarrow}(r) &\implies \boxtimes(\widehat{\text{con}}_{\downarrow}(r) \cup \{r\}) \sim \emptyset \\ &\iff \boxtimes(\widehat{\text{con}}_{\downarrow}(r) \cup \{r_1, r_2\}) \sim \emptyset, \quad \text{by 3(b) and Lem. 5, 6,} \\ &\iff \boxtimes(\widehat{\text{con}}_{\downarrow}(r_1) \cup \{r_1\}) \sim \emptyset \end{aligned}$$

shows that $\widehat{\text{con}}_{\downarrow}(r_1) \cup \{r_1\}$ is inconsistent.

By the two additional conditions that hold in case 2, we know that neither r_1 nor r_2 can be suspended.

Furthermore, suspension of $s \in \widehat{\text{con}}_{\downarrow}(r)$ can be shown to fail exactly as in case 1; only this time, the last implication is not due to suspension but due to reformulation using $\widehat{\text{con}}_{\downarrow}(r_1) = \widehat{\text{con}}_{\downarrow}(r) \cup \{r_2\}$.

statement 5

The proof is symmetric to that of statement 4, case 1.

statement 6

This follows immediately from the definitions of $E_{\downarrow}(\lambda)$, $E_{\uparrow}(\lambda)$, and Def. 15.

statement 7

The precondition and Def. 15 imply $\widehat{con}_{\uparrow}(r) = \widehat{con}_{\uparrow}(r_1)$. The assertion follows immediately.

statement 8

Here, we know that $\widehat{con}_{\uparrow}(r) = \widehat{con}_{\uparrow}(r_2)$, and we are done.

statement 9

The precondition and Def. 15 yield the conditions $\widehat{con}_{\downarrow}(r_2) = \widehat{con}_{\uparrow}(r_1) \setminus \{r_2\}$ and $r_2 \in \widehat{con}_{\downarrow}(r_1)$. Thus, by 3(a), $r_2 \in \widehat{con}_{\uparrow}(r_1)$. Consequently,

$$\widehat{con}_{\downarrow}(r_2) \cup \{r_2\} = (\widehat{con}_{\uparrow}(r_1) \setminus \{r_2\}) \cup \{r_2\} = \widehat{con}_{\uparrow}(r_1).$$

This equality shows that $E_{\uparrow}(r_1)$ implies $E_{\downarrow}(r_2)$.

statement 10

Let us show, by bottom-up induction on Δ , that

$$\forall r \in V(\Delta(\rho)) \quad E_{\downarrow}(r) \implies E_{\uparrow}(r).$$

Then, $E_{\uparrow}(\rho)$ follows from statement 2.

Statement 6 shows the assertion for all leaves.

Let now r, r_1, r_2 be as before. We need to show the assertion for r and may assume that it holds for r_1, r_2 . We may furthermore constrain ourselves to the case $\widehat{con}_{\downarrow}(r) \neq \star$, for otherwise $\widehat{con}_{\downarrow}(r) = \widehat{con}_{\uparrow}(r) = \star$, by statement 1, and there is nothing to show. Definition 15 implies that then not both of $\widehat{con}_{\downarrow}(r_1)$ and $\widehat{con}_{\downarrow}(r_2)$ equal \star . So, there remain three cases:

case 1: $\widehat{con}_{\downarrow}(r_1) \neq \star \wedge \widehat{con}_{\downarrow}(r_2) \neq \star$. Statement 4 yields the validity of $E_{\downarrow}(r_1)$, and so $E_{\uparrow}(r_1)$ by induction. Now, statement 9 applies, and we obtain $E_{\downarrow}(r_2)$, and $E_{\uparrow}(r_2)$ by induction. Finally, statement 8 produces $E_{\uparrow}(r)$.

case 2: $\widehat{con}_{\downarrow}(r_1) = \star \wedge \widehat{con}_{\downarrow}(r_2) \neq \star$. Statement 5 gives $E_{\downarrow}(r_2)$, and hence $E_{\uparrow}(r_2)$ by induction. Again, statement 8 yields $E_{\uparrow}(r)$.

case 3: $\widehat{con}_{\downarrow}(r_1) \neq \star \wedge \widehat{con}_{\downarrow}(r_2) = \star$. Statement 4 yields $E_{\downarrow}(r_1)$, and - as above - $E_{\uparrow}(r_1)$. We have also $\widehat{con}_{\uparrow}(r_2) = \star$, by statement 1. Therefore, statement 7 applies to give us $E_{\uparrow}(r)$, and we are done. ■

A.9 Lemma 8**statement 1:**

It is sufficient to verify that the union of all leaves in Φ_1 equals C_1 .

Step 1 of the given procedure ensures that no constraint of $C_0 \setminus C_1$ is going to belong to Φ_1 . Step 2 does not mark any leaf node. So, steps 1 and 2 guarantee that Φ_1 contain $C_0 \cap C_1$. The newly-introduced constraints in $C_1 \setminus C_0$ will explicitly be inserted into Φ_1 in step 3. Hence, the set of leaves in Φ_1 equals C_1 .

statement 2:

Suppose two trees $\Delta_1, \Delta_2 \in \Phi_1$ mention the same variable x . There are three cases.

case 1: Both Δ_1, Δ_2 are just leaf nodes of $C_1 \setminus C_0$. Then, there is nothing to show since those leaf nodes coincide with the two respective root nodes.

case 2: One, say Δ_1 , is just a leaf constraint of $C_1 \setminus C_0$; and Δ_2 is a subtree of Φ_0 . Then, since x appears in Δ_1 , Δ_2 must not contain $\epsilon(x)$ due to step 2 of the given procedure. Therefore, x must also appear in $\rho(\Delta_2)$.

case 3: Both Δ_1 and Δ_2 are subtrees of Φ_0 . Then $\Lambda(\rho(\Delta_i))$, $i \in \{1, 2\}$, are connected sets, in the sense of Def. 10. Moreover, since x is mentioned in both trees, even $\Lambda(\rho(\Delta_1)) \cup \Lambda(\rho(\Delta_2))$ must be connected. Therefore, Δ_1 and Δ_2 must be subtrees of the *same* aggregation tree in Φ_0 . But then, due to the connectedness condition for x in Φ_0 , $\epsilon(x)$ cannot be a node in $V(\Delta_1) \cup V(\Delta_2)$. But that means that x must neither be eliminated in Δ_1 nor in Δ_2 ; and x must thus also be mentioned in $\rho(\Delta_1)$ and $\rho(\Delta_2)$. ■

A.10 Lemma 9

According to the argumentation around (3.18), it is sufficient to prove that $\Theta(\rho(\Upsilon))$ is an aggregation tree for some constraint problem $C \cup D \cup P$, where D is empty or contains copies of some constraints in C and P is a possibly empty set of projections of constraints in C .

plan of the proof:

The proof first constructs a generic aggregation strategy Υ' from the DAG Υ , that is a tree. The leaves of that tree will be the constraints C and copies D , as above. Also, $\Theta'(\cdot)$ and $X'(\cdot)$ will be defined for any node of Υ' , such that $\Theta'(n) = \Theta(n)$ for all cluster nodes n . The cluster nodes of Υ and Υ' are going to coincide.

In a second step, a tree Υ'' will be obtained by altering Υ' . $\Lambda(\Upsilon'')$ is going to be $C \cup D \cup P$, where D and P are as above. Again, the cluster nodes of Υ' and Υ'' will be identical. Together with a translation function Θ'' , Υ'' will be proved to be a generic aggregation strategy for $C \cup D \cup P$, and $\Theta''(\rho(\Upsilon''))$ be shown to be an aggregation tree for that constraint problem. Θ'' and Θ' are going to agree for all cluster nodes.

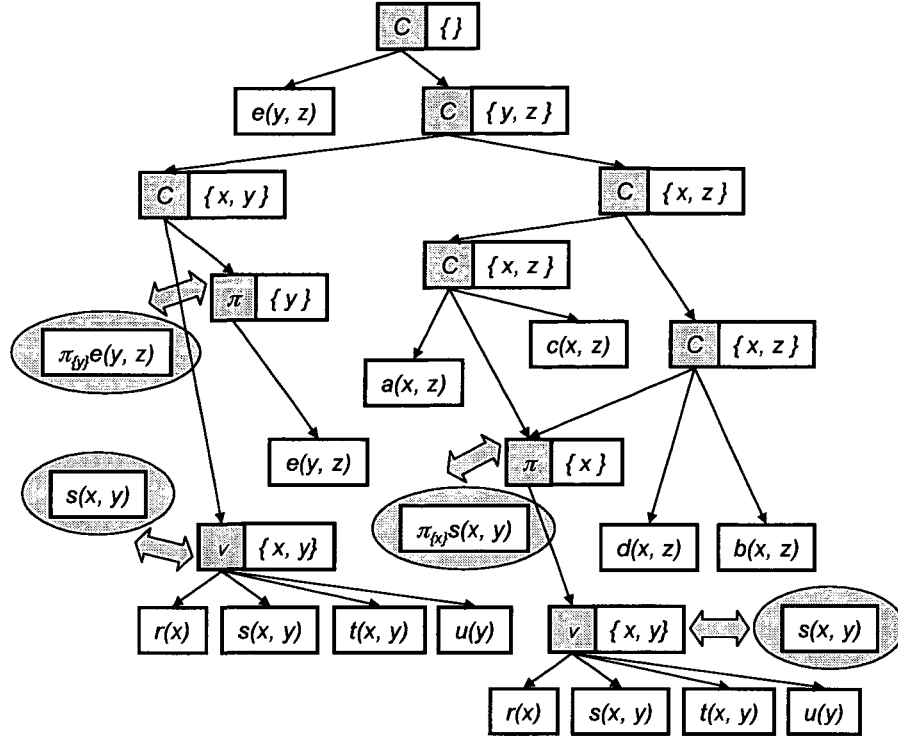
The prove concludes by showing that $\rho = \rho(\Upsilon)$ is a cluster node that is also the root of Υ' and Υ'' . Then, we have by construction, $\Theta''(\rho) = \Theta'(\rho) = \Theta(\rho)$. Therefore, $\Theta(\rho)$ is shown to be an aggregation tree for $C \cup D \cup P$, and we are done.

Figure A.1 depicts the generic aggregation strategies Υ' and Υ'' , derived from Υ as presented in Fig. 3.16: Ignoring the “bubble” nodes, we can view Υ' . Υ'' is the same as Υ' , except that the “bubble” nodes replace the corresponding subtrees.

construction of Υ' :

Beginning at the leaves, and going bottom-up, we stop at each node n of Υ that has more than one predecessor. It can be assumed, by bottom-up induction, that n together with all nodes m , for which a path $n \xrightarrow{*} m$ exists, already form a tree. We attach a copy of that tree to each of the predecessors of n .

Note that no cluster node n of Υ needs to be copied since n has at most one predecessor, by Def. 18, 5. Moreover, we simply define $\Theta'(n') \stackrel{\text{def}}{=} \Theta(n)$ for $n' \equiv n$ and all

Figure A.1: The Trees Υ' and Υ'' for the Strategy Υ in Fig. 3.16

copies of n ; likewise $X'(n') \stackrel{\text{def}}{=} X(n)$. Then, Υ' has the above properties.

construction of Υ'' :

Using Υ' , we can eliminate all one-of nodes and projection nodes, to arrive at Υ'' : A subtree of Υ' rooted at a one-of node is replaced by the successor that corresponds to the alternative belonging to the context C .

After that alteration, any projection node n must be the predecessor of a leaf node $c \in C \cup D$. We then replace the subtree rooted at n by the constraint $\pi_X(c)$, where $X = X'(n) \cap \text{vars}(c)$.

Note that these alterations follow the defining properties Lem. 9, 2. and 3. We set $\Theta''(n) \stackrel{\text{def}}{=} \Theta'(n)$ and $X''(n) \stackrel{\text{def}}{=} X'(n)$, for all nodes n that have not been altered. Furthermore, we define $\Theta''(\lambda) \stackrel{\text{def}}{=} \lambda$ and $X''(\lambda) \stackrel{\text{def}}{=} \text{vars}(\lambda)$, for all leaves of Υ'' . This yields the tree Υ'' with the above properties.

$\rho(\Upsilon)$ is a cluster node:

Definition 18 implies that projection nodes have only one successor which is a constraint or a one-of node. Furthermore, any one-of node is succeeded by plain constraints. The condition $|\mathcal{F}| + |\mathcal{G}| \geq 2$, as given in Def. 18 and assumed for Γ , implies then that there must exist at least one cluster node in Υ . Consequently, the root of Υ must be a cluster node.

coincidence of root nodes:

Neither the construction of Υ' nor that of Υ'' touches cluster nodes. Therefore, the roots of all three graphs must coincide. Let us call that common root node ρ .

 $\Theta''(\rho)$ is an aggregation tree for $C \cup D \cup P$:

Let us show by induction that the set of aggregation trees

$$\{\Theta''(n) \mid n \in N\}$$

forms an aggregation forest for $C \cup D \cup P$, whenever the set of nodes N satisfies

$$\forall n_1, n_2 \in N \quad n_1 \neq n_2 \implies \Lambda(n_1) \cap \Lambda(n_2) = \emptyset, \quad \text{and} \\ \bigcup_{n \in N} \Lambda(n) = \Lambda(\rho),$$

Then, by choosing $N = \{\rho\}$, we conclude that $\Theta''(\rho)$ must be an aggregation tree for $C \cup D \cup P$.

We know that $C \cup D \cup P$ is a connected constraint problem, since C is assumed to be connected, and D and P mention a subset of all variables mentioned in C . Also, by construction, $\Lambda(\Upsilon'') = C \cup D \cup P$, and the set of all leaves form hence an aggregation forest for $C \cup D \cup P$, by Def. 10. Therefore, the above assertion holds for $N = \Lambda(\rho)$.

Suppose now, for a bottom-up induction on Υ'' , n is an arbitrary non-leaf node, and for its set of successors M , $M \subseteq N$ holds. Let us try to replace M by $\{n\}$ in N , producing $N' \stackrel{\text{def}}{=} (N \setminus M) \cup \{n\}$, and see whether the assertion still holds for the new set N' : Obviously, n must be a cluster node. Therefore by construction, $X''(n) = X(n)$, and from Def. 18, 5., we know that $X''(n)$ protects those variables that appear not exclusively in $\Lambda(n)$. Due to the connectedness of $\Lambda(\rho)$, the local strategy σ will always propose a pair of operands for the next aggregation, which will eliminate all unprotected variables. Therefore, $\Theta''(n)$ will be an aggregation tree built from the trees $\{\Theta''(m) \mid m \in M\}$, such that no variable is eliminated which appears in $\Lambda(\rho) \setminus \Lambda(n)$. This proves the assertion for the set N' , concluding the induction.

According to the above plan of the proof, this ends the argumentation. ■

A.11 Lemma 10**assertion (4.1)**

We assume that the left hand-side of the claimed implication is true. Let us start with the cases in which one of c', d' is a trivial constraint.

cases $c' \equiv \emptyset$ and $d' \equiv \square$: Here, $c' \rightarrow d'$ follows immediately from Def. 19.

case $c' \equiv \square$: Due to $c \sim c'$, we obtain that c is either \square itself or a non-trivial constraint that is non-restrictive. In the former case, $c \rightarrow d$ and Def. 19 yield $d \sim \square$. In order to see that this also holds in the latter case, regard condition 1. of Def. 19: According to Lem. 1, the set of assignments on the left-hand side contains all possible assignments to $X \cap Y$; and consequently also the set on the right. This proves

that $\pi_{X \cap Y}(d) \sim \square$. In addition to that, Def. 19, 2., says that any $y \in Y \setminus X$ must be a free variable of d which results in $d \sim \square$.

Now, with $d \sim d'$ we obtain $d' \sim \square$. Remembering that $c' \equiv \square$, Def. 19 is applicable, and we arrive at $c' \rightarrow d'$.

case $d' \equiv \emptyset$: This time, we get $d \sim \emptyset$. Thus, we either have $d \equiv \emptyset$ or d is non-trivial and has an empty set of assignments. As in the above argumentation, the former case provides us immediately with $c \sim \emptyset$. Let us try to verify that condition also for the latter case: By Def. 2, d has no free variables and thus $Y \setminus X = \emptyset$, by Def. 19, 2. So, all variables of d must also appear in c . But then, condition 1. of Def. 19 implies that $\pi_{X \cap Y}(c) \sim \emptyset$ and hence $c \sim \emptyset$.

Knowing that $c \sim \emptyset$ and $c \sim c'$, we obtain $c' \sim \emptyset$. This, together with $d' \equiv \emptyset$, yields the assertion $c' \rightarrow d'$, according to Def. 19.

