Techniques for Satisfying Assembly Constraints

A dissertation submitted to the SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of Doctor of Technical Sciences

presented by BERNHARD JAKOB SEYBOLD
Dipl. Informatik-Ing. ETH
born 20.12.1969
citizen of Germany

accepted on the recommendation of Prof. Dr. Klaus Simon, examiner
Prof. Dr. Dr. Jürgen Richter-Gebert, co-examiner

1999
für meinen geliebten Opa
der mich als erster in die faszinierende Welt
der Mathematik und Informatik mitnahm
Seite Leer / Blank leaf
Preface

This thesis originated when I was a research assistant in the computer science department at the ETH Zurich. During that time, I was involved in the ROMA and TORINO projects, partly supported by the Swiss Federal Office for Professional Education and Technology (KTI2726.1 and KTI3590.1). Three organizations were involved in those projects: the Swiss CAD/CAM company Precisionsoft, the Institute of Manufacturing Technology and Machine Tools (IWF) and the Institute for Theoretical Computer Science (IfTI) where I have been working during the last four years.

In this thesis, I focus on the parts of the projects on which I have been working. This involves everything concerned with the design of the ROMA-solver and the global solving routines. The geometric solving methods shall not be discussed in detail here. Nevertheless, I provide necessary background information.

Although a doctoral thesis is a personal thing, it cannot originate without the aid of other people. In the following, I want to take the opportunity to thank them.

First of all, I am very grateful to my supervisor Prof. Klaus Simon who gave me the opportunity to work on this highly interesting project. Likewise, I am indebted to my co-examiner Prof. Jürgen Richter-Gebert who supported me a lot during the final phase of the project. In addition, I would like to thank the Swiss Federal Office for Professional Education and Technology (CTI) for their financial support.

The success of the ROMA-project, in which I had the honor of participating, is mainly caused by its outstanding team members. Jens Bathelt, Fabian Collenberg, Felix Metzger, and Gül Ogan made work in the project pleasant, motivating, thrilling, and fruitful at the same time.

Doing research in the fields of constraints satisfied me a lot, not only because of the interesting topic, but also because of the sympathetic people, whom I had the pleasure of meeting at various conferences. In particular, I would like to mention Harald Meyer auf m Hofe, which whom I had numerous interesting discussions on CSPs, artificial intelligence, and everything. A big cheers goes to Jeremy Frank, who carefully examined a draft of this thesis. I also would like to thank Meera Sitharam, with whom I had interesting email exchanges.
A necessary condition for successful research is that you must be encircled by fascinating people. I was very lucky that this was indeed the case in the computer science department of the ETH Zurich. Especially, my fellow PhD students in the group of graph algorithms—Thomas Bickel, Nicola Galli, Thomas Raschle, and Paul Trunz—always inspired me in one way or another. Of the others, I would particularly like to thank Jochen Giesen, Johannes Blömer, and AleXX Below for their helpful comments on parts of this thesis.

It is impossible to mention all the other people who influenced me during my time as a research assistant. Therefore, I want to thank them in globo. I am happy that each of them was there for me. What would life be without friends?

Zurich, June 1999

Bernhard Jakob Seybold
Seite Leer /
Blank leaf
Abstract

The assembly problem consists in arranging rigid bodies in 3D-space such that a given set of geometric constraints is satisfied. Solutions to this problem are used to understand the kinematic behavior of mechanisms. The general assembly problem is NP-hard.

A commercial assembly solver for CAD/CAM systems must meet several requirements: it must support modeling and solving practically relevant assemblies efficiently, it must be able to localize mobility, redundancy, and inconsistency, and it also needs to have an extendible software architecture.

In this thesis, we introduce techniques used by the ROMA-solver for solving assembly problems. In the first part, we concentrate on the design of the solver. The ROMA-solver always runs in polynomial time. Hence, it cannot cover the entire problem class. Nevertheless, the ROMA-solver successfully solves most practical assemblies. The solving process resembles constructing. Thus, step-wise constructible assemblies are solvable and unsolvable cases are often badly constructed. Furthermore, the ROMA-solver can easily be extended to cover additional cases.

In the second part, we formalize the solving strategy using constraint satisfaction theory. Usually, a constraint satisfaction problem (CSP) is assumed to have finite domains. However, in the assembly they are uncountable. We extend the standard algorithms and show how to use the (possibly imprecise) results to gain insight into the kinematic behavior of the mechanisms.