In the remainder of this part of the proof, we shall assume that c' and d' are non-trivial constraints. Let us take a look at variables of d that are not free variables. I.e. be $x \in \text{vars}(d) \setminus \text{free}(d)$ arbitrary.

The second condition of Def. 19 implies $x \in \text{vars}(c)$. Furthermore, if $x \in \text{free}(c)$, then there were an assignment to x that could be extended to an assignment in c but not to one in d . Since that would contradict Def. 19, 1., we conclude that $x \notin \text{free}(c)$. Summarising, that argumentation shows the set inclusion $\text{vars}(d) \setminus \text{free}(d) \subseteq \text{vars}(c) \setminus \text{free}(c)$. So, together with (3.3) for $c \sim c'$ and $d \sim d'$, respectively, we obtain

$$\begin{aligned} Z &\stackrel{\text{def}}{=} \text{vars}(d') \setminus \text{free}(d') \\ &= \text{vars}(d) \setminus \text{free}(d) \\ &\subseteq \text{vars}(c) \setminus \text{free}(c) \\ &= \text{vars}(c') \setminus \text{free}(c'). \end{aligned}$$

Thereby, we have already shown condition 2. of Def. 19 for c' and d' , because, obviously, all variables of d' that do not appear in c' can only be free variables of d' .

In order to show also the first condition of Def. 19, regard the projections of c and d onto Z : By the above chain, $Z \subseteq \text{vars}(c) \cap \text{vars}(d)$. And so Def. 19, 1., interpreted for $c \rightarrow d$, tells us that any assignment in $\pi_Z(c)$ must also be an assignment of $\pi_Z(d)$. By (3.3), $\pi_Z(c') = \pi_Z(c)$ and $\pi_Z(d) = \pi_Z(d')$, which implies that any assignment in $\pi_Z(c')$ must belong to the assignments of $\pi_Z(d')$. Now, for establishing Def. 19, 1., we only need to explain why Z can be replaced by $\text{vars}(c') \cap \text{vars}(d')$.

By definition of Z , all variables of $(\text{vars}(c') \cap \text{vars}(d')) \setminus Z$ must be free variables of d' . And so, replacing Z by $\text{vars}(c') \cap \text{vars}(d')$ will clearly preserve the inclusion of assignments that we derived above.

So finally, also in the case of non-trivial c' and d' , we obtain $c' \rightarrow d'$.

assertion (4.2)

The idea now is to utilise (4.1), by replacing c and d by appropriate *lifts*: Let c', d' be constraints that arise from c and d by adding all variables in $\text{vars}(d) \setminus \text{vars}(c)$ and $\text{vars}(c) \setminus \text{vars}(d)$, respectively, as free variables. This produces *lifts*, i.e. constraints c' and d' with coinciding sets of variables and the properties $c \sim c'$ and $d \sim d'$.

Now, we apply (4.1) twice and obtain $c' \rightarrow d'$ and $d' \rightarrow c'$. Since c' and d' have the same variables, Def. 19, 1., provides us with the equality of their sets of assignments.

Hence, $c' \equiv d'$, and (3.3) yields $\pi_{\text{vars}(c') \setminus \text{free}(c')}(c') \equiv \pi_{\text{vars}(d') \setminus \text{free}(d')}(d')$. But, by construction of c' , the left-hand side of that equality coincides with $\pi_{\text{vars}(c) \setminus \text{free}(c)}(c)$. Likewise, the right term can be replaced by $\pi_{\text{vars}(d) \setminus \text{free}(d)}(d)$. The new equality proves, again according to (3.3), the assertion $c \sim d$.

assertion (4.3)

Lifting c, d and e , we obtain c', d' and e' with coinciding sets of variables and $c \sim c', d \sim d'$ and $e \sim e'$. Application of (4.1) produces $c' \rightarrow d'$ and $d' \rightarrow e'$. The set inclusions provided by Def. 19, 1., and the transitivity of set inclusion establish that first condition of Def. 19 also for the two constraints c' and e' . The second condition of Def. 19 need not be verified due to coinciding variable sets. Therefore, $c' \rightarrow e'$. Applying once more (4.1) yields $c \rightarrow e$.

assertion (4.4)

Let again c'_1 be a lift of c_1 such that the variables in $\text{vars}(c_2) \setminus \text{vars}(c_1)$ are free variables of c'_1 . Likewise be c'_2 a lift of c_2 such that c'_1 and c'_2 have coinciding variable sets.

Also, make this construction for d_1 and d_2 , producing d'_1 and d'_2 .

(4.1) provides us with $c'_1 \rightarrow c'_2$ and $d'_1 \rightarrow d'_2$. Due to coinciding variables and the first condition of Def. 19, this means that any assignment of c'_1 or d'_1 belongs also to c'_2 or d'_2 , respectively. But then, the definition of join, cf. Def. 4, implies that any assignment of $c'_1 \bowtie d'_1$ must belong to $c'_2 \bowtie d'_2$. Since both joins must still have the same variables, this can be rewritten as $c'_1 \bowtie d'_1 \rightarrow c'_2 \bowtie d'_2$.

The next step is to show that $c'_1 \bowtie d'_1 \sim c_1 \bowtie d_1$ and $c'_2 \bowtie d'_2 \sim c_2 \bowtie d_2$. Then (4.1) establishes the claimed implication of constraints. For that, let us consider some $x \in \text{free}(c'_1)$:

case $x \notin \text{vars}(d'_1)$: Here, x must also be a free variable of $c'_1 \bowtie d'_1$, see Def. 2 and Lem. 1. Moreover, with the abbreviation $Z \stackrel{\text{def}}{=} \text{vars}(c'_1) \setminus \{x\}$, $\pi_{(\text{vars}(c') \cup \text{vars}(d')) \setminus \{x\}}(c'_1 \bowtie d'_1) \equiv \pi_Z(c'_1) \bowtie d'_1$.

case $x \in \text{vars}(d'_1)$: This time, we can omit x in c'_1 to obtain $c'_1 \bowtie d'_1 \equiv \pi_Z(c'_1) \bowtie d'_1$, where Z is the same set as in the previous case.

In the first case, $c'_1 \bowtie d'_1 \sim \pi_Z(c'_1) \bowtie d'_1$ follows once more from (3.3). In the second case, this condition also clearly holds.

The entire argument shows that we can eliminate, from c'_1 any free variable, without losing equivalence. Repetitive application proves that we can hence replace c'_1 even by c_1 , i.e. $c'_1 \bowtie d'_1 \sim c_1 \bowtie d'_1$. Of course, by a symmetric argument, also d'_1 can be replaced by d_1 which finally results in $c'_1 \bowtie d'_1 \sim c_1 \bowtie d_1$.

By symmetry, also $c'_2 \bowtie d'_2 \sim c_2 \bowtie d_2$ must hold. As already mentioned above, one application of (4.1) establishes now the claim $c_1 \bowtie d_1 \rightarrow c_2 \bowtie d_2$.

assertion (4.5)

Let us directly verify Def. 19 for the two projections. The first condition follows from the set inclusion given by $c \rightarrow d$: We just need to replace the sets of assignments, i.e., the α 's and β 's, by the sets of restrictions of α 's and β 's to the appropriate smaller set of variables. Note that this process preserves the given set inclusion.

For the second part consider

$$\begin{aligned}
 (X \cap \text{vars}(d)) \setminus (X \cap \text{vars}(c)) &= X \cap (\text{vars}(d) \setminus \text{vars}(c)) \\
 &\subseteq X \cap \text{free}(d) \\
 &\subseteq \text{free}(\pi_{X \cap \text{vars}(d)}(d)),
 \end{aligned}$$

where the first set inclusion follows from Def. 19, 2., instantiated for $c \rightarrow d$. The second inclusion is also easily verified from Def. 2 and Lem. 1.

This shows the last assertion and finishes the proof of Lem. 10. ■

A.12 Invariants (4.6) and (4.7)

Let us assume that the used implementation of `isEmpty` always yields the correct answer. The proof concerns the pseudo-code in Fig. 4.7 and is according to the following plan:

1. (4.7) is valid for the implementation at `LeafNode`.
2. If (4.6) is valid for a call in `AggNode`'s implementation, then we may assume, by induction, that (4.7) is valid for that call.
3. (4.6) is valid for line (1).
4. We can conclude that (4.7) is also valid for line(1).

Now we show 1.-3.:

implementation at class `LeafNode`

We show (4.7): Since Def. 15 includes the leaf relation into $\widehat{\text{con}}_{\uparrow}(\lambda)$, we know that $\widehat{\text{con}}_{\uparrow}(\lambda) \cap \Lambda(\lambda) = \{\lambda\}$. And this is exactly what the method returns.

implementation at class `AggNode`

Let r, r_1 and r_2 be the forward relations associated with the given aggregation node and its two successors. c denote the relation represented by the method's argument. By induction, we may assume that (4.6) holds, i.e. $\bowtie \widehat{\text{con}}_{\downarrow}(r) = c$.

subcase: line (2) approached:

The precondition ensures that $r_1 \bowtie c \sim \emptyset$. Then $\bowtie \widehat{\text{con}}_{\downarrow}(r) = c$ turns this into $\bowtie (\widehat{\text{con}}_{\downarrow}(r) \cup \{r_1\}) \sim \emptyset$, and consequently Def. 15 implies $\widehat{\text{con}}_{\downarrow}(r) = \widehat{\text{con}}_{\downarrow}(r_1)$. Thus $\bowtie \widehat{\text{con}}_{\downarrow}(r_1) = c$ and (4.6) holds for the recursive call in line (2).

By induction, we may assume that line (2) returns, according to (4.7), the set $\widehat{\text{con}}_{\uparrow}(r_1) \cap \Lambda(r_1)$. In order to serve the above induction, we need to show that this equals $\widehat{\text{con}}_{\uparrow}(r) \cap \Lambda(r)$.

There was no recursive call for r_2 , and so $\widehat{\text{con}}_{\downarrow}(r_2) = \widehat{\text{con}}_{\uparrow}(r_2) = \star$. Definition 15 gives thus $\widehat{\text{con}}_{\uparrow}(r) = \widehat{\text{con}}_{\uparrow}(r_1)$. Furthermore

$$\begin{aligned}
 \emptyset &= \widehat{\text{con}}_{\downarrow}(r) \cap \Lambda(r_2), && \text{because of Lem. 7, 3(b),} \\
 &= \widehat{\text{con}}_{\downarrow}(r_1) \cap \Lambda(r_2) \\
 &= \widehat{\text{con}}_{\uparrow}(r_1) \cap \Lambda(r_2), && \text{by Lem. 7, 3(a),} \\
 &= \widehat{\text{con}}_{\uparrow}(r) \cap \Lambda(r_2).
 \end{aligned}$$

Putting the pieces together, we obtain $\widehat{con}_\uparrow(r_1) \cap \Lambda(r_1) = \widehat{con}_\uparrow(r) \cap \Lambda(r_1)$ and $\widehat{con}_\uparrow(r) \cap \Lambda(r_2) = \emptyset$; e.g. $\widehat{con}_\uparrow(r_1) \cap \Lambda(r_1) = \widehat{con}_\uparrow(r) \cap \Lambda(r)$.

subcase: line(3) approached:

This is completely symmetric to the previous subcase.

subcase: processing lines (4) and (5):

Note that none of the preconditions of line (2) and line (3) evaluated to **true**. Thus Def. 15 yields $\widehat{con}_\downarrow(r_1) = \widehat{con}_\downarrow(r) \cup \{r_2\}$ and $\widehat{con}_\downarrow(r_2) = \widehat{con}_\uparrow(r_1) \setminus \{r_2\}$. Moreover, $\widehat{con}_\uparrow(r_1)$ does indeed contain r_2 , i.e. $\widehat{con}_\uparrow(r_1) = \widehat{con}_\downarrow(r_2) \cup \{r_2\}$. But $r \notin \widehat{con}_\downarrow(r)$ because of Lem. 7, part 3(b); and so $r \notin \widehat{con}_\downarrow(r) \cup \{r_2\} = \widehat{con}_\downarrow(r_1)$.

In order to verify (4.6) for the recursive call in line (4), we have to show $\bowtie \widehat{con}_\downarrow(r_1) = r_2 \bowtie c$. Taking into account that $\bowtie \widehat{con}_\downarrow(r) = c$, we immediately arrive at that assertion. In the following, we can assume (4.7) for line (4), by induction, i.e. $S1 = \widehat{con}_\uparrow(r_1) \cap \Lambda(r_1)$.

Next, let us verify (4.6) for the recursive call in line (5). First, note that $\widehat{con}_\uparrow(r_1) \cap V(\Delta(r_2)) = (\widehat{con}_\downarrow(r_2) \cap V(\Delta(r_2))) \cup \{r_2\} = \{r_2\}$ because of Lem. 7, 3(b). Secondly,

$$\begin{aligned} \bowtie(\widehat{con}_\uparrow(r_1) \setminus V(\Delta(r))) &= \bowtie(\widehat{con}_\downarrow(r_1) \setminus V(\Delta(r))), \quad \text{by Lem. 7, 3(a)} \\ &= \bowtie(\widehat{con}_\downarrow(r) \setminus V(\Delta(r))) \\ &= \bowtie \widehat{con}_\downarrow(r), \quad \text{by Lem. 7, 3(b)} \\ &= c. \end{aligned}$$

Therefore,

$$\begin{aligned} \bowtie \widehat{con}_\uparrow(r_1) &= \bowtie(\widehat{con}_\uparrow(r_1) \setminus V(\Delta(r))) \bowtie \\ &\quad \bowtie(\widehat{con}_\uparrow(r_1) \cap \{r\}) \bowtie \\ &\quad \bowtie(\widehat{con}_\uparrow(r_1) \cap V(\Delta(r_1))) \bowtie \\ &\quad \bowtie(\widehat{con}_\uparrow(r_1) \cap V(\Delta(r_2))) \\ &= c \bowtie \square \bowtie \bowtie(\widehat{con}_\uparrow(r_1) \cap V(\Delta(r_1))) \bowtie r_2. \end{aligned}$$

Consequently,

$$\begin{aligned} \bowtie \widehat{con}_\downarrow(r_2) &= \bowtie(\widehat{con}_\uparrow(r_1) \setminus \{r_2\}) \\ &= c \bowtie \bowtie(\widehat{con}_\uparrow(r_1) \cap V(\Delta(r_1))) \\ &= c \bowtie \bowtie(\widehat{con}_\uparrow(r_1) \cap \Lambda(r_1)), \quad \text{by Lem. 7, 3(c),} \\ &= c \bowtie \bowtie S1, \end{aligned}$$

which proves (4.6) for the recursive call in line (5).

Again, by induction, we know that then (4.7) will hold for the invocation in line (5).

This means that $S2 = \widehat{con}_\uparrow(r_2) \cap \Lambda(r_2)$. Moreover, we have

$$\begin{aligned} \widehat{con}_\uparrow(r_2) \setminus V(\Delta(r_2)) &= \widehat{con}_\downarrow(r_2) \setminus V(\Delta(r_2)), \quad \text{by Lem. 7, 3(a),} \\ &= \widehat{con}_\uparrow(r_1) \setminus V(\Delta(r_2)), \quad \text{since } \widehat{con}_\downarrow(r_2) = \widehat{con}_\uparrow(r_1) \setminus \{r_2\}. \end{aligned}$$

In particular, this implies that the portions of $\widehat{con}_\uparrow(r_2)$ and $\widehat{con}_\uparrow(r_1)$ in $\Lambda(r_1)$ must agree, i.e. $\widehat{con}_\uparrow(r_2) \cap \Lambda(r_1) = \widehat{con}_\uparrow(r_1) \cap \Lambda(r_1)$.

The last set coincides with $S1$ and summarising we obtain

$$\begin{aligned}
 \widehat{con}_\uparrow(r_2) \cap \Lambda(r_1) &= S1 && \text{and} \\
 \widehat{con}_\uparrow(r_2) \cap \Lambda(r_2) &= S2 \\
 \Rightarrow \widehat{con}_\uparrow(r_2) \cap \Lambda(r) &= S1 \cup S2 \\
 \Rightarrow \widehat{con}_\uparrow(r) \cap \Lambda(r) &= S1 \cup S2, \text{ by Def. 15.}
 \end{aligned}$$

This proves the second invariant (4.7) in the last subcase.

(4.6) for line (1)

Taking a look at Def. 15, we see that the root node's associated set $\widehat{con}_\downarrow(.)$ is empty. Thus the join in (4.6) is defined to be \square . And indeed, the argument of the call in line (1) is a representation of that trivial constraint.