In the third part, we present two novel strategies for consistency enforcement. Both strategies are applicable to general CSPs. They enable the ROMA-solver to run more efficiently and to compute more structural information. First, we show that consistency enforcement can be restricted to biconnected pairs of variables. Second, we present a new approach to deal with immobile pairs of bodies. We provide an elegant and efficient implementation based on the Union/Find data structure.

The ROMA-solver was evaluated on practical examples. Because of its design and the strategies mentioned above, the ROMA-solver handles a larger problem class efficiently and provides more information than previous solvers.
Zusammenfassung

Das Zusammenbauproblem besteht darin, starre Körper im dreidimensionalen Raum so anzuordnen, dass eine gegebene Menge geometrischer Bedingungen erfüllt ist. Lösungen dieses Problems werden benutzt, um die kinematische Wirkungsweise mechanischer Apparate zu verstehen. Das allgemeine Zusammenbauproblem ist NP-schwer.

An kommerzielle Zusammenbau-Solver für CAD/CAM-Systeme werden verschiedene Anforderungen gestellt: sie müssen effizientes Modellieren und Lösen praxisnaher Zusammenbauten erlauben, sie müssen Beweglichkeit, Redundanz und Inkonsistenz lokalisieren können, und sie müssen eine erweiterbare Software-Architektur besitzen.


Im zweiten Teil formalisieren wir die Auswertungsstrategie mit Hilfe der Theorie der Constraint Satisfaction. Normalerweise hat ein Constraint Satisfaction Problem (CSP) endliche Wertebereiche. Im Zusammenbau sind sie überabzählbar. Deswegen erweitern wir die Algorithmen und zeigen, wie deren (teilweise ungenauen) Resultate benutzt werden können, um Einblicke in die Kinematik der Apparate zu erhalten.


Seite Leer / Blank leaf
# Contents

1 Project Overview

1.1 Outline .............................................. 4

2 Introduction

2.1 Geometric Constraints ............................... 7
  2.1.1 The Assembly .................................... 8
  2.1.2 Related Areas .................................... 10

2.2 Requirements to Assembly Solvers .................. 13
  2.2.1 Assembly Constraints ............................ 14
  2.2.2 Information ...................................... 15
  2.2.3 Software Aspects ............................... 16

2.3 Summary ............................................ 18

3 Assembly Modeling

3.1 Constraint Types ................................. 19
  3.1.1 Base Constraints .............................. 22
  3.1.2 Compound Constraints ......................... 24
  3.1.3 Kinematic Constraints ........................ 24
  3.1.4 Driving Input Constraints ..................... 25

3.2 Constraint Graph .................................. 27

3.3 Degrees of Freedom ............................... 28

3.4 Solving Approaches ................................ 31
  3.4.1 Algebraic Methods ............................. 31
## Contents

6.4 Extending the ROMA-Solver .................................. 101
6.5 Summary ...................................................... 104

7 Rigid Constraints .............................................. 105
7.1 Introduction .................................................. 105
7.2 Related Work .................................................. 108
7.3 Definitions and Notation .................................... 109
7.4 Functional Constraints ..................................... 110
   7.4.1 Pivot Consistency ..................................... 111
   7.4.2 Detecting Redundancy .................................. 112
7.5 Isolating Block-Consistency ................................ 114
7.6 Reformulating Geometric Constraints ....................... 117
   7.6.1 Roma-Isolate .......................................... 118
   7.6.2 Algorithms ............................................ 120
   7.6.3 Data Structure and Algorithms ....................... 122
7.7 The Ultimate ROMA-Solver .................................. 129
7.8 Summary ...................................................... 129

8 Discussion ....................................................... 133
8.1 Examples ..................................................... 133
   8.1.1 Elevator Door .......................................... 134
   8.1.2 STARRAG Milling Machine ............................. 138
   8.1.3 Saber Saw ............................................. 143
8.2 Conclusions .................................................. 151
8.3 Open Problems ............................................... 153
8.4 Lessons Learnt ............................................... 155

Bibliography ....................................................... 157

List of Figures .................................................. 165

List of Algorithms ............................................... 169

Index .............................................................. 171
Currently, mechanisms are constructed with the help of interactive CAD/CAM systems. An important step in the design of a mechanism is assembly. It consists in arranging the rigid parts which have been designed in earlier phases. The arrangement can elegantly be described using constraints.

Assembly constraints describe relative mobility between parts implied by physical contact. For instance: a spindle in a hole, two surfaces touching each other, etc. Typically, assembly situations are described by kinematic joints. These joints refer to only two parts. Thus, they correspond to binary constraints. Hence, an assembly can be modeled as a constraint graph, see Figure 1.1. The nodes correspond to the bodies and there is an edge between two nodes when the relative mobility of the corresponding parts is constrained.

To compute global information out of locally defined constraints an assembly solver is needed. There are many requirements to modern assembly solvers. They should allow practically relevant 3D-assemblies to be modeled easily. Assemblies that can be modeled should also be solvable. The solving must be fast to enable interactive design. An assembly solver should allow the bodies to be entered and deleted in an arbitrary order (variational design). In addition to a solution, an assembly solver should also report the rigidity structure, the remaining mobility, inconsistency, redundancy, degeneracy, and numerical problems of the mechanism. Finally, the software should be open, such that new constraint types may be introduced.
The assembly problem is NP-hard. Hence, we do not expect to build a solver that solves all assemblies efficiently. However, constructions are often clearly designed and contain only a limited number of typical building blocks. Most of them can be solved by a special routine. Now an assembly solver could be built which integrated these routines in a clever way. Such a solver would always be fast on a limited class of assemblies depending on the number of special routines. The drawback is that the number of special routines can soon become large. Thus, it is crucial to design and integrate the routines carefully.

To build a feasible assembly solver covering a large class of problems was the challenge facing us at the start of the ROMA-project. The ROMA-solver deals with practical problems occurring in CAD/CAM systems. It is based on new concepts and techniques which yield the following features:

- The ROMA-solver allows assemblies to be entered directly as constraint graphs. Because this corresponds to the language of the engineer, the modeling is efficient and intuitive.
- The results of the ROMA-solver are represented also as constraint graphs. Hence, the solution has a clear semantic meaning to the engineer.
- The ROMA-solver supports variational design because the data structure constraint graph can be changed arbitrarily without affecting the solving process.
• The ROMA-solver can deal with over- and under-constrained assemblies.

• The ROMA-solver provides a powerful set of 3D-constraints and a mechanism to combine base constraints to higher level constraints. In this way, the important kinematic joints are supported.

• The ROMA-solver allows interactive design because it is always fast. The time complexity is cubic in the number of bodies with small constants. Therefore, the user always receives a result quickly.

• The ROMA-solver cannot compute cover the whole class of assembly problems. However, it solves almost all practical assemblies. Solving resembles constructing. Therefore, constructible cases correspond to solvable cases. Likewise, non-solvable assemblies are often badly designed and should be changed.

• The functionality of the ROMA-solver is well scalable. It is possible to introduce new geometric routines without affecting the global solving strategy. In this way, the power of the ROMA-solver can be adapted to the needs of the user. Nevertheless, we need few special routines to obtain more functionality than Kramer's degrees of freedom analysis.

• The solving process of the ROMA-solver corresponds to consistency enforcement. More precisely, we enforce strong 3-consistency. In constraint satisfaction theory, the variable domains are commonly assumed to be finite. The domains in the assembly problem are uncountable. We show how to adapt the standard consistency algorithms to work partially with those domains.

• For non-solvable assemblies the ROMA-solver still provides partial information. In this case we relax the problem to a minimum spanning forest of the initial constraint graph. For the relaxed problem solutions can be found easily. Although they do not satisfy all constraints, they can nevertheless be used for two purposes. First, they are a good initial assignment for numeric iterative approaches. Second, in rigid situations the unsatisfied constraints indicate where there is still work to be done.

• The ROMA-solver is based on a new, weaker kind of consistency, block-consistency (BC), considering only biconnected sets of variables. BC can be enforced more efficiently than ordinary consis-
Chapter 1. Project Overview

tency. Nevertheless, we show that by enforcing strong 3-block-consistency all important properties of the assembly are already computed. During BC-enforcement, the block structure is preserved. Hence, redundancy and inconsistency can be localized on the block level.

- The ROMA-solver uses a new approach to deal with functional constraints. In the assembly, this approach covers rigidity. However, the scheme is more general and can be used in all constraint satisfaction problems. We show that in the presence of functional constraints, the graph can be reformulated to simplify the problem. It might even happen that a block collapses into smaller blocks, which is an advantage in combination with block-consistency.