The remainder of the proof has to deal only with the two lower implementations of `getOneConflict(c)`. It is easy to verify that a call of one of those will take place if and only if both corresponding sets $\widehat{con}_\downarrow(.)$ and $\widehat{con}_\uparrow(.)$ are not the special symbol \star ; see Def. 15.

The above steps prove the validity of both invariants for any invocation of the method `getOneConflict(c)`. ■

A.13 Lemma 11

The proof will assume that the static creation methods implemented at **Factory** satisfy their specification. That means that they return correct representations and the sole instance of **Empty** only when the given relation is indeed unsatisfiable. With this assumption, we may forget about representations of relations, but rather talk about the represented relations themselves. See also Fig. 5.5 to follow the details of the proof.

Let r denote the relation for which `eliminateBasic` is invoked with the set X of variables to be eliminated. r is assumed to be in PSF for X . Set $Y \stackrel{\text{def}}{=} \text{vars}(r) \setminus X$.

implementation at **Empty** and **Full**:

If the representation of r is the sole instance of **Empty** or **Full**, then r has been found unsatisfiable or nonrestrictive, respectively. In the first case, (3.7) ensures that $\pi_Y(r) \sim \emptyset$.

In the second case, $r \sim \square$ must hold, and all variables of r are hence free variables. But then (3.3) implies $\pi_Y(r) \sim \square$. So, in both cases, the representation of r is also an appropriate representation of $\pi_Y(r)$, and line (1) of Fig. 5.6 yields the correct result.

implementation at **Or**:

Here, r has the structure $\bigvee_{i \in I} d_i$ with some finite index set I and the disjuncts d_i being conjunctions of arithmetic atomic constraints. Let us assume that the implementation at **AbstractAnd** always return the result as stated in Lem. 11. (The proof follows below.) Then the embedded call of `eliminateBasic` for each d_i returns a representation of $\pi_{Y \cap \text{vars}(d_i)}(d_i)$.

Obviously, the given pseudo-code produces the correct result if and only if

$$\pi_Y \left(\bigvee_{i \in I} d_i \right) \equiv \bigvee_{i \in I} \pi_{Y \cap \text{vars}(d_i)}(d_i).$$

But this is easily verified using the definitions of *project* and *or*; see Defs. 5 and 20: We shall first check the equality of the variable sets on both sides of the above claim: By definition, Y is a subset of $\text{vars}(r)$. So, the left-hand side relation is well-defined and has the variables Y . The right-hand side is clearly well-defined. Moreover, for any $y \in Y$, there must exist some d_i in which it appears. And so the set of variables of the right-hand side constraint is also Y .

In order to show that the sets of assignments on both sides coincide, we are going to show both set inclusions, \subseteq and \supseteq .

\subseteq :

Be α an assignment (to Y) of the left-hand side. Then α can be extended to some assignment α' of $\bigvee_{i \in I} d_i$. By Def. 20, there must exist some $k \in I$ such that $\alpha'|_{\text{vars}(d_k)}$ is an assignment of d_k . Writing $Y_k \stackrel{\text{def}}{=} Y \cap \text{vars}(d_k)$, this implies that $\alpha'|_{Y_k}$ belongs to $\pi_{Y_k}(d_k)$. But $\alpha \equiv \alpha'|_Y$ and consequently α and α' agree on Y_k . Therefore, $\alpha|_{Y_k}$ is an assignment of $\pi_{Y_k}(d_k)$. Applying once more Def. 20 shows that α must thus belong to the right-hand side set of assignments.

\supseteq :

Going the other, somewhat more difficult direction, we take an arbitrary assignment α of the right-hand side. We know already that α is then defined on Y . By Def. 20, there must exist some $k \in I$ such that $\alpha|_{Y \cap \text{vars}(d_k)}$ belongs to $\pi_{Y_k}(d_k)$, where Y_k is defined as in the above proof of the other set inclusion.

By the definition of *project*, we can extend $\alpha|_{Y_k}$ to some α' that assigns a value to each variable in $\text{vars}(d_k)$ and that belongs to d_k . Extension means that $\alpha'|_{Y_k} \equiv \alpha|_{Y_k}$. Now define a further assignment α'' on Z , where

$$\begin{aligned} Z &\stackrel{\text{def}}{=} \bigcup_{i \in I} \text{vars}(d_i), \\ \alpha''|_Y &\stackrel{\text{def}}{=} \alpha, \\ \alpha''|_{(Z \setminus Y) \cap \text{vars}(d_k)} &\stackrel{\text{def}}{=} \alpha'|_{(Z \setminus Y) \cap \text{vars}(d_k)}, \quad \text{and} \\ \alpha''|_{(Z \setminus Y) \setminus \text{vars}(d_k)} &\text{arbitrary.} \end{aligned}$$

Obviously, $\alpha''|_{Y_k} \equiv \alpha|_{Y_k} \equiv \alpha'|_{Y_k}$, since $Y_k \subseteq Y$. Thus α'' and α' agree on the whole of $\text{vars}(d_k) = Y_k \cup ((Z \setminus Y) \cap \text{vars}(d_k))$, see above definition of α'' . But this means that α'' is an extension of α' , and - by Def. 20 - that α'' is an assignment of $\bigvee_{i \in I} d_i$. Note also that $\alpha''|_Y \equiv \alpha$, and so α must be an assignment of the left-hand side constraint.

implementation of AbstractAnd:

Let us write $r = \bowtie_{i \in I} a_i$ with arithmetic atomic constraints a_i and I again a finite index set as above. The set A in Fig. 5.6 equals hence $\{a_i \mid i \in I\}$.

First note that all lines of code before line (8) add only implied constraints to A . So $\bowtie A$ is not at all altered. The result of line (8), in turn, is - due to the removal of several members of A - an overestimation of that combined join. We conclude that

the bottom implementation of `eliminateBasic` returns always an overestimation of r .

Furthermore, the resulting set of atoms in line (8) does not mention any of X any longer. (See the pseudo-code, and remember that r is in PSF for X .)

Summarising, we obtain, for the returned relation s , that $r \rightarrow s$ and $\text{vars}(s) \subseteq \text{vars}(r) \setminus X = Y$. The application of (4.5) yields

$$\begin{aligned} \pi_{Y \cap \text{vars}(r)}(r) &\rightarrow \pi_{Y \cap \text{vars}(s)}(s), \quad \text{that is,} \\ \pi_Y(r) &\rightarrow s. \end{aligned}$$

According to (4.2), the proof can be completed by showing the converse implication. For technical reasons, we need to consider a lift of s , that is, a constraint s' with $s' \sim s$ and $\text{vars}(s') = Y$ in which the additional variables $Y \setminus \text{vars}(s)$ (if any) are free variables.

Let us now pick an arbitrary assignment α of s' . Then, α assigns a value to each element of Y . In what follows, we try to extend α to some assignment of r . In order to accomplish that, we have to find an extension so that, in addition to s' , also each atomic constraint in A_X is satisfied.

To this end, take $x \in X$, and write $A_x \subseteq A_X$ for the set of constraints that mention x as basic variable. Let us extend α to α_x defined on $Y \cup \{x\}$, such that α_x is also an assignment for each $a \in A_x$.

There are two cases: A_x is a set of inequalities (the test in line (3), Fig. 5.6, yielded `true`), or consists of exactly one equation. In both cases, any atom in A_x has the form $x \diamond t$, where $x \notin \text{vars}(t)$ and $\diamond \in \{=, \leq, <, \geq, >\}$.

In the case of one equation, the constraint $x = t$ determines the value for x , since $\text{vars}(t) \subseteq Y$ and α assigns values to all elements of Y . So here, the extension of α to α_x is obvious.

Similarly, in the case of inequalities, we consider the sets of terms as defined after line (3) of Fig. 5.6: Any of these terms can be evaluated to a real number, using the assignment α . Note that no evaluation will fail since s' takes care of all implicit constraints of A , as inserted in line (2). Let us denote the sets of evaluated terms by

$$T_\diamond(\alpha) \stackrel{\text{def}}{=} \{t(\alpha) \mid t \in T_\diamond\}, \quad \text{where } \diamond \in \{\leq, <, \geq, >\}.$$

Note that, by the construction of s , we are provided with the two conditions

$$\begin{aligned} \forall (v_{\text{lower}}, v_{\text{upper}}) \in T_{\leq}(\alpha) \times T_{\geq}(\alpha) \quad &v_{\text{lower}} \leq v_{\text{upper}}, \\ \forall (v_{\text{lower}}, v_{\text{upper}}) \in ((T_{\leq}(\alpha) \cup T_{<}(\alpha)) \times (T_{\geq}(\alpha) \cup T_{>}(\alpha))) \setminus (T_{\leq}(\alpha) \times T_{\geq}(\alpha)) & \\ &v_{\text{lower}} < v_{\text{upper}}. \end{aligned}$$

This gives us a clue as to how to extend α to α_x : We define $\alpha_x(x)$ to be an arbitrary value with the properties

$$\begin{aligned} m < \alpha_x(x) < M, \quad &\text{if } m < M \text{ and,} \\ \alpha_x(x) &\stackrel{\text{def}}{=} m, \quad \text{if } m = M, \text{ where} \\ m &= \max(T_{\leq}(\alpha) \cup T_{<}(\alpha)), \quad \text{and} \\ M &= \min(T_{\geq}(\alpha) \cup T_{>}(\alpha)). \end{aligned}$$

Regarding the above two conditions, it is straightforward to verify $m \leq M$. Moreover, α_x is then an assignment of each atom in A_x .

Repeated variable-wise extension of α in the elaborated way will finally produce an assignment of r . This construction establishes the implication $s' \rightarrow \pi_Y(r)$. Also, $s \sim s'$, and so (4.1) gives us the desired converse implication $s \rightarrow \pi_Y(r)$.

This finishes the proof of the lemma ■

A.14 Lemma 12

Pick any $i \in \{1, 2, \dots, n\}$ and fix the variables $x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n$. Let us consider the function

$$\begin{aligned} t_i : \text{dom}(x_i) &\longrightarrow \mathbb{R} \\ x_i &\longmapsto t(x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n). \end{aligned}$$

We will show that t_i is monotonous on $\text{dom}(x_i)$.¹ For that, we shall examine the derivative dt_i/dx_i . We may assume that t is well-defined on the whole of K , and thus - as a rational function - arbitrarily often differentiable. We claim that dt_i/dx_i never changes sign on $\text{dom}(x_i)$, which would clearly suffice to establish the stated (non-strict) monotonicity. Regard the differentiation rules

$$\begin{aligned} \frac{d}{dx_i} u \pm v(x_i) &= \pm \frac{d}{dx_i} v(x_i), & \frac{d}{dx_i} u \cdot v(x_i) &= u \cdot \frac{d}{dx_i} v(x_i), \\ \frac{d}{dx_i} \frac{u}{v(x_i)} &= -\frac{u}{v(x_i)^2} \cdot \frac{d}{dx_i} v(x_i), & \frac{d}{dx_i} x_i &= 1, \end{aligned}$$

where u shall not mention x_i .

Remember that t mentions x_i exactly once, and so computing dt_i/dx_i according to the above rules will yield a product with the factors $\pm 1, u, -u/v(x_i)^2$ and finally one 1. Consequently, dt_i/dx_i never changes sign on $\text{dom}(x_i)$ if and only if neither u nor $u/v(x_i)^2$ change sign. But the former term does not mention x_i , and the second always has a positive divisor.

The above argument shows that t is monotonous in each variable and must hence take each of its extremal values in some corner of the cuboid K .

For the lemma's final conclusion, note that the naive evaluation of t using interval algebra, basically evaluates t for every element of K *in parallel*. Thus, with V denoting the result of naive interval algebra, m the minimum and M the maximum of t on K , we obtain $V \supseteq [m, M]$.

For the reverse set inclusion regard the well-known calculation rules for $+$, $-$, \cdot and \div on intervals: Those just combine the boundary points of the two argument intervals appropriately. Therefore, V will also be an interval, the boundary points of which are evaluations of t in some corners of K . But this implies that $\min(V)$ and $\max(V)$ lie in $[m, M]$, resulting in $V \subseteq [m, M]$.

This shows that the result of naive interval algebra coincides with $[m, M]$, and we are done. ■

¹Note however that t_i need not be *strictly* increasing or decreasing, as e.g. $t_i(x_i) = 0 \cdot x_i$.

Appendix B

XML String Representations

B.1 The 1-Bit Full Adder

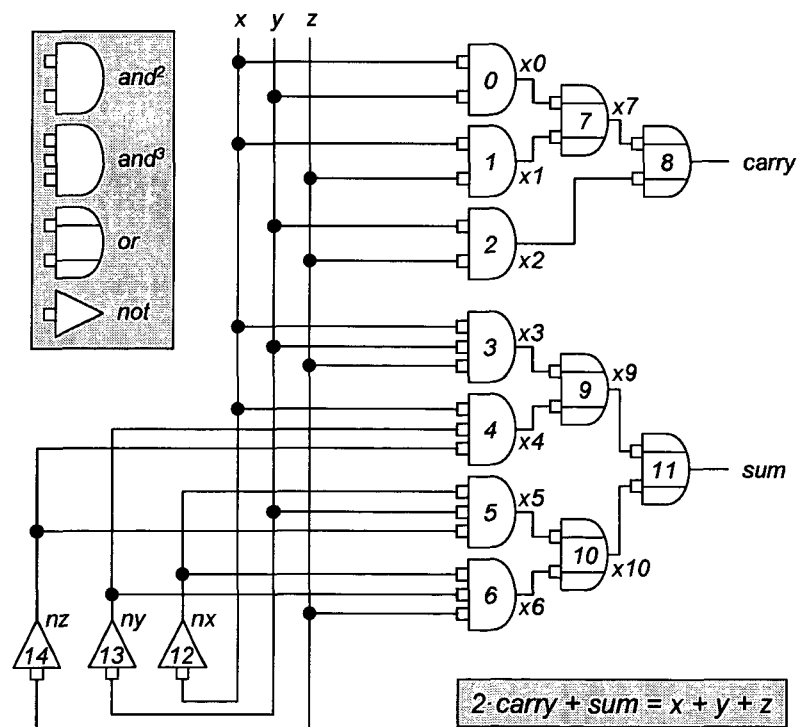


Figure B.1: The 1-Bit Full Adder

Figure B.1 depicts a 1-bit full adder. The three boolean entries x , y , and z are added to yield a *sum* and the overflow *carry*.