- The reformulation because of functional constraints can elegantly be implemented by an adaption of the Union/Find data structure. We show that reformulation then can be done efficiently. Furthermore, the position and orientation of the rigid structures can be manipulated in almost constant time per operation, independent of the size of the structure.

1.1 Outline

The purpose of this thesis is to explain and justify the features of the ROMA-solver. In detail, it is organized as follows.

In Chapter 2, a short introduction to geometric constraints and their applications is given. Then we focus on the assembly problem and elaborate on the requirements to an assembly solver for practical purposes.

In Chapter 3, the constraint set supported by the ROMA-solver is introduced. Afterwards, we compare certain solving approaches for assembly problems with this particular list of constraints.

In Chapter 4, the design of the ROMA-solver is presented. First, we show how mobility is represented by constraints. Then we shortly describe the geometric library used. Finally, we discuss the limitations of our approach and argue that, nevertheless, most relevant cases can be handled.

In Chapter 5, we formalize the solving process with the help of constraint satisfaction techniques. We show how these techniques are
adapted in order to work in conjunction with assembly constraints. In addition, we present algorithms to explore the resulting data structure.

In Chapter 6, we introduce block-consistency, a weaker kind of consistency. We show that in assemblies it is sufficient to enforce only 3-block-consistency instead of ordinary 3-consistency.

In Chapter 7, we present a new approach to deal with functional constraints. We show that this approach leads to automatic problem simplifications. Furthermore, we provide an elegant and efficient implementation based on the Union/Find data structure.

Finally, in Chapter 8, the merits of the ROMA-solver are demonstrated in three real-world examples. Furthermore, conclusions, open problems, and future research directions are offered.
Chapter 2

Introduction

In this thesis, we introduce the ROMA-solver, a new efficient solver for geometric constraints. The ROMA-solver is specifically designed for use in an assembly module of a commercial CAD/CAM system. One of its main merits is that it meets the requirements to practical assembly solvers to a larger extent than previous systems.

In this chapter, we introduce the assembly and formulate requirements to assembly solvers. In Section 2.1, we begin by introducing the domain of geometric constraints in general. After a short overview on the assembly problem related applications of geometric constraints are reviewed briefly. Then we elaborate on the requirements to modern assembly solvers, Section 2.2. This is done on three levels. First, we discuss issues concerning the supported constraints. Second, we analyze what kind of information must be computed. Third, we examine conceptual properties needed for an assembly solver in a commercial CAD/CAM system.

2.1 Geometric Constraints

Constraints naturally occur in many parts of our world. We all must restrict ourselves in order to fulfill the constraints emerging from law, social morality, financial matters, etc. Although it is often easy to state the constraints to be fulfilled, we all know how difficult it is to satisfy them simultaneously.
Geometric constraints are inherent in most of our environment even if we do not recognize them in the first place. A big part of the world works only because parts are in a certain position and orientation with respect to each other. Mechanic watches work because numerous parts are assembled exactly. In air traffic certain minimal distances between airplanes are required. An IKEA cupboard can only be built if the parts exactly fit to each other. Geometric constraints describe such relations. However, there are various kinds of geometric constraints depending on the domain they are used in.

A general set-up to model situations with geometric constraints is the geometric constraint satisfaction problem (GCSP). It is the spatial joining of objects such that their relative positions and orientations satisfy a given set of geometric constraints. The semantics of the objects depends on the application. They can be protons and neutrons, molecules, cells, particles, parts of a mechanism, planets in space, objects on a computer screen, mathematical constructs—virtually everything that can be positioned and oriented in some space. The objects themselves are assumed to be rigid. Furthermore, we abstract from the weight, the shape, and other physical properties. The only thing known is a coordinate system attached to each object, which must be placed in space.

Geometric constraints represent restrictions imposed on the relative positions and orientations of the objects with respect to each other. These restrictions can be described in various ways. Depending on the application the constraints either define one exact relative position and orientation directly. They indicate partial restrictions like "two points must coincide" or a qualitative relation, such as "a part touches another part".

In the following, we mention some typical applications of geometric constraints and discuss the differences among them.

2.1.1 The Assembly

Throughout this thesis, we are especially interested in the assembly as part of a CAD/CAM system for mechanical engineering. The assembly is one of the later steps in the development of a mechanism. The engineer has designed all parts (bodies) of which the mechanism consists. During the assembly step the parts are put together to form a whole. Most of the basic concepts are already determined but details still have
2.1. Geometric Constraints

Figure 2.1: A plug.

to be worked out. Steps following the assembly can be collision checks, simulation, production, etc.