In the following, an XML sheet is listed that models all gate constraints and a few additional relations for setting up some contexts of interest.

```

<?xml version='1.0' encoding=UTF-8'?>
<!DOCTYPE objects SYSTEM 'relations_v0.10.dtd'>
<!-- based on the relations DTD version 0.10 -->
<!-- a 1-bit full adder: 2carry + sum = x + y + z. -->
<objects>
  <!-- a special variable domain: the two Boolean values -->
  <defSet n='Boolean'> <symbol v='F T'> </defSet>

  <!-- all variables of the full adder; all are Boolean -->
  <defVar n='x y z sum carry nx ny nz
           x0 x1 x2 x3 x4 x5 x6 x7 x9 x10'>
    <setRef n='Boolean'>
  </defVar>

  <!-- relation template for the NOT gate -->
  <defRel n='NOT' v='in out'>
    <or>
      <and>
        <eq> <var n='in'> <symbol v='F'> </eq>
        <eq> <var n='out'> <symbol v='T'> </eq>
      </and>
      <and>
        <eq> <var n='in'> <symbol v='T'> </eq>
        <eq> <var n='out'> <symbol v='F'> </eq>
      </and>
    </or>
  </defRel>

  <!-- relation template for the OR gate -->
  <defRel n='OR' v='in1 in2 out'>
    <or>
      <and>
        <eq> <var n='in1'> <symbol v='F'> </eq>
        <eq> <var n='in2'> <symbol v='F'> </eq>
        <eq> <var n='out'> <symbol v='F'> </eq>
      </and>
      <and>
        <eq> <var n='in1'> <symbol v='F'> </eq>
        <eq> <var n='in2'> <symbol v='T'> </eq>
        <eq> <var n='out'> <symbol v='T'> </eq>
      </and>
      <and>
        <eq> <var n='in1'> <symbol v='T'> </eq>
        <eq> <var n='in2'> <symbol v='F'> </eq>
        <eq> <var n='out'> <symbol v='T'> </eq>
      </and>
      <and>
        <eq> <var n='in1'> <symbol v='T'> </eq>
        <eq> <var n='in2'> <symbol v='T'> </eq>
        <eq> <var n='out'> <symbol v='T'> </eq>
      </and>
    </or>
  </defRel>

```



```

        <eq> <var n='in1' /> <symbol v='T' /> </eq>
        <eq> <var n='in2' /> <symbol v='T' /> </eq>
        <eq> <var n='out' /> <symbol v='T' /> </eq>
    </and>
</or>
</defRel>

<!-- relation template for the binary AND gate -->
<defRel n='AND2' v='in1 in2 out'>
    <or>
        <and>
            <eq> <var n='in1' /> <symbol v='F' /> </eq>
            <eq> <var n='in2' /> <symbol v='F' /> </eq>
            <eq> <var n='out' /> <symbol v='F' /> </eq>
        </and>
        <and>
            <eq> <var n='in1' /> <symbol v='F' /> </eq>
            <eq> <var n='in2' /> <symbol v='T' /> </eq>
            <eq> <var n='out' /> <symbol v='F' /> </eq>
        </and>
        <and>
            <eq> <var n='in1' /> <symbol v='T' /> </eq>
            <eq> <var n='in2' /> <symbol v='F' /> </eq>
            <eq> <var n='out' /> <symbol v='F' /> </eq>
        </and>
        <and>
            <eq> <var n='in1' /> <symbol v='T' /> </eq>
            <eq> <var n='in2' /> <symbol v='T' /> </eq>
            <eq> <var n='out' /> <symbol v='T' /> </eq>
        </and>
    </or>
</defRel>

<!-- relation template for the 3-ary AND gate -->
<defRel n='AND3' v='in1 in2 in3 out'>
    <or>
        <and>
            <eq> <var n='in1' /> <symbol v='F' /> </eq>
            <eq> <var n='in2' /> <symbol v='F' /> </eq>
            <eq> <var n='in3' /> <symbol v='F' /> </eq>
            <eq> <var n='out' /> <symbol v='F' /> </eq>
        </and>
        <and>
            <eq> <var n='in1' /> <symbol v='F' /> </eq>
            <eq> <var n='in2' /> <symbol v='F' /> </eq>
            <eq> <var n='in3' /> <symbol v='T' /> </eq>
            <eq> <var n='out' /> <symbol v='F' /> </eq>
        </and>
    </or>
</defRel>

```

```

</and>
<and>
  <eq> <var n='in1' /> <symbol v='F' /> </eq>
  <eq> <var n='in2' /> <symbol v='T' /> </eq>
  <eq> <var n='in3' /> <symbol v='F' /> </eq>
  <eq> <var n='out' /> <symbol v='F' /> </eq>
</and>
<and>
  <eq> <var n='in1' /> <symbol v='F' /> </eq>
  <eq> <var n='in2' /> <symbol v='T' /> </eq>
  <eq> <var n='in3' /> <symbol v='T' /> </eq>
  <eq> <var n='out' /> <symbol v='F' /> </eq>
</and>
<and>
  <eq> <var n='in1' /> <symbol v='T' /> </eq>
  <eq> <var n='in2' /> <symbol v='F' /> </eq>
  <eq> <var n='in3' /> <symbol v='F' /> </eq>
  <eq> <var n='out' /> <symbol v='F' /> </eq>
</and>
<and>
  <eq> <var n='in1' /> <symbol v='T' /> </eq>
  <eq> <var n='in2' /> <symbol v='F' /> </eq>
  <eq> <var n='in3' /> <symbol v='T' /> </eq>
  <eq> <var n='out' /> <symbol v='F' /> </eq>
</and>
<and>
  <eq> <var n='in1' /> <symbol v='T' /> </eq>
  <eq> <var n='in2' /> <symbol v='T' /> </eq>
  <eq> <var n='in3' /> <symbol v='F' /> </eq>
  <eq> <var n='out' /> <symbol v='F' /> </eq>
</and>
<and>
  <eq> <var n='in1' /> <symbol v='T' /> </eq>
  <eq> <var n='in2' /> <symbol v='T' /> </eq>
  <eq> <var n='in3' /> <symbol v='T' /> </eq>
  <eq> <var n='out' /> <symbol v='T' /> </eq>
</and>
</or>
</defRel>

<!-- here come all gates implementing the full adder -->
<!-- (we just instantiate the above (named) templates) -->
<rel n='r0' r='AND2' v='x y x0' />
<rel n='r1' r='AND2' v='x z x1' />
<rel n='r2' r='AND2' v='y z x2' />
<rel n='r3' r='AND3' v='x y z x3' />

```

```

<rel n='r4' r='AND3' v='x ny nz x4' />
<rel n='r5' r='AND3' v='nx y nz x5' />
<rel n='r6' r='AND3' v='nx ny z x6' />
<rel n='r7' r='OR' v='x0 x1 x7' />
<rel n='r8' r='OR' v='x2 x7 carry' />
<rel n='r9' r='OR' v='x3 x4 x9' />
<rel n='r10' r='OR' v='x5 x6 x10' />
<rel n='r11' r='OR' v='x9 x10 sum' />
<rel n='r12' r='NOT' v='x nx' />
<rel n='r13' r='NOT' v='y ny' />
<rel n='r14' r='NOT' v='z nz' />

<!-- some variable assignments for testing the full adder -->
<eq n='x.T'> <var n='x' /> <symbol v='T' /> </eq>
<eq n='y.T'> <var n='y' /> <symbol v='T' /> </eq>
<eq n='z.T'> <var n='z' /> <symbol v='T' /> </eq>
<eq n='sum.T'> <var n='sum' /> <symbol v='T' /> </eq>
<eq n='x.F'> <var n='x' /> <symbol v='F' /> </eq>
<eq n='y.F'> <var n='y' /> <symbol v='F' /> </eq>
<eq n='z.F'> <var n='z' /> <symbol v='F' /> </eq>
<eq n='sum.F'> <var n='sum' /> <symbol v='F' /> </eq>

</objects>

```

After loading the above XML file, the prototype will have, in its pool of known relations, the 15 gate relations `r0 - r14` and 8 additional variable assignments. In order to analyse a certain situation, the user must activate the former 15 constraints and some of the latter 8, so that the context of interest is modelled.

B.2 Context Space for the 1-Bit Full Adder

Our implementation is able to sequentially analyse all 16 contexts of the above 1-bit full adder. Those 2^4 contexts arise from four `OneOfs` with two alternatives each. Those need to be defined (and named) before the context space itself is declared. The `OneOfs` and the corresponding context space can be imported from the XML file given below.

```

<?xml version='1.0' encoding=UTF-8'?>
<!DOCTYPE objects SYSTEM 'relations_v0.10.dtd'>
<!-- This is an XML file containing a context space -->
<!-- definition for the 1-bit full adder. -->

<objects>

  <!-- defining the four oneOfs -->
  <oneOf n='x' rel='x.T x.F' />
  <oneOf n='y' rel='y.T y.F' />
  <oneOf n='z' rel='z.T z.F' />
  <oneOf n='sum' rel='sum.T sum.F' />

```

```

<!-- the context space with 16 contexts -->
<contextSpace>
  <oneOf n='x' />
  <oneOf n='y' />
  <oneOf n='z' />
  <oneOf n='sum' />
  <rel n='r0' />
  <rel n='r1' />
  <rel n='r2' />
  <rel n='r3' />
  <rel n='r4' />
  <rel n='r5' />
  <rel n='r6' />
  <rel n='r7' />
  <rel n='r8' />
  <rel n='r9' />
  <rel n='r10' />
  <rel n='r11' />
  <rel n='r12' />
  <rel n='r13' />
  <rel n='r14' />
</contextSpace>
</objects>

```

B.3 XML Strings for Modelling Electric Components

The previous sections deal with finite domain variables and constraints. Now we shall give an idea of how the XML representations of the problems R_k and D_k of Fig. 2.7 look like; see also Fig. 2.8.

To this end, we just list the relation template declarations. With those, both problems can easily be stated in XML, simply by making appropriate instantiations.

Note that, as mentioned in Subsect. 2.1.2, currents always point inwards with respect to the modelled component. Besides the simple model of an Ohmic resistor as utilised in Fig. 2.8, we also give a model with an *ok* and a *broken* mode. For later reference in App. B.5, models for a two-mode bulb and a switch shall be provided, too.

```

<!-- relation template for an electric wire -->
<!-- i...(directed) current variable -->
<!-- v...(undirected) voltage variable -->
<defRel n='Wire' v='i1 i2 v1 v2'>
  <and>
    <eq>
      <plus> <var n='i1' /> <var n='i2' /> </plus>
      <float v='0.0' />
    </eq>
  </and>
</defRel>

```

```

    </eq>
    <eq> <var n='v1'/> <var n='v2'/> </eq>
  </and>
</defRel>

<!-- relation template for a Kirchhoff node -->
<defRel n='KirchhoffNode' v='i1 i2 i3 v1 v2 v3'>
  <and>
    <eq>
      <plus>
        <plus> <var n='i1'/> <var n='i2'/> </plus>
        <var n='i3'/>
      </plus>
      <float v='0.0'/>
    </eq>
    <eq> <var n='v1'/> <var n='v2'/> </eq>
    <eq> <var n='v1'/> <var n='v3'/> </eq>
  </and>
</defRel>

<!-- relation template for a simple Ohmic resistor -->
<defRel n='SimpleOhmicResistor' v='i1 i2 v1 v2 r'>
  <and>
    <eq>
      <plus> <var n='i1'/> <var n='i2'/> </plus>
      <float v='0.0'/>
    </eq>
    <eq>
      <minus> <var n='v1'/> <var n='v2'/> </minus>
      <mult> <var n='i1'/> <var n='r'/> </mult>
    </eq>
  </and>
</defRel>

<!-- an additional finite domain variable for the -->
<!-- working modes of an ideal diode -->
<defSet n='workingMode'> <symbol v='through blocking'/> </defSet>

<!-- relation template for an ideal diode -->
<defRel n='IdealDiode' v='i1 i2 v1 v2 mode'>
  <or>
    <and>
      <eq> <var n='mode'/> <symbol v='through'/> </eq>
      <eq>
        <plus> <var n='i1'/> <var n='i2'/> </plus>
        <float v='0.0'/>
      </eq>
      <geq> <var n='i1'/> <float v='0.0'/> </geq>
    </and>
  </or>

```

```

    <eq> <var n='v1'/'> <var n='v2'/'> </eq>
  </and>
  <and>
    <eq> <var n='mode'/'> <symbol v='blocking'/'> </eq>
    <eq> <var n='i1'/'> <float v='0.0'/'> </eq>
    <eq> <var n='i2'/'> <float v='0.0'/'> </eq>
    <lt> <var n='v1'/'> <var n='v2'/'> </lt>
  </and>
</or>
</defRel>

<!-- an additional finite domain variable for the -->
<!-- behavioural modes 'ok' and 'broken' -->
<defSet n='behaviouralMode'> <symbol v='ok broken'/'> </defSet>

<!-- relation template for a two-mode Ohmic resistor -->
<defRel n='TwoModeOhmicResistor' v='i1 i2 v1 v2 r m'>
  <or>
    <and>
      <eq> <var n='m'/'> <symbol v='ok'/'> </eq>
      <eq>
        <plus> <var n='i1'/'> <var n='i2'/'> </plus>
        <float v='0.0'/'>
      </eq>
      <eq>
        <minus> <var n='v1'/'> <var n='v2'/'> </minus>
        <mult> <var n='i1'/'> <var n='r'/'> </mult>
      </eq>
    </and>
    <and>
      <eq> <var n='m'/'> <symbol v='broken'/'> </eq>
      <eq> <var n='i1'/'> <float v='0.0'/'> </eq>
      <eq> <var n='i2'/'> <float v='0.0'/'> </eq>
    </and>
  </or>
</defRel>

<!-- an additional finite domain variable for the -->
<!-- light status of a bulb -->
<defSet n='lightStatus'> <symbol v='on off'/'> </defSet>

<!-- relation template for a two-mode bulb -->
<defRel n='TwoModeBulb' v='i1 i2 v1 v2 r l m'>
  <or>
    <and>
      <eq>
        <plus> <var n='i1'/'> <var n='i2'/'> </plus>
        <float v='0.0'/'>

```

```

    </eq>
    <eq> <var n='m' /> <symbol v='ok' /> </eq>
    <neq> <var n='i1' /> <float v='0.0' /> </neq>
    <eq>
      <minus> <var n='v1' /> <var n='v2' /> </minus>
      <mult> <var n='i1' /> <var n='r' /> </mult>
    </eq>
    <eq> <var n='l' /> <symbol v='on' /> </eq>
  </and>
  <and>
    <eq>
      <plus> <var n='i1' /> <var n='i2' /> </plus>
      <float v='0.0' />
    </eq>
    <eq> <var n='m' /> <symbol v='ok' /> </eq>
    <neq> <var n='i1' /> <float v='0.0' /> </neq>
    <eq> <var n='v1' /> <var n='v2' /> </eq>
    <eq> <var n='l' /> <symbol v='off' /> </eq>
  </and>
  <and>
    <eq>
      <plus> <var n='i1' /> <var n='i2' /> </plus>
      <float v='0.0' />
    </eq>
    <eq> <var n='m' /> <symbol v='broken' /> </eq>
    <neq> <var n='i1' /> <float v='0.0' /> </neq>
    <eq> <var n='l' /> <symbol v='off' /> </eq>
  </and>
</or>

<!-- an additional finite domain variable for the -->
<!-- two positions of a switch -->
<defSet n='switchPosition'> <symbol v='open closed' /> </defSet>

<!-- relation template for a switch -->
<defRel n='Switch' v='i1 i2 v1 v2 pos'>
  <or>
    <and>
      <eq> <var n='pos' /> <symbol v='closed' /> </eq>
      <eq>
        <plus> <var n='i1' /> <var n='i2' /> </plus>
        <float v='0.0' />
      </eq>
      <eq> <var n='v1' /> <var n='v2' /> </eq>
    </and>
    <and>
      <eq> <var n='pos' /> <symbol v='open' /> </eq>

```

```

    <eq> <var n='i1'/> <float v='0.0'/> </eq>
    <eq> <var n='i2'/> <float v='0.0'/> </eq>
  </and>
</or>

```

B.4 A Small Bus Communication Problem

We list the complete XML file for encoding the bus communication problem depicted in Fig. C.1 and discussed in Sect. C.1.

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE objects SYSTEM 'TestVersion.dtd'>
<objects>
  <!-- definition of the variables' domains -->
  <defSet n='busValues'> <sets/> </defSet>

  <!-- variable definitions -->
  <defVar n='b0in b0out'> <setRef n='busValues'/> </defVar>
  <defVar n='b1in b1out'> <setRef n='busValues'/> </defVar>

  <!-- relations -->
  <and n='Computer0'>
    <in> <int v='1'/> <var n='b0out'/> </in>
    <implies>
      <in> <int v='2'/> <var n='b0in'/> </in>
      <in> <int v='3'/> <var n='b0out'/> </in>
    </implies>
    <implies>
      <in> <int v='3'/> <var n='b0out'/> </in>
      <in> <int v='2'/> <var n='b0in'/> </in>
    </implies>
  </and>

  <and n='Transmission'>
    <eq n='rel2'> <var n='b0out'/> <var n='b1in'/> </eq>
    <eq n='rel3'> <var n='b0in'/> <var n='b1out'/> </eq>
  </and>

  <and n='Computer1'>
    <implies>
      <in> <int v='1'/> <var n='b1in'/> </in>
      <in> <int v='2'/> <var n='b1out'/> </in>
    </implies>
    <implies>
      <in> <int v='2'/> <var n='b1out'/> </in>
      <in> <int v='1'/> <var n='b1in'/> </in>
    </implies>
  </and>

```



```
</objects>
```

B.5 An Electric Circuit with 32 Contexts

Below, the XML file for encoding the electric circuit of Fig. 6.1 is presented. All relation template definitions have been omitted and are the same as in App. B.3.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE objects SYSTEM 'TestVersion.dtd'>

<objects>

  <!-- definition of the variables' domain -->
  <defSet n='switchPosition'> <symbol v='open closed'> </defSet>
  <defSet n='behaviouralMode'> <symbol v='ok broken'> </defSet>
  <defSet n='workingMode'> <symbol v='through blocking'> </defSet>
  <defSet n='lightStatus'> <symbol v='on off'> </defSet>

  <!-- variable definitions -->
  <defVar n='c_SRC v_SRC'> <reals/> </defVar>
  <defVar n='c_GND v_GND'> <reals/> </defVar>
  <defVar n='c1_NSRC c2_NSRC c3_NSRC v1_NSRC v2_NSRC v3_NSRC'>
    <reals/> </defVar>
  <defVar n='c1_NGND c2_NGND c3_NGND v1_NGND v2_NGND v3_NGND'>
    <reals/> </defVar>
  <defVar n='c1_S c2_S v1_S v2_S'> <reals/> </defVar>
  <defVar n='pos_S'> <setRef n='switchPosition'> </defVar>
  <defVar n='c1_B c2_B v1_B v2_B'> <reals/> </defVar>
  <defVar n='mode_B'> <setRef n='behaviouralMode'> </defVar>
  <defVar n='c1_R c2_R v1_R v2_R'> <reals/> </defVar>
  <defVar n='mode_R'> <setRef n='behaviouralMode'> </defVar>
  <defVar n='c1_D c2_D v1_D v2_D'> <reals/> </defVar>
  <defVar n='mode_D'> <setRef n='workingMode'> </defVar>
  <defVar n='light'> <setRef n='lightStatus'> </defVar>
  <defVar n='r_B'> <reals/> </defVar>
  <defVar n='r_R'> <reals/> </defVar>

  <!-- relation templates -->
  ...see App. B.3

  <!-- relations -->
  <eq n='voltageAtSRC'>
    <var n='v_SRC'>
      <interval b='c'> <!-- this is a closed interval -->
        <float v='11.9'> <float v='12.1'>
      </interval>
    </var>
  </eq>
  <eq n='voltageAtGND'> <var n='v_GND'> <float v='0.0'> </eq>
  <eq n='resistanceAtR'>
```

```

    <var n='r_R' />
    <interval b='c'> <!-- this is a closed interval -->
      <float v='90.0' /> <float v='110.0' />
    </interval>
  </eq>
  <eq n='resistanceAtB'>
    <var n='r_B' />
    <interval b='c'> <!-- this is a closed interval -->
      <float v='190.0' /> <float v='210.0' />
    </interval>
  </eq>
  <rel n='NSRC' r='KirchhoffNode'
    v='c1_NSRC c2_NSRC c3_NSRC v1_NSRC v2_NSRC v3_NSRC' />
  <rel n='NGND' r='KirchhoffNode'
    v='c1_NGND c2_NGND c3_NGND v1_NGND v2_NGND v3_NGND' />
  <rel n='S' r='Switch' v='c1_S c2_S v1_S v2_S pos_S' />
  <rel n='B' r='TwoModeBulb'
    v='c1_B c2_B v1_B v2_B r_B light mode_B' />
  <rel n='R' r='TwoModeOhmicResistor'
    v='c1_R c2_R v1_R v2_R r_R mode_R' />
  <rel n='D' r='IdealDiode' v='c1_D c2_D v1_D v2_D mode_D' />
  <rel n='wireSRCtoNSRC' r='Wire' v='c_SRC c1_NSRC v_SRC v1_NSRC' />
  <rel n='wireNGNDtoGND' r='Wire' v='c1_NGND c_GND v1_NGND v_GND' />
  <rel n='wireNSRCtoS' r='Wire' v='c2_NSRC c1_S v2_NSRC v1_S' />
  <rel n='wireStoB' r='Wire' v='c2_S c1_B v2_S v1_B' />
  <rel n='wireBtoNGND' r='Wire' v='c2_B c2_NGND v2_B v2_NGND' />
  <rel n='wireNSRCtoR' r='Wire' v='c3_NSRC c1_R v3_NSRC v1_R' />
  <rel n='wireRtoD' r='Wire' v='c2_R c1_D v2_R v1_D' />
  <rel n='wireDtoNGND' r='Wire' v='c2_D c3_NGND v2_D v3_NGND' />
</objects>

```

B.6 Quadratic Resistors

In the following, we give an XML template definition of a quadratic electric resistor. This has been used in experiments related to Subsect. 6.1.2.

```

<!-- relation template for a quadratic resistor -->
<!-- i...(directed) current variable -->
<!-- v...(undirected) voltage variable -->
<defRel n='QuadraticResistor' v='i1 i2 v1 v2 q'>
  <or>
    <and>
      <eq>
        <plus> <var n='i1' /> <var n='i2' /> </plus>
        <float v='0.0' />
      </eq>
    </and>
  </or>
</defRel>

```

```

    </eq>
    <leq> <float v='0.0'> <var n='i2'> </leq>
    <eq>
      <minus> <var n='v2'> <var n='v1'> </minus>
      <mult>
        <var n='q'>
        <mult> <var n='i2'> <var n='i2'> </mult>
      </mult>
    </eq>
  </and>
  <and>
    <eq>
      <plus> <var n='i1'> <var n='i2'> </plus>
      <float v='0.0'>
    </eq>
    <leq> <float v='0.0'> <var n='i1'> </leq>
    <eq>
      <minus> <var n='v1'> <var n='v2'> </minus>
      <mult>
        <var n='q'>
        <mult> <var n='i1'> <var n='i1'> </mult>
      </mult>
    </eq>
  </and>
</or>
</defRel>

```

Appendix C

Extensions

C.1 Extending the Prototype for Modelling Bus Communication

This section is to emphasise the modular architecture of our prototypic implementation as explicated in previous chapters. We will give an idea how the implementation can be extended to equip it for new, previously unaddressed types of constraints. Naturally, those extensions give rise to new language features that enable the user to actually exploit the advanced facilities.

In order to describe that process, we take a look at a simple example that deals with the problem of modelling and handling bus communication in a network of signal sources. That question has in fact been tackled thoroughly in practice, and latest versions of our prototype are fit to solve a bunch of first problem instances.

Figure C.1 depicts the problem. Two computers may interchange numeric signals via a bus cable. Each computer possesses an in- and an out-bus. Computer 0 is known to emit the signal "1", and to answer any received "2" by a "3". Likewise, computer 1 will answer "1" by "2". These are the constraints stated below the boxes named "Computer 0" and "Computer 1"; cf. Fig. C.1.

Obviously, after a sequence of send and receive steps, computer 1 will have "1" and "3" on its out-bus and "2" on its in-bus. For computer 0 the situation is vice versa, as ensured by the "Transmission" constraint.

How has the problem actually been modelled and solved?

We implemented a new class `SetTerm` that is derived from `NaryExpression`; see Fig. 5.9. Each instance maintains a set of signals, represented by `Values`, that are known to be on the bus, *in*, and a set for those signals that are certainly off, *out*. Note that the two sets represent partial knowledge concerning the bus: With S denoting the exact set of signals on the bus, we only know that

$$in \subseteq S \quad \wedge \quad out \cap S = \emptyset. \quad (*)$$

In Fig. C.1, each computer is modelled by two variables that can be assigned an instance of `SetTerm`. That class implements the intersection of two bus sets,

simply by unioning the *in*'s and the *out*'s, respectively. Clearly, (*) implies that $in \cap out = \emptyset$, and we may utilise that condition to discover conflicts after the intersection of two bus sets in the outlined manner. One may verify that this is also the reason why the top-most join in Fig. C.1 leaves us with only one conjunction instead of $2 \cdot 2 = 4$.

With that infrastructure, we can easily join two assignments of the form $x = s_1$ and $x = s_2$, where s_1 and s_2 are represented by instances of **SetTerm**. Therefore, an aggregation tree as the one shown in Fig. C.1 can easily be built, and the entire problem can hence be handled according to the **RCS** framework.

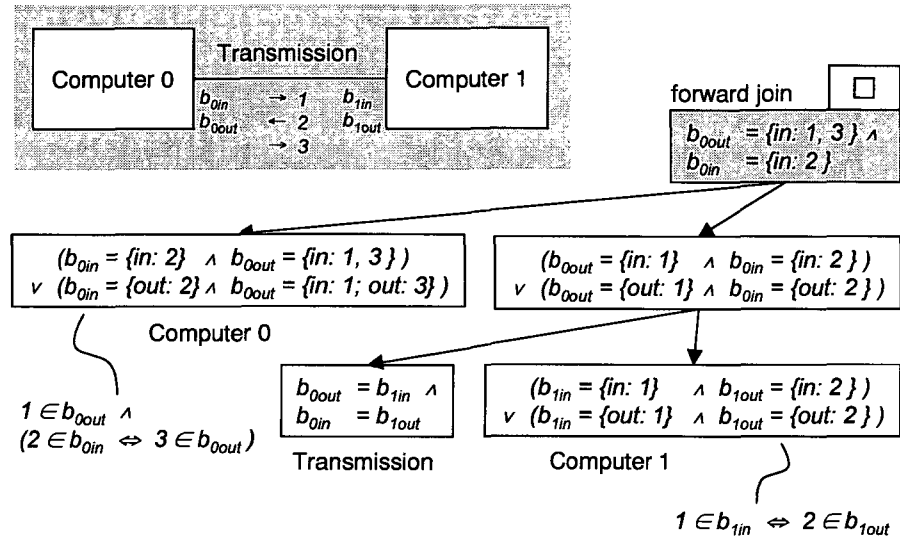


Figure C.1: Aggregation Tree for Two Computers Exchanging Messages

Actually, **SetTerm**, as used in the above example, could have been simply derived from **Value**. This is because there are neither variables nor more involved terms stored in the attributes *in* and *out*. But due to further improvements not mentioned here, it is in our implementation actually possible to include variables in those two sets. This explains why **SetTerm** has not been derived from **Value** but from **Term**.

In order to provide the new capability to the user, we need to extend our constraint language. Obviously, in order to instantiate **SetTerm**, we just need to specify the *in* and the *out* set, by providing the respective elements. Appendix B.4 presents the XML file that encodes the problem depicted in Fig. C.1.

Appendix D

Screenshots of Our Java Implementation of RCS

This part of the appendix presents some screenshots of our prototypic Java implementation of RCS.

Though the implementation is currently used as a hidden module in some first applications inside the DaimlerChrysler AG, all first versions come with a simple browser for inspection, code debugging and displaying computational results. Besides menu items for loading and saving XML problem descriptions, aggregation strategies and context space definitions, there are tabulars for inspecting aggregation forests, successfully parsed constraints, variables and their domains as well as all computed solutions.

D.1 A Showcase Example from Electrics

All screenshots presented in this section show our prototype dealing with the small electric example depicted in Fig. 6.1. As also stated there, the corresponding XML problem description can be found in App. B.5.

D.1.1 The Inspection of Relations

Figure D.1 depicts a screenshot taken after loading the XML problem for the small electric circuit in question. The “Relations” tabulator has been chosen, and so the user may inspect all parsed constraints. The selected constraint “B”, representing the only bulb in the circuit, is expanded on the right-hand side. This panel shows a disjunction with three cases.

In total, there are 18 constraints that provide an exhaustive description of the circuit and enable the prototype to find solutions for all unknowns.

D.1.2 Building an Aggregation Forest

After loading the problem, we may use the appropriate menu item to solve the problem. This will deploy the machinery developed in this thesis and run the forward

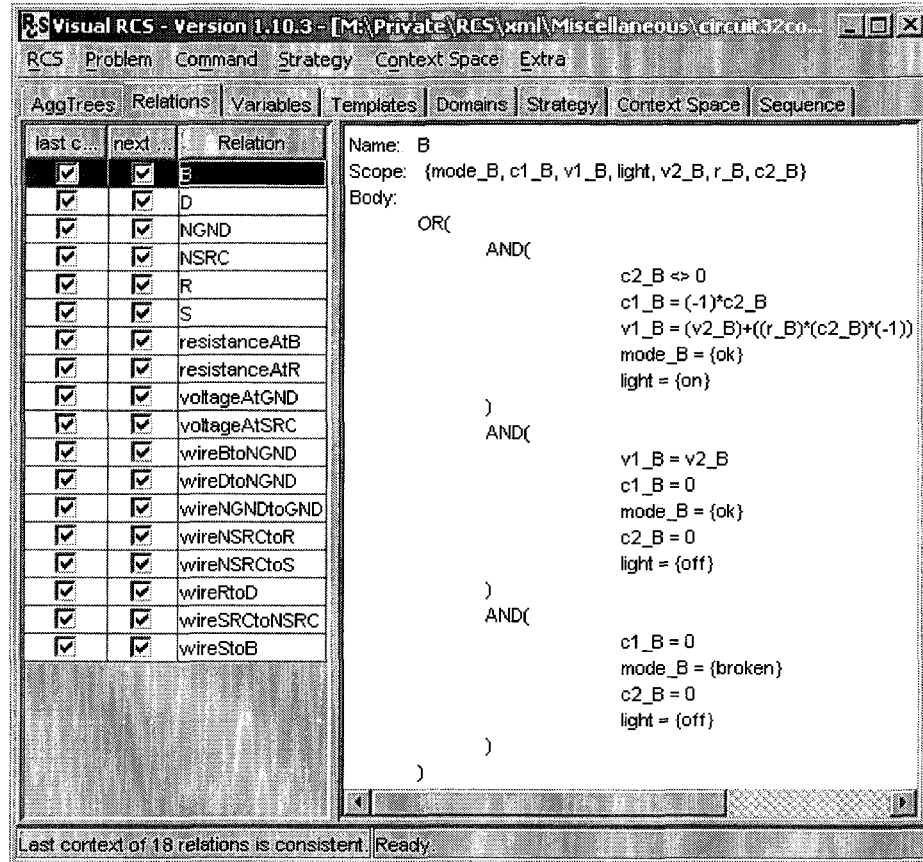


Figure D.1: A Screenshot Taken During the Inspection of Constraints

phase. Additionally, in case of consistence¹, all variables will be solved by means of the backward phase.

The forward phase produces an aggregation forest for the problem. After the backward phase has terminated, the forest's nodes will also hold references to all backward relations. The left-hand side of Fig. D.2 shows the resulting forest in an almost entirely expanded state.

There is just one tree in the forest: It has a *full* root and signals thus consistency; cf. Th. 1. The top line captures a total computation time of 411 milliseconds, which splits into 50 milliseconds and 361 milliseconds for forward and backward phase, respectively. The leading "10s" is to say that the utilisation of the built-in on-the-fly aggregation strategy took 10 milliseconds.

By selecting a node, the user may inspect forward and backward relation, as well as the stored forward join; cf. Fig. 4.5. In Fig. D.2, the right-hand side shows these relations for the aggregation of the voltage setting at ground and the wire connecting ground to the Kirchhoff node NGND.

¹Remember again that this only means that our prototype failed to detect inconsistency.

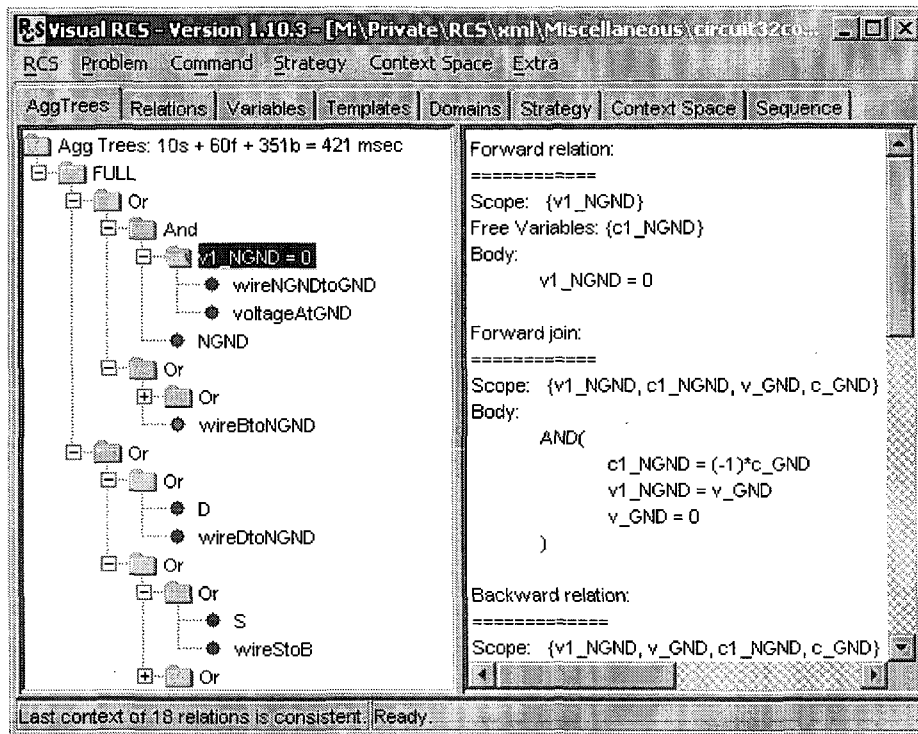


Figure D.2: An Aggregation Forest for the Circuit in Fig. 6.1

Visual RCS - Version 1.10.3 - [M:\Private\RCS\xml\Miscellaneous\circuit32co...]