As an example consider the plug in Figure 2.1. It consists of 2 housing parts (1 and 2) which are rigidly connected by a screw (3). 3 pins (4, 5, and 6), fit into the outlet. To each of them a screw is attached to fix the cable (7, 8, and 9). Furthermore, there are 4 parts (10, 11, 12, and 13) embedded at the other end of the lower shell, which fix the cable in the plug. How the plug is actually modeled by geometric constraints is shown in Section 3.2.

The constraints of an assembly are typically not arbitrary descriptions of the relative positions and orientations, but reflect restrictions implied by physical contact. For instance, in the plug example it is required that pin 4 is pushed into the housing 2. Other examples are a spindle put into a hole, two sticks connected by a screw, a part glued on a surface, etc.

The geometric constraints used in the assembly are mostly binary. That is they impose restrictions on only two parts. Commonly, they are expressed by describing coincidence of some geometric entities attached to either of the two parts. For instance, a pt-coi-pt constraint requires that a given point of the first part and a given point of the second part coincide. This constraint leads to a partial restriction of the two parts. Their relative remaining mobility is a rotation centered at the two coinciding points. The pl-coi-pl constraint requires that a given plane attached on the first part coincides with a given plane attached
on the second part. In this case, the remaining mobility is a rotation orthogonal to the plane in conjunction with a translation in the plane.

A mechanical construction typically contains up to 100 parts and about 300 constraints. Larger assemblies are split into sub-assemblies to define interfaces and to reduce complexity. Often, large parts of the mechanism are rigid and only some groups of parts are movable with respect to each other.

2.1.2 Related Areas

Beside the assembly there are other applications involving geometric constraints. Some of them are similar to the assembly, some of them differ significantly. In the following, we mention a few other applications and explain the similarities and differences compared to the assembly.

An important area in mechanical engineering is kinematics. Kinematics is the science of treating motion regardless of causing forces. Almost all of the mechanism modeled in a CAD/CAM system for mechanical engineering involve some kind of kinematics. For instance, even the almost rigid plug is modeled by kinematic constraints, so-called joints. Therefore, when we describe the assembly we will always include kinematic problems as well.

Robots or working machines are typical mechanisms built of kinematic joints. A robot arm consists of several parts connected in series, see [Cra89] for a comprehensive introduction to robotics. The actuator at the end of the arm is moved by manipulating the the arm parts. Two problems occur in this context. First, given the relative position and orientations of the arm parts, compute the position and orientation of the actuator. This task is called forward kinematics. Second, given a goal position and orientation of the actuator, how to position the arm parts. Such problems are referred to as inverse kinematics. Forward kinematic problems are typically easy to solve, whereas inverse kinematic problems contain loops and are hard.

Another application of geometric constraints in the CAD/CAM-world is sketching. Sketching is the process of designing the atomic parts of a mechanism, which are put together in the assembly step. Commonly, the parts are built by drawing a 2D shape, see Figure 2.2. This shape is extended to 3D by rotation around an axis or by extrusion along a vector.
2.1. Geometric Constraints

Figure 2.2: The sketch of an Oldham-coupling, [Ehr95].

Figure 2.3: Constructing the middle between two points.

The geometric constraints are used to model the dimensions of the drawing. In this way, the shape is parameterized and changes can be performed easily. The parts are mostly points, axes, or vectors. The geometric constraints involve coincidences, angles, distances, orthogonality, and parallelity.

Geometric constructions are similar to sketching. Pupils do them in school to learn about geometry. For instance, a typical exercise is to construct the middle of two given points. This is done by the following steps, see Figure 2.3.

1. Given are two points \((A, B)\).
2. Draw a line through \(A\) and \(B\) \((g)\).
3. Draw two circles with identical radii around \(A\) and \(B\) such that they intersect in two points \((C, D)\).
4. Draw a line through \(C\) and \(D\) \((h)\).
5. The intersection of the \(g\) and \(h\) gives us the middle between the two initial points \((M)\).
As can be seen in the example geometric constructions are typically history-based. That is there is a clear sequence in which the objects (here points, lines, and circles) are derived by earlier objects. Changes in the construction are performed by going back, changing a particular step and recomputing all entities influenced later. For instance, when we change the location of \( B \), we need to redraw the circle around it. This change affects \( C \) and \( D \). Thus, also \( h \) and finally \( M \) need to be recomputed. Constructions of this kind are supported for instance by Cinderella \([\text{Cin}]\) and Cabri \([\text{Cab}]\) or in 3D by 3D-Geometer \([\text{Geo}]\).