RCS Problem Command Strategy Context Space Extra

AggTrees Relations Variables Templates Domains Strategy Context Space Sequence

Variable Name	Domain	Solution
c3_NGND	(-oo, +oo)	{0, [0.10818182, 0.13444445]}
c3_NSRC	(-oo, +oo)	{[-0.13444445, -0.10818182], 0}
c_GND	(-oo, +oo)	{0, [0.05666667, 0.06368421], [0.10818182, 0.13444445], [0.16...
c_SRC	(-oo, +oo)	{[-0.19812866, -0.16484849], [-0.13444445, -0.10818182], [-0.0...
light	{on, off}	{on, off}
mode_B	{ok, broken}	{ok, broken}
mode_D	{through, blocking}	{through, blocking}
mode_R	{ok, broken}	{ok, broken}
pos_S	{open, closed}	{open, closed}
r_B	(-oo, +oo)	[190, 210]
r_R	(-oo, +oo)	[90, 110]

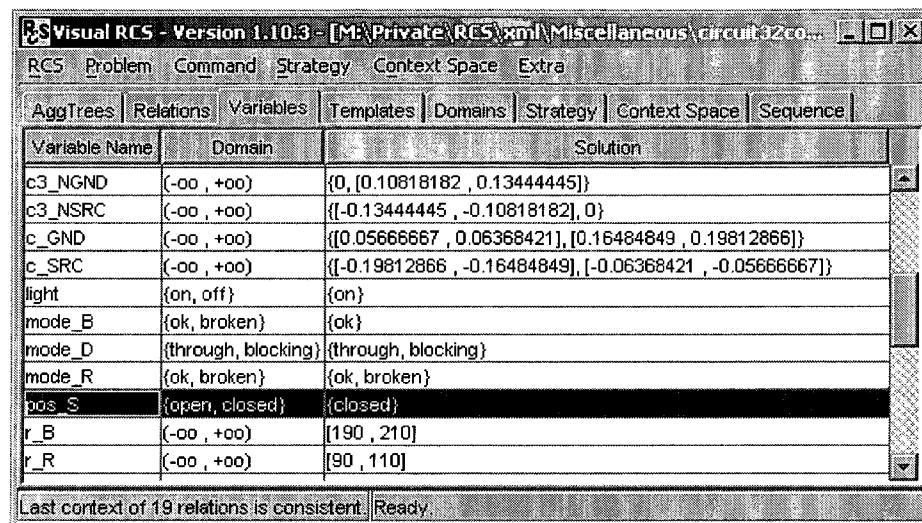
Last context of 18 relations is consistent. Ready.

Figure D.3: Solutions for All Unknowns

D.1.3 Obtaining Tightest Variable Restrictions

Also, after having solved the problem, the “Variables” tabulator will list solutions for all the variables in the original problem description; see Fig. D.3. The first two columns show names and domains. The tightest restriction of the domain, as set by the constraint problem, can be found in the “Solution” column.

Note that this may be a set of numbers and intervals, as in the case of the selected variable: *c_GND*, the current “flowing into ground”, may be zero or take its values in one of the three shown intervals.



Variable Name	Domain	Solution
c3_NGND	$(-\infty, +\infty)$	{0, [0.10818182, 0.13444445]}
c3_NSRC	$(-\infty, +\infty)$	{[-0.13444445, -0.10818182], 0}
c_GND	$(-\infty, +\infty)$	{[0.05666667, 0.06368421], [0.16484849, 0.19812866]}
c_SRC	$(-\infty, +\infty)$	{[-0.19812866, -0.16484849], [-0.06368421, -0.05666667]}
light	{on, off}	{on}
mode_B	{ok, broken}	{ok}
mode_D	{through, blocking}	{through, blocking}
mode_R	{ok, broken}	{ok, broken}
pos_S	{open, closed}	{closed}
r_B	$(-\infty, +\infty)$	[190, 210]
r_R	$(-\infty, +\infty)$	[90, 110]

Last context of 19 relations is consistent. Ready.

Figure D.4: Solutions for the Altered Problem with the Bulb Being Lit

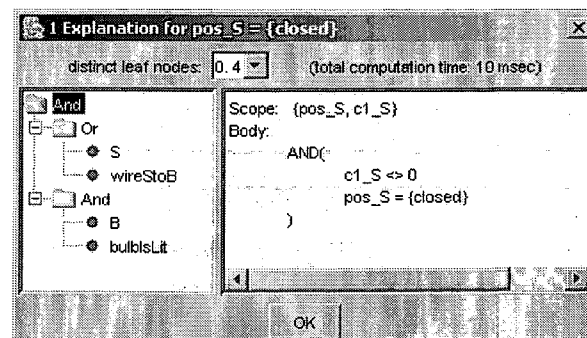


Figure D.5: A Minimal Explanation of Size 4 for the Switch Being Closed

D.1.4 Retrieving a Minimal Explanation

Figure D.4 presents a similar screenshot as Fig. D.3: Here, the problem has been slightly altered by adding a new constraint (`bulbIsLit`) that captures the observation that bulb B is lit.

Note that, consequently, `mode.B` and `pos.S` have been found to be *ok* and *closed*, respectively; see Fig. 6.1.

Though intuitively clear, we may obtain a minimal explanation for the latter fact, by deploying the appropriate menu item. This gives us the explanation window depicted in Fig. D.5. The explanation has been found within 10 milliseconds, and consists - as expected - of four constraints; the added observation (`bulbIsLit`), the component constraints for B and S and the connecting wire.

D.2 Term Rewrites for the Basic Parallel Circuit

As mentioned in Subsect. 6.1.2, our Java prototype is able to solve the constraint problem that captures the basic electric circuit with two parallel quadratic resistors. After having solved that problem, we may retrieve the set of applied term rewrites, as shown in Fig. D.6.

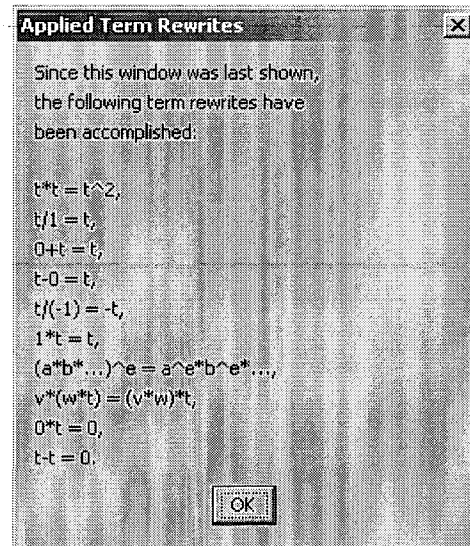
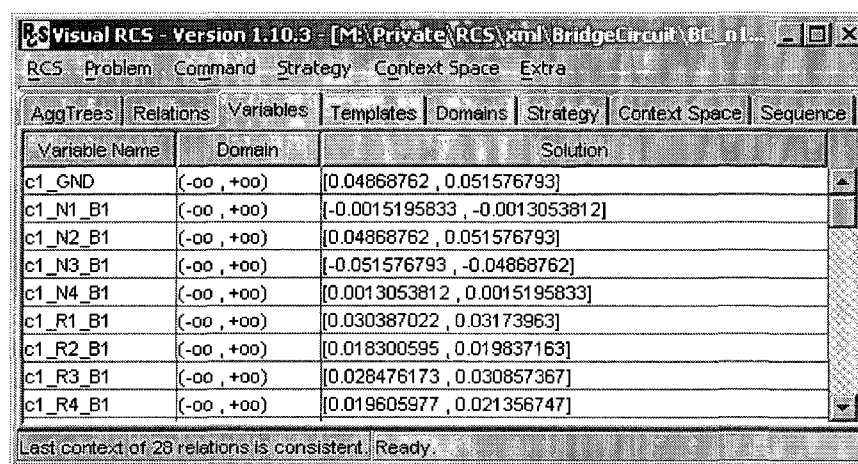


Figure D.6: All Term Rewrites Applied to Solve a Simple Quadratic Circuit

D.3 The Utilisation of Monotonous Terms

In Subsect. 6.1.3., we have explained when and how RCS can utilise extra knowledge concerning the monotonicity of functional dependencies in the constraint network to be solved. On the settings panel of RCS the user may mark the next constraint

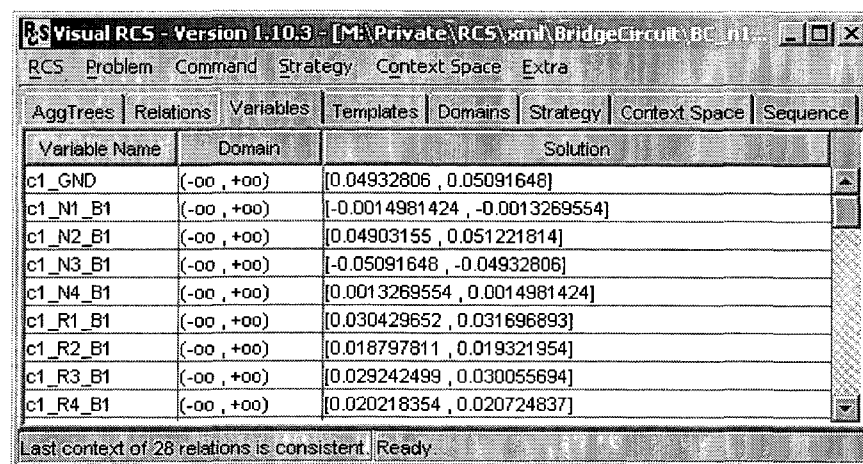
problem as one in which all unknowns depend monotonically on the inputs. A typical example is a resistive network in which all current and voltage variables happen to be monotonous functions of the input resistances. Figure D.7 and Fig. D.8 display the solutions for the bridge circuit problem R_1 with interval values for the ideal voltage source and for all 5 resistances in the circuit. Consequently, RCS determines all unknowns to lie within some interval. Figure D.7 shows the results based on naive interval calculus. With the extra knowledge concerning monotonicity, RCS



Variable Name	Domain	Solution
c1_GND	$(-\infty, +\infty)$	[0.04868762, 0.051576793]
c1_N1_B1	$(-\infty, +\infty)$	[-0.0015195833, -0.0013053812]
c1_N2_B1	$(-\infty, +\infty)$	[0.04868762, 0.051576793]
c1_N3_B1	$(-\infty, +\infty)$	[-0.051576793, -0.04868762]
c1_N4_B1	$(-\infty, +\infty)$	[0.0013053812, 0.0015195833]
c1_R1_B1	$(-\infty, +\infty)$	[0.030387022, 0.03173963]
c1_R2_B1	$(-\infty, +\infty)$	[0.018300595, 0.019837163]
c1_R3_B1	$(-\infty, +\infty)$	[0.028476173, 0.030857367]
c1_R4_B1	$(-\infty, +\infty)$	[0.019605977, 0.021356747]

Last context of 28 relations is consistent. Ready.

Figure D.7: Non-Monotonous Solutions for the Bridge Circuit with one Box



Variable Name	Domain	Solution
c1_GND	$(-\infty, +\infty)$	[0.04932806, 0.05091648]
c1_N1_B1	$(-\infty, +\infty)$	[-0.0014981424, -0.0013269554]
c1_N2_B1	$(-\infty, +\infty)$	[0.04903155, 0.051221814]
c1_N3_B1	$(-\infty, +\infty)$	[-0.05091648, -0.04932806]
c1_N4_B1	$(-\infty, +\infty)$	[0.0013269554, 0.0014981424]
c1_R1_B1	$(-\infty, +\infty)$	[0.030429652, 0.031696893]
c1_R2_B1	$(-\infty, +\infty)$	[0.018797811, 0.019321954]
c1_R3_B1	$(-\infty, +\infty)$	[0.029242499, 0.030055694]
c1_R4_B1	$(-\infty, +\infty)$	[0.020218354, 0.020724837]

Last context of 28 relations is consistent. Ready.

Figure D.8: Monotonous Solutions for the Bridge Circuit with one Box

manages to find the *exact*, that is, narrowest intervals for all unknowns. Cf. Fig. D.8 which lists as solution intervals proper subsets of the findings of Fig. D.7. The cost

for the higher accuracy amounts to a runtime that is about four times as long as in the non-monotonous case of naive interval calculus.

D.4 Obtaining both Minimal Conflicts for the 3-Queens Problem

In Ex. 6 we have had a look at the prominent n -queens problem. Figure 3.5 depicts an aggregation tree proving that there is no solution for $n = 3$.

We may use the explanatory facilities of RCS to retrieve all minimal conflicts. Note that Fig. 3.7 and the subsequent discussion come to the conclusion that there are exactly two minimal conflicts: As a first possibility, it suffices to join the four relations (and no less) shown at the top left in Fig. D.9, in order to produce the empty relation. Secondly, the bottom right window provides us with the other minimal conflict of size 5.

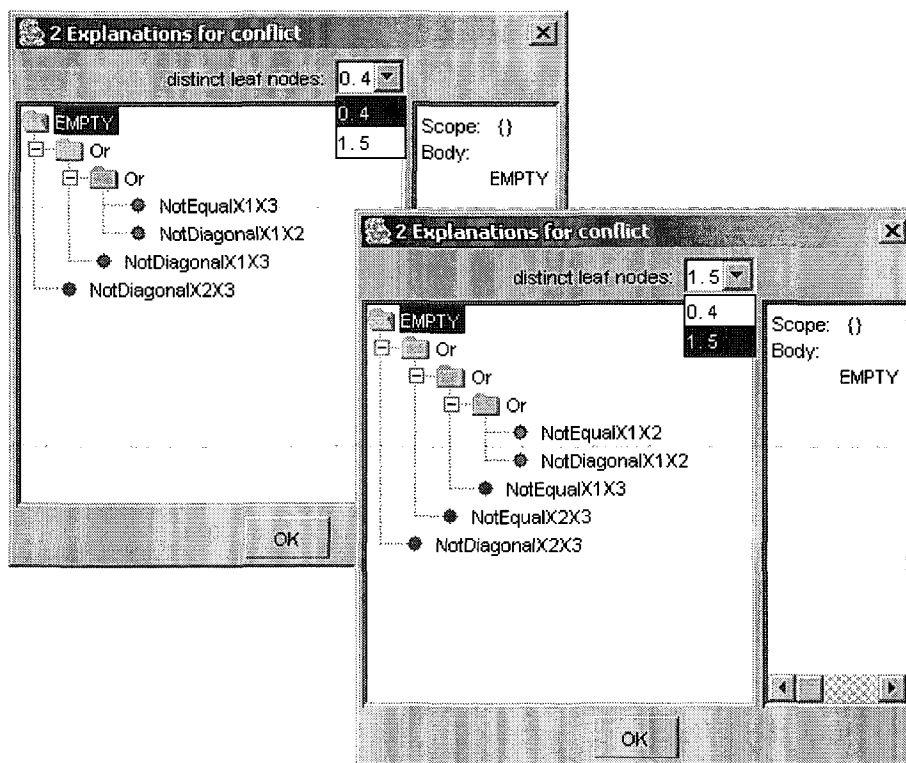


Figure D.9: Both Minimal Conflicts for the 3-Queens Problem

Note that - despite the suggestion supported by the figure - RCS will at runtime open only one window. The user can then freely navigate among all minimal conflicts by means of the shown combo box. Therein, the conflicts are named by an ordering number (starting with 0), a separating dot and finally by the size of the respective conflict. In our example, that naming convention produces the conflict

identifiers “0.4” and “1.5”; cf. Fig. D.9.