Geometric constructions directly imply a recipe how to get from the input to the output. There is no direct possibility to evaluate in the reverse direction. This is a significant difference to the assembly problem, where the model is a description of the situation without indicating an evaluation order. Furthermore, geometric constructions mostly deal with geometric objects in 2D, whereas for the assembly rigid parts in 3D are needed.

In spatial reasoning we are confronted with more informal geometric constraints. Like the assembly the objects are arbitrary rigid parts and the constraints describe their relative position and orientation. The constraints, however, are less explicit. For instance, they demand that a certain body is "on" another body, that two bodies "intersect", or that a body is "included" in another one, etc.

These qualitative constraints are classified. Allen introduced a temporal logic framework in \([\text{All83}]\). With this algebra qualitative information on time intervals can be expressed, such as "task A starts before B", "A overlaps B" etc. Güsgen extended the algebra to spatial reasoning \([\text{GH93}]\). The idea is to introduce linguistic variables for qualitative spatial information.

The assembly is more constructive than spatial reasoning. There are more hints how to realize the geometric constraints. Although the concrete realization is not given (contrary to geometric constructions), a solver can directly derive actions to satisfy the constraints. For instance, a pt-coi-pt constraint requires that two points coincide. It can be satisfied by translating one body along the vector between the points. It is much more difficult to derive such an invariant from an "a part touches another part" constraint, in case the bodies have complicated shapes.

Like spatial reasoning the layout problem also involves an informal description how to place certain objects in space such that geometric
constraints are satisfied. But in addition, it is required to find a solution that optimizes a certain cost function. The geometric constraints only serve to define the allowed set of possible situations. As a describing language the framework of G"usgen [GH93] could be used. For example, the *map labeling* problem is to place descriptions of places onto a map such that no two descriptions overlap. In this solution space a solution is looked for where the labels are closest to the points which they describe.

A famous problem is to compute the smallest sphere into which $n$ spheres of equal size can be packed. The set of possible arrangements can be specified by introducing a constraint between each pair of spheres requiring that the spheres do not intersect. In the implied solution space the densest arrangement is searched for.

Another layout problem is to arrange electronic parts for VLSI design. Given is a graph where the nodes represent certain electronic objects and the edges are connections between the objects. The goal is to find a solution using minimal space in which the objects do not overlap.

A similar problem occurs also in the context of CAD/CAM. For instance, the positioning of cables in an airplane. There are many different wires for electronic systems and steering, pipelines for water and air, etc. The elements are restricted by geometric constraints, mostly involving minimal distances and non-intersection. The task is to find an arrangement such that the risk and costs are minimized.

Alike in spatial reasoning a layout problem is more informally described than an assembly problem, with the additional difference that optimizing aspects are not treated by the assembly.

### 2.2 Requirements to Assembly Solvers

In the previous sections, we introduced some variants of geometric constraints. Now, we return to the assembly problem and discuss approaches to solve it. In the following, we collect the requirements for an assembly solver such that it can be used in a commercial CAD/CAM system. In particular, we consider the constraint types supported, the information computed as well as aspects concerning usability and software engineering.
2.2.1 Assembly Constraints

In this section, we discuss general points that must be considered when choosing the constraints supported of an assembly solver. The concrete constraint list provided by the ROMA-solver is listed in Section 3.1.

When choosing the constraints to be supported by an assembly solver, one is confronted with a trade-off between expressiveness and feasibility. If for instance only constraints were considered involving translations but no rotations, the problem becomes linear and independent in each dimension. For that problem a solution can be found by using Gaussian elimination in polynomial time. Solutions involving optimality criteria are computed by linear programming. Linear constraints (equalities and inequalities) are expressive enough for window layouts as demonstrated by the QUOCA system [BMSX97]. But in the assembly and especially for kinematic applications, we also need objects that can be arbitrarily rotated in 3D.