D.5 Solving All Instances in a Context Space

With our prototypic implementation of RCS we are able to specify entire context spaces of related constraint problems. This can be done either by reading an appropriate XML fragment that defines the fixed portion and the one-ofs of the respective context space; cf. Def. 17. Or the user may deploy some shortcut functionality implemented in the prototype, for recognising a context space. This latter possibility works whenever the user follows a special naming convention for the relations: All alternatives of a one-of must start with the same prefix, e.g. with “y”. The program recognises that a relation is an alternative of a one-of, if and only if that prefix is

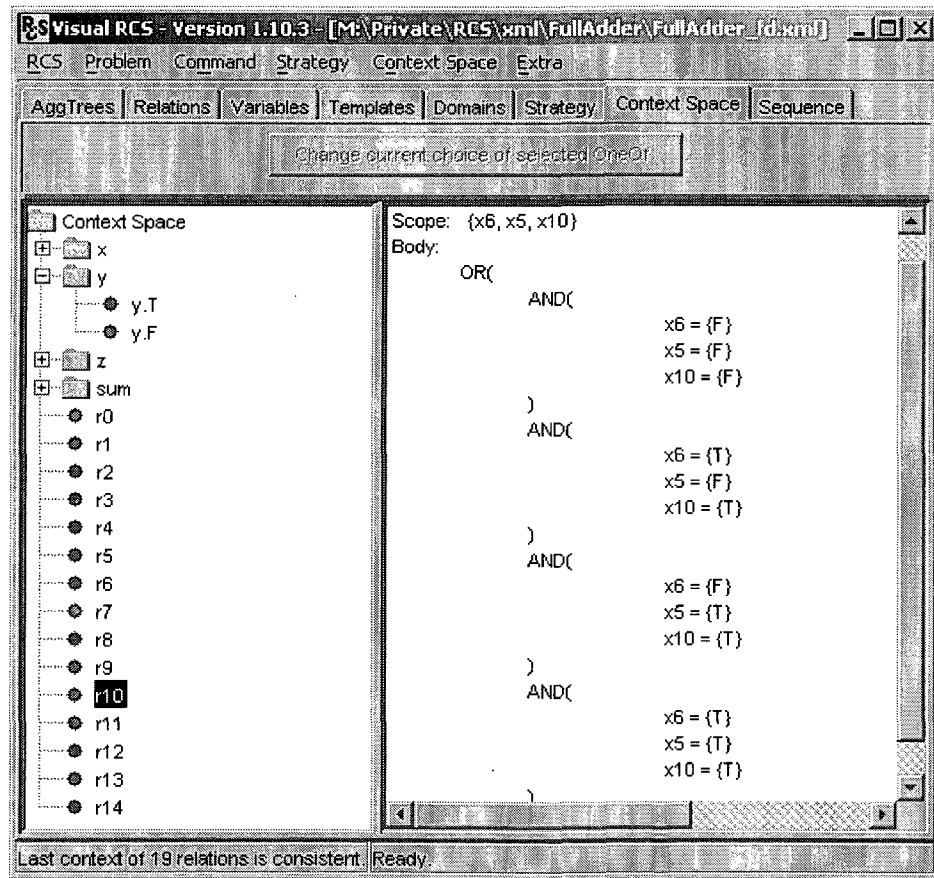


Figure D.10: Context Space for the 1-Bit Full Adder in Fig. B.1

followed by a dot “.”.

Figure D.10 shows a recognised context space that captures all 16 contexts for the 1-bit full adder shown in Fig. B.1. As the above example says, two relations starting with “y.” have been identified as the alternatives in a one-of.

The figure shows furthermore the or-relation r_{10} which relates the variables x_5 , x_6 and x_{10} and which has as body a table of four alternative sets of assignments. By using the appropriate menu item, the prototype will sequentially solve all 16 contexts in the problem. Unless turned off, the implementation utilises the discussed reuse facilities. Solving each context here means to record the runtimes for forward and backward phase (including an extra measurement for the built-in aggregation strategy) and all solutions for all variables in all contexts that have not been found to be inconsistent.

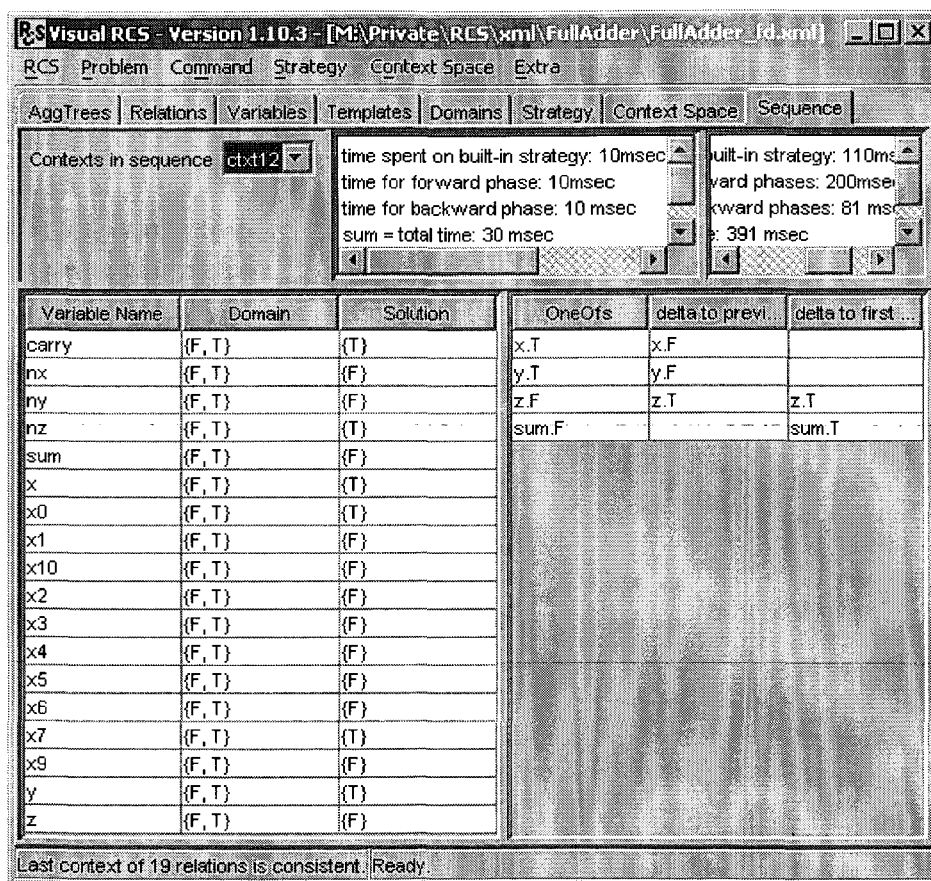


Figure D.11: Solution Table for the Context Space in Fig. D.10

This view is - again for the 1-bit full adder - presented in Fig. D.11. The screenshot shows that the user can select each context via a combo box. After the selection, the lower left table will present all solutions for all variables. The lower right area provides the user with information as to which alternatives of the one-ofs were active in the selected context. In Fig. D.11 these are " $x.T$ ", " $y.T$ ", " $z.F$ ", " $sum.F$ ", capturing the facts that x and y were set to true and z and sum to false. Note that this implies that $carry$ takes as well the value true which is reflected in the first row of the lower left table.

The upper middle panel shows the runtimes for strategy, forward phase (including the strategic portion), backward phase and the entire context. On the upper right sheet, the corresponding cumulative measurements for all 16 contexts are presented. There, we see that solving all 16 contexts took 391 milliseconds.

Appendix E

Structural Runtime Figures

E.1 Structural Data for the Analysis of $(R_k)_{k \in \mathbb{N}_+}$

Figure E.1 shows the average numbers of atoms per leaf node and non-leaf node, measured during the analysis of the problem instances $R_{10}, R_{20}, \dots, R_{100}$. The point

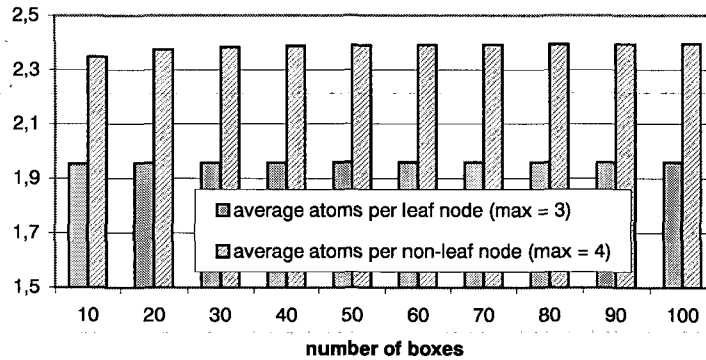


Figure E.1: Number of Atoms per Node; $(R_k)_{k \in \{10, 20, \dots, 100\}}$

to be made here is that non-leaf relations are more complex than leaf relations: They contain, in their disjunctive normal form, about half an atom more than leaf relations. Still, as Fig. 6.3 suggests, that increase in complexity is too small to negatively affect the scaling behaviour of our prototype for the family $(R_k)_{k \in \mathbb{N}_+}$: We obtain a runtime that is “almost linear” in k .

E.2 Structural Data for the Analysis of $(D_k)_{k \in \mathbb{N}_+}$

The analysis of the family $(D_k)_{k \in \mathbb{N}_+}$ reveals significant differences to that of $(R_k)_{k \in \mathbb{N}_+}$. Each circuit D_k captures basically 4^k possibilities for the current to flow through the system. The reason for that are the two diodes per box, each of which opens two potential scenarios. The practical problem implied by that characteristic is that any solver will have to deal with 4^k cases, unless it is able to keep the number of

possibly cases small at runtime.

Depending on how RCS builds its aggregation trees, it may or may not be able to discover inconsistent cases earlier and thus reduce the overall runtime for the problem instance D_k . So far, we have not implemented and studied a special aggregation strategy that attempts to minimise the number of disjuncts.

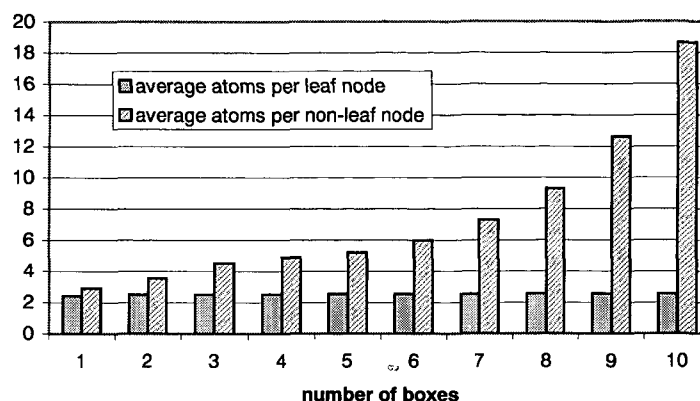


Figure E.2: Average Number of Atoms per Node; $(D_k)_{k \in \{1,2,\dots,10\}}$

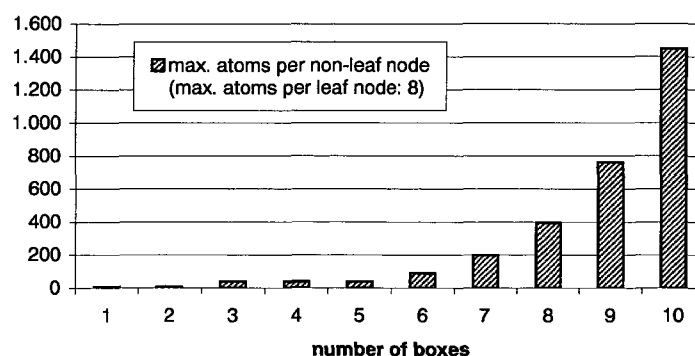


Figure E.3: Maximum Number of Atoms per Non-Leaf Node; $(D_k)_{k \in \{1,2,\dots,10\}}$

Consequently, Figs. E.2 and E.4 show that our prototype does not behave well for the family $(D_k)_{k \in \mathbb{N}_+}$: Both the total number of atoms and the number of disjuncts in an average non-leaf node rise when an aggregation tree is built. Therefore, the complexity of non-leaf relations increases as the tree grows. As a result, we will normally not observe a runtime linear in k .

Figures E.3 and E.5 show that - along with an increasing *average* complexity of non-leaf relations - we get far worse behaviour for the *most* complex non-leaf relations: For $k = 10$, there are even over 1400 atoms or over 120 disjuncts in the “worst” relation.

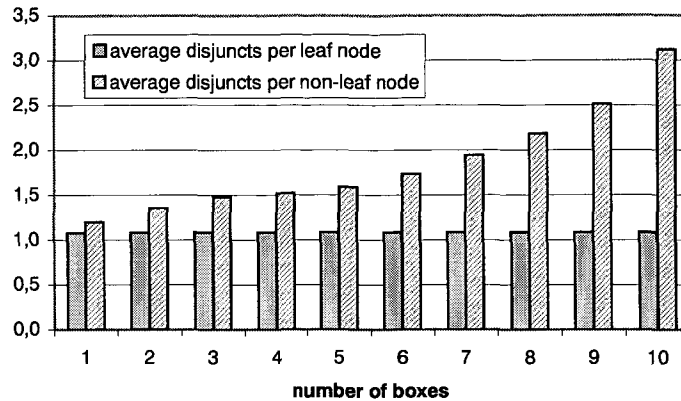


Figure E.4: Average Number of Disjuncts per Node; $(D_k)_{k \in \{1,2,\dots,10\}}$

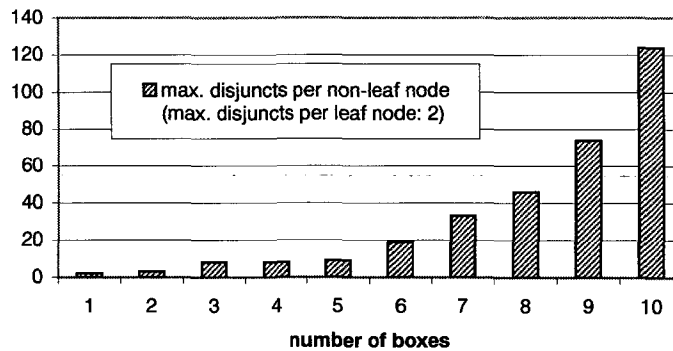


Figure E.5: Maximum Number of Disjuncts per Non-Leaf Node; $(D_k)_{k \in \{1,2,\dots,10\}}$

E.3 Structural Data for the Analysis of the ATV Family

By taking a good look at Fig. E.6 we learn that our prototypic implementation of RCS behaves well for the ATV propulsion system, and as forecasted by our theoretical investigations in Chap. 3.

First of all, there are only minor deviations between the characteristics of leaf and non-leaf relations. This shows that aggregation can indeed preserve the complexity of the initial relations, both in terms of variables per relation and in terms of atoms, conjuncts and disjuncts. Also, the figures for incremental analysis, that is, for reuse-based analysis are only a little higher than those for an analysis from scratch.

A final observation concerns the balancedness of the evolving aggregation trees: In the incremental case, trees are being repaired over and over again which inevitably results in less balanced trees. Figure E.6 shows these parameters in the rows named “tree depth”. (The second value denotes the maximum observation, i.e. the maximal number of nodes in a root-leaf path.) As has already been pointed out in Sub-

Structural Analysis	* Average Measurements of 1000 Runs			
	System1	System2	System3	System4
model properties				
components	13	31	39	62
relations	122	288	395	598
variables	142	345	474	718
leaf relations (average; max.)				
atoms per relation	2.71; 14	2.67; 14	2.81; 14	2.75; 14
conjuncts per disjunct	1.79; 3	1.79; 3	1.83; 3	1.81; 3
disjuncts per relation	1.52; 5	1.49; 5	1.54; 5	1.52; 5
variables per relation	2.42; 6	2.46; 6	2.50; 6	2.48; 6
non-leaf relations (*; max.)	non-incremental case			
atoms per relation	2.16; 13	2.02; 13	2.77; 62	2.49; 57
conjuncts per disjunct	1.85; 5	1.82; 5	2.26; 8	2.10; 8
disjuncts per relation	1.16; 4	1.11; 4	1.23; 15	1.18; 10
variables per relation	2.76; 6	3.06; 7	3.29; 12	3.23; 11
tree depth	12.75; 20	14.73; 22	13.41; 26	14.32; 29
	incremental case			
atoms per relation	2.35; 13	2.14; 13	2.83; 75	2.49; 80
conjuncts per disjunct	1.93; 5	1.89; 5	2.24; 9	2.08; 9
disjuncts per relation	1.22; 4	1.14; 4	1.26; 16	1.20; 15
variables per relation	2.83; 6	3.16; 6	3.29; 12	3.20; 10
tree depth	12.90; 22	16.44; 32	13.92; 34	15.86; 39

Figure E.6: Structural Data for the Analysis of the ATV Systems

sect. 6.4.2, the incremental analysis of 1000 ATV contexts included very few (less than 10) restarts from scratch, due to very unbalanced aggregation trees. Therefore, the results given in Fig. E.6 need to be interpreted by taking into account a small number of non-incremental restarts.

Bibliography

- [1] 3DCS. World Wide Web. <http://www.3dcs.com/>.
- [2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, Reading, Massachusetts, USA, 1995. ISBN 0-201-53771-0.
- [3] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csp - application to configuration. In *ECAI Workshop Notes, Modelling and Solving Problems with Constraints*, Berlin, Germany, August 22 2000.
- [4] Hauke Arndt, Frank Feldkamp, Michael Heinrich, and Klaus Dieter Meyer-Gramann. System design for reusability - task, current state, and future research. In *ECAI Workshop Notes, Knowledge-based Systems for Model-Based Engineering*, Berlin, Germany, August 22 2000.
- [5] Eric Bensana, Taufiq Mulyanto, and Gérard Verfaillie. Dealing with uncertainty in design and configuration problems. In *ECAI Workshop Notes, Configuration*, Berlin, Germany, August 21 2000.
- [6] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Verlag Nauka, Moskau & BSB B.G. Teubner Verlagsgesellschaft, Leipzig, DDR, 1989. ISBN 3-322-00259-4.
- [7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume 35, pages 677–691, August 1986.
- [8] CATIA. World Wide Web. <http://www.catia.com/>.
- [9] A. Colmerauer. Naive solving of non-linear constraints. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 89–112. MIT Press, London, 1993. ISBN 0-262-02353-9.
- [10] Computer Algebra Information Network. World Wide Web. <http://www.can.nl/>.
- [11] Computer Algebra Systems. World Wide Web. <http://sal.kachinatech.com/A/1/>.
- [12] Adnan Darwiche. Model-based diagnosis using structured system descriptions. In *Journal of Artificial Intelligence Research (JAIR)*, volume 8, pages 165–222, 1998.

- [13] Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *Proceedings of the Sixteenth IJCAI*, volume 1, pages 284–289, Stockholm, Sweden, 1999.
- [14] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra - Systems and Algorithms for Algebraic Computation*. Academic Press Inc., San Diego, CA 92101, USA, 1993. ISBN 0-12-209232-8.
- [15] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting. *Readings in Model-based Diagnosis*, pages 3–24, 1992. ISBN 1-55860-249-6.
- [16] Johan de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.
- [17] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Readings in Model-based Diagnosis*, pages 124–130. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992. ISBN 1-55860-249-6.
- [18] Rina Dechter. Bucket elimination: A unifying framework for structure-driven inference. *Learning and Inference in Graphical Models*, 1998.
- [19] *ECAI Workshop Notes, Knowledge-based Systems for Model-Based Engineering*, Berlin, Germany, August 22 2000.
- [20] *ECAI Workshop Notes, Modelling and Solving Problems with Constraints*, Berlin, Germany, August 22 2000.
- [21] *ECAI Workshop Notes, Configuration*, Berlin, Germany, August 21 2000.
- [22] Yousri El Fattah. An elimination algorithm for model-based diagnosis. *DX-98, Ninth International Workshop on Principles of Diagnosis*, pages 47–54, 1998.
- [23] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *IJCAI*, pages 1742–1749, 1995.
- [24] Frank Feldkamp, Michael Heinrich, and Klaus Dieter Meyer-Gramann. Product models and reusability. *AAAI'99 Workshop on Configuration*, July 19 1999.
- [25] Frank Feldkamp, Michael Heinrich, and KlausDieter Meyer-Gramann. SyDeR - system design for reusability. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AI-EDAM, 1998*, September 1998.
- [26] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *14h European Conference on Artificial Intelligence (ECAI'2000)*, pages 146–150, 2000.
- [27] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. The MIT Press, Cambridge, Massachusetts, USA, 1993. ISBN 0-262-06157-0.
- [28] J.-B.J. Fourier. Analyse des travaux de l'academie royale des sciences, pendant l'annee 1824, partie mathematique. *Histoire de l'Academie Royale des Sciences de l'Institut de France*, 7:xlvi–lv, 1827.

- [29] Martin Fowler and Kendall Scott. *UML Distilled - A Brief Guide to the Standard Object Modeling Language, Second Edition*. Addison-Wesley, Reading, Massachusetts, USA, October 1999. ISBN 020165783X.
- [30] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. In *39th Annual Symposium on Foundations of Computer Science*, page p.706, Palo Alto, California, USA, November 8-11 1998.
- [31] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394-399. Morgan Kaufmann, 1999.
- [32] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decomposition and tractable queries. *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21-32, May 31 - June 03 1999.
- [33] Ian S. Graham and Liam Quin. *XML Specification Guide*. John Wiley & Sons Publishing, New York, USA, 1999. ISBN 0-471-32-753-0.
- [34] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in Model-based Diagnosis*. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992. ISBN 1-55860-249-6.
- [35] Warwick Harvey, Peter J. Stuckey, and Alan Borning. Fourier elimination for compiling constraint hierarchies. *Constraints*, 7(2):199-219, April 2002.
- [36] Joachim Hollman and Lars Langemyr. Algorithms for non-linear algebraic constraints. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 113-131. MIT Press, London, 1993. ISBN 0-262-02353-9.
- [37] Hoon Hong. Risc-CLP(Real): Logic programming with non-linear constraints over the reals. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 133-159. MIT Press, London, 1993. ISBN 0-262-02353-9.
- [38] John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson, and Hak-Jin Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 136-141. AAAI, The AAAI Press / The MIT Press, July 1999. ISBN 026251106-1.
- [39] E. Hyvonen and S. De Pascale. Interval computations on the spreadsheet. *Applications of Interval Computations*, pages 169-210, 1996.
- [40] Eero Hyvonen. Constraint reasoning based on interval arithmetic. *Proceedings of IJCAI-89*, pages 1193-1198, 1989.

- [41] ILOG S.A. *ILOG OPL STUDIO 3.0.2*, 2000. User Manual.
- [42] Jean-Louis Imbert. About redundant inequalities generated by fourier's algorithm. In P. Jorrand and V. Sgurev, editors, *Artificial Intelligence IV: Methodology, Systems, Applications*, pages 117–127. North-Holland, Amsterdam, 1990.
- [43] Jean-Louis Imbert and Pascal van Hentenryck. On the handling of disequations in CLP over linear rational arithmetic. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 49–72. MIT Press, London, 1993. ISBN 0-262-02353-9.
- [44] Sebastian Iwanowski. An algorithm for model-based diagnosis that considers time. In *Annals of Mathematics and Artificial Intelligence*, volume 11, pages 415–437, 1994.
- [45] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [46] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Projecting clp(r) constraints. *New Generation Computing*, 11:449–469, 1993.
- [47] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(R language and system). In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 14, pages 339–395, July 1992.
- [48] Ulrich John. *Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung*. PhD thesis, Technical University of Berlin, Berlin, Germany, February 22 2002.
- [49] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints (CONS-1)*, pages 75–82, 2001.
- [50] Narendra Jussien. e-constraints: Explanation-based constraint programming. In *CP'01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, December 2001.
- [51] Vladik Kreinovich, Anatoly Lakeyev, Jiří Rohn, and Patrick Kahl. *Computational Complexity and Feasibility of Data Processing and Interval Computations*. Kluwer Academic Publishers, P. O. Box 17, 3300 AA Dordrecht, The Netherlands, 1998. ISBN 0-7923-4865-6.
- [52] Bernhard Kutzler, Franz Lichtenberger, and Franz Winkler. *Softwaresysteme zur Formelmanipulation - Praktisches Arbeiten mit den Computer-Algebra-Systemen REDUCE, MACSYMA, DERIVE*. Kontakt & Studium, Band 310. Expert Verlag, Ehningen bei Bblingen, Germany, 1990. ISBN 3-8169-0445-9.
- [53] Yahia Lebbah, Michel Rueher, and Claude Michel. A global filtering algorithm for handling systems of quadratic equations and inequations. In Pascal van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP*

- 2002, *Proceedings*, pages 109–123, Ithaca, N.Y., USA, September 9–13 2002. Springer. ISBN 3-540-44120-4.
- [54] H. Leemhuis, R. Baumann, U. Kaufmann, F. Swoboda, T. Kühn, and Z. Ragan. Function-oriented product modelling based on feature technology and integrated constraint management. *Product Data Technology Europe 2002*, 11th Symposium (PDT Europe 2002), May 2002.
 - [55] David Maier. *The Theory of Relational Databases*. Computer Science Press, 11 Taft Court, Rockville, Maryland 20850, USA, 1983. ISBN 0-914894-42-0.
 - [56] Jakob Mauss. Local analysis of linear networks by aggregation of characteristic lines. 9th *International Workshop on Principles of Diagnosis (DX98)*, pages 78–85, 1998.
 - [57] Jakob Mauss, Volker May, and Mugur Tatar. Towards model-based engineering: Failure analysis with MDS. *ECAI-2000 Workshop on Knowledge-Based Systems for Model-Based Engineering*, August 22 2000.
 - [58] Jakob Mauss and M. Sachenbacher. Conflict-driven diagnosis using relational aggregations. In R. Milne, editor, *Working Papers of the 10th International Workshop on Principles of Diagnosis (DX'99)*, Loch Awe Hotel, Scotland, Great Britain, June 8–11 1999. ISBN 0-903878-55-0.
 - [59] Jakob Mauss and Mugur Tatar. Computing minimal conflicts for rich constraint languages. 15th *European Conference on Artificial Intelligence ECAI 2002*, 2002.
 - [60] William J. Older and André Velino. Constraint arithmetic on reals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. MIT Press, 1993. ISBN 0262023539.
 - [61] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Ellis Horwood Limited, Chichester, Market Cross House, Cooper Street, Chichester, West Sussex, PO191EB, England, 1984.
 - [62] M. Reischl, G. Baratoff, B. Kounovsky, and Carsten Schön-Schmidt. Straßenbahndiagnose mobil gemacht - ein mobiles, AR-basiertes diagnosesystem. submitted to 1. Paderborner Workshop AR/VR in der Produktenstehung.
 - [63] J. E. Sammet. Software for nonnumerical mathematics. In John R. Rice, editor, *Mathematical Software*, ACM Monograph Series, pages 295 – 330, 111 Fifth Avenue, New York 10003, USA, 1971. Academic Press Inc.
 - [64] Stuart Charles Shapiro, editor. *Encyclopedia of Artificial Intelligence*, volume 1. Wiley-Interscience Publication, John Wiley & Sons, New York, USA, 1987. ISBN 0-471-62974-x.
 - [65] J. Smit. The efficient calculation of symbolic determinants. In *Proceedings of the ACM Symposium on Symbolic and Algebraic Computation*, pages 105–113, Yorktown Heights, N.Y., USA, August 10–12 1976.

- [66] J. Smit. A cancellation free algorithm, with factoring capabilities, for the efficient solution of large sparse sets of equations. In *Proceedings of the 4th ACM Symposium on Symbolic and Algebraic Computation*, pages 146–154, Snowbird, Utah, USA, 1981.
- [67] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *Artificial Intelligence*, 127(1):1–29, 2001.
- [68] Mugur Tatar. Diagnosis with cascading defects. *DX-95, Sixth International Workshop on Principles of Diagnosis*, October 2-4 1995.
- [69] Mugur Tatar. *Dependent Defects and Aspects of Efficiency in Model-Based Diagnosis*. PhD thesis, University of Hamburg, Hamburg, Germany, 1997.
- [70] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 24-28 Oval Road, London NW1 7DX, 1996. ISBN 0-12-701610-4.
- [71] Pascal van Hentenryck, David McAllester, and Deepak Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, April 1997.
- [72] Pascal van Hentenryck, Laurent Michel, and Yves Deville. *Numerica - A Modeling Language for Global Optimization*. The MIT Press, Cambridge, Massachusetts, USA, 1997. ISBN 0-262-72027-2.
- [73] Nageshwara Rao Vempaty. Solving constraint satisfaction problems using finite state automata. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 453–458, 1992.
- [74] XML - Extensible Markup Language. World Wide Web. <http://www.w3.org/XML/>.
- [75] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, Proceedings*, pages 82–94, Cannes, France, September 9-11 1981. IEEE Computer Society Press.
- [76] Neil Yorke-Smith and Carmen Gervet. On constraint problems with incomplete or erroneous data. In Pascal van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, Proceedings*, pages 732–737, Ithaca, N.Y., USA, September 9-13 2002. Springer. ISBN 3-540-44120-4.

Curriculum Vitae

Angaben zur Person

Name: Frank Seelisch
geboren: am 30.04.1971 in Sömmerda, Thüringen, als drittes Kind von Hans Seelisch, Diplomingenieur für Hochfrequenztechnik, und Helga Seelisch, Lehrerin für Deutsch und Russisch

Familienstand: geschieden
Staatsangehörigkeit: deutsch
Adresse: Platzgasse 9, 89073 Ulm, Deutschland
Telefon: privat: +49 (0)731 1517590,
beruflich: +49 (0)731 5054875
Email: Frank.Seelisch@DaimlerChrysler.com

Ausbildung

Schule

1977 - 1987 Besuch der zehnklassigen allgemeinbildenden polytechnischen Oberschule "Diesterweg" in Sömmerda

1987 - 1989 Abitur in den Spezialklassen für Mathematik und Physik der Martin-Luther-Universität in Halle/Saale

Wehr- und Zivildienst

01.11.1989 - 26.04.1990 Wehrdienst in der Nationalen Volksarmee der DDR
02.05.1990 - 26.08.1990 Zivildienst in der Wasserwirtschaft Sömmerda

Studium

WS 1990/91 - WS 1997/98 Diplomstudiengang Mathematik an der Technischen Universität Berlin
Schwerpunkte: Funktionentheorie, Algebra und Topologie
Nebenfach: Informatik (Schwerpunkte im Hauptstudium: Künstliche Intelligenz und Theoretische Informatik)
Diplomarbeit: "Koenigs' Funktion und lokal gleichmäßig beschränkte Multivalenz" (Note 1,3)
Prädikate: Vordiplom und Diplom "sehr gut"

WS 1992/93 Vordiplom

WS 1994/95 - SS 1995

Auslandsstudium an der University of Durham in Northumbria (Nordengland) über das Erasmusprogramm für Europa

Lehre

Tutor

SS 1993 - WS 1997/98

Tätigkeit als Tutor am Fachbereich Mathematik der TU Berlin mit zweisemestriger Unterbrechung aufgrund des Auslandsstudiums

Fächer: Höhere Mathematik 1 und 2 für Ingenieure, Analysis 1 und 2 für Mathematiker

Lehrbeauftragter

SS 1998

Ausführung zweier Lehraufträge an der TU Berlin (Wahrscheinlichkeitsrechnung für Lehrer und Höhere Mathematik 2 für Ingenieure)

Praktika

15.09.1989 - 31.10.1989

Bearbeiter für Systemtechnik im Softwarezentrum des Büromaschinenwerks Sömmerda, Erstellung von Schulungssoftware in Pascal

22.01.1996 - 31.05.1996

Wissenschaftliche Hilfskraft am Max-Planck-Institut für extraterrestrische Physik - Außenstelle Berlin, Programmierung in IDL zur Archivierung großer Datenmengen aus dem Weltall

05/1998 - 12/1998

Praktikant in einem Softwareunternehmen, Programmierer in Visual Basic

Berufserfahrung

01.01.1999 - 31.07.1999

Projektbetreuer für Programmieraufgaben in Visual Basic 5.0 und 6.0 bei der Datenhaus GmbH in Berlin

Schwerpunkte: Client - Server - Kommunikation, Browser, Anwendungen und Masken für Access-Datenbanken unter Visual Basic, schnelle Graphikroutinen

01.09.1999 - 31.05.2003

Doktorand in einer Berliner Forschungsabteilung der DaimlerChrysler AG

	Thema:	Constraint-Verarbeitung mittels relationaler Aggregation für modellbasierte Engineering-Anwendungen, wie z.B. Diagnose; Entwicklung des Software-Moduls "Relational Constraint Solver" in Java
12/2001 - 05/2003		DaimlerChrysler-seitiger Koordinator einer Forschungsk Kooperation mit dem Institut "Datenbanken und Artificial Intelligence" der TU Wien zur Entwicklung und Evaluierung alternativer Lösungsstrategien für den "Relational Constraint Solver"
seit 01.06.2003		wissenschaftlicher Mitarbeiter in der DaimlerChrysler AG und Mitglied der DaimlerChrysler-internen Austauschgruppe

Kenntnisse und Fähigkeiten

Sprachen

Deutsch	Muttersprache
Englisch	fließend in Wort und Schrift
Französisch	fortgeschrittene Grundkenntnisse
Russisch	Grundkenntnisse

Didaktik

sehr gute didaktische Fähigkeiten aufgrund der Tätigkeiten als Tutor und Lehrbeauftragter, gute Präsentation komplexer Zusammenhänge

EDV

Programmiersprachen	sehr fortgeschrittene Kenntnisse in Visual Basic, Pascal, Modula, Lisp, Smalltalk; Expertenkenntnisse in Java
Anwendungen	MS Office, diverse relationale Datenbanken, sehr gute Kenntnisse in SQL
Algorithmierung	Datenbank-Handling via Programmierumgebungen, schnelle Routinen für Graphik und numerische Probleme, Constraint-Verarbeitung

Wissenschaftliche Arbeit

Konferenzen

Teilnahme an der internationalen Konferenz "Principles and Practice of Constraint Programming (CP)" in den Jahren 2000-2002; 2001 und 2002 aktiver Teilnehmer des Doktorandenprogramms der CP mit spezieller Förderung

Veröffentlichungen

jeweils im Rahmen des Doktorandenprogramms sowie
2002 mit zwei Kollegen der DaimlerChrysler AG in
den Proceedings der CP; kommende Veröffentlichung
im Europäischen Journal "Artificial Intelligence Com-
munications"