On the other hand, we could allow the constraints to be arbitrary equations involving the coordinate systems of the parts. Then the assembly corresponds to an algebraic system of equalities and inequalities. This set-up is so general that it is not a priori clear whether it can even be decided if there exists a solution or not. If the equations turn out to be polynomials in the variables—what is often the case—then a general result of Tarski [Tar48] on semi-algebraic sets implies that the problem is decidable and that a solution can be approximated whenever it exists. Basu, Pollack, and Roy [BPR97] state that there is an algorithms finding a solution in singly exponential time in the size of the system.

Assembly constraints do not induce arbitrary algebraic equations. Still, the underlying problem is NP-hard. It is open whether or not it is NP-complete, although Liu and Popplestone say so in [LP90]. Their argumentation, however, is false. In either case we do not expect to build a solver that is always efficient on all problems. But if the constraint set is chosen with care, a solver can nevertheless often find solutions efficiently as first demonstrated by Kramer in [Kra92a].

Another important issue when selecting the constraint language is to consider the abstraction level. Geometric constraints can be modeled as algebraic equations. However, besides the bad complexity of algebraic solvers, there are two additional drawbacks. First, the resulting output cannot be mapped onto parts of the input that cause the effect. However, Kramer demonstrated that exactly this knowledge is important
to be able to build an efficient assembly solver. Second, the language of the engineer are kinematic joints. Hence, the mechanism is initially designed in terms of kinematic joints which must then be mapped onto algebraic equations either by hand or automatically. After solving the results must be reinterpreted in terms of constraints.

2.2.2 Information

A CAD/CAM system is a visual software. Most of the time the engineer works in front of a computer screen that shows a projection of the current mechanism or a part of it. Therefore, an assembly solver must be able to compute a placement of all parts which satisfies all given geometric constraints. Most assembly solvers concentrate on this task.

However, there are more requirements. For instance, the engineer wants to know if there is remaining mobility left in the mechanism. If so, he/she also wants to know where and what kind of mobility. Is it a translation along an axis or a rotation around a point? Are there multiple discrete solutions? Of course, it is also important to know which parts are rigid with respect to each other.

Redundancy is a topic in mechanical design as well. Redundancy means that some mobility is restricted more than once. Then a chosen position and orientation must be consistent with all constraints between the bodies at the same time. Otherwise there is no solution. But this requires high accuracy. During assembly solving numerical accuracy needs to be high, and in production small tolerances must be met. Small tolerances, however, might make a product expensive. For these reasons the engineer should be aware of redundancies and avoid them as the case may be. Hence, it is essential to report redundancies in a mechanism.

Furthermore, to avoid redundancy, information about its source is needed. Which constraint(s) must be relaxed to eliminate the redundancy? Often, not a single constraint is responsible for the redundancy but a constraint group. For instance, if parts are pairwise constrained to form a redundant loop, each constraints can be relaxed to make the system redundancy-free. In more complex situations it would be a big help if the solver returned the loop in which the redundancy occurred.

In addition, there should also be information on the numerical instability of the mechanism. For instance, if the position of some point is defined to be the intersection of two axes, and the angle between the axes is small, then small inaccuracies in the values of the axes yield big
differences in the values of the point. Basically, this leads to the same accuracy problems as in the case of redundancies. Likewise, the solver should report the numerical instabilities as well, such that the engineer can avoid them.

Another issue is degeneracy. Normally, a constraint reduces the same amount of mobility always. However, in certain cases depending on the input parameters, the size of the solution set is not as expected. For instance, assume that the possible locations of a point are given as the intersection of a plane and a sphere. Normally, the solution set would be a circle. However, if the distance between the center of the circle and the plane is equal to the radius, then the intersection is only a single point. Like redundancies and numerical instability, degeneracy should be reported for a full understanding of the mechanism.

2.2.3 Software Aspects

Beside the requirements concerning the information to be computed, user-friendliness and software design of an assembly solver are important for its success.

Even an experienced engineer needs to change the design often to elaborate the details correctly. Sometimes it happens that redundancy needs to be removed, that dimensions are adapted, or that the overall design becomes outdated. The assembly should be modeled in a way that can be edited easily. Hence, the data structure must reflect the objects and terms in which an engineer thinks. For instance, algebraic equation systems need to be encapsulated by a higher-level interface.

Contrary to solver for geometric constructions an assembly solver must support variational design, that is there is no distinction between input or output. Nevertheless, several assembly solvers are history-based to overcome the difficulty of geometric constraints. This makes changes very tedious. If possible, history-based modeling should be avoided. Furthermore, the solver must be able to handle situations that are rigid, over-constrained or under-constrained. In general, all situations that are describable by constraints should also be tractable.

During assembly the engineer often wants to query information about the current state of the mechanism. Whenever a new constraint is added to or deleted from the design, the screen must be updated and the information on rigid structures and redundancy must be recomputed. This should be done without disturbing the design process significantly.
Solving geometric constraints symbolically is difficult. Even situations with only two parts might require special solving routines. A difficult problem is for instance the Steward platform. A Steward platform is connected to the ground by six (hydraulic) struts, see Figure 2.4. It can be moved by changing the contraction (length) of the struts. Now, given the points and the lengths of the struts, compute symbolically the position and orientation of the platform. The solution to this problem is not known. Even if the ground points are coplanar, the resulting polynomial (which is also not known) is of degree 20 and has up to 40 real solutions [WH98].

For such cases to be handled an assembly solver should be open for enhancements. Then a Steward platform can be handled by a new steward constraint, which triggers the special solving routines. However, in order to use steward in combination with other constraints, a clear abstraction between special solving routines and global solver is required.

However, in order to keep the system tractable and supportable, the number of special solving routines should not be too numerous. Fortunately, it is good style to make constructions easy and clear such that a large number of construction is expected not to be too complex. However, even if solving cannot be performed properly on the whole situation, the solver should be able to report information on the easy parts. The solver must be robust with respect to the input situation. That is, reported information should be true.
2.3 Summary

In this chapter we have introduced the domain of geometric constraints. There are various applications which involve geometric constraints. One of them is the assembly in the context of a CAD/CAM system. In general, geometric constraints can be hard to solve. Nevertheless, an assembly solver for a practical CAD/CAM system must be efficient and must be able to compute various things, such as an assignment, rigidity structure, redundancy, degeneracy, and numerical instability. Thus, we cannot expect that an assembly solver always is both efficient and informative. Fortunately, constructions are often simple and clearly stated and involve only a limited number of difficult situations. These can be handled by special solving routines, provided that the solver is open and extendable.
Chapter 3

Assembly Modeling

Geometric constraints are a natural language to describe spatial situations. Solving the situation, however, is a difficult task. There are many solving approaches which deal with geometric constraints. Assembly constraints, which are a special class of geometric constraints, correspond to algebraic equations, thus any algebraic solver can be used. However, they do not meet the requirements to become practical assembly module.

The ROMA-solver is based on KRAMER’s degree of freedom analysis combined with constraint satisfaction theory. Our method emulates a human assembly solver and uses properties typical to constructions.

This chapter is organized as follows. In Section 3.1, we list the geometric constraints provided by the ROMA-solver. Then, in Section 3.4, we briefly review some solving approaches to deal with those constraints and compare how the approaches meet the requirements to a practical solver as described in the previous chapter.

3.1 Constraint Types

As mentioned in Section 2.2.1 the selection of the supported constraints has a strong effect on the feasibility of the assembly. There, we already mentioned two extreme sets: linear equations and arbitrary algebraic equations. Linear equations are too less expressive, whereas arbitrary equations cannot be solved efficiently.
More expressive than linear equations are ruler-compass constructions. They are 2D-constructions which can be drawn with a ruler and a compass only. In this set-up a large classes of engineering drawings can be described. Hence, it is suitable for sketchers. A method that guarantees termination and completeness for ruler-compass constructions is presented by Brüderlin in [Brü87, Brü93]. It uses the Knuth-Bendix critical pair algorithm and is implemented in Prolog.

For the assembly we need to support rigid parts which live in 3D-space. A constraint set of practical importance comes from kinematics. There, the terms joint is used for constraints and link for parts. The joints can be classified into lower pairs and higher pairs [Reu76]. A lower pair is a joint describable by two surfaces sliding over each other. There are only 6 possible lower pairs, as shown in Figure 3.1.

The Revolute joint (R) defines a rotation around an axis, the Prismatic joint (P) a translation along a line, the Screw joint (H) a combined rotation and translation, the Cylindrical joint (C) a trans-
3.1. Constraint Types

A translation and rotation with respect to the same line, the Planar joint \((F)\) a translation on a plane, and the Spherical joint \((S)\) a full rotation around a point. The higher pairs involve a point or line contact to a plane. One important higher pair is the Universal joint \((U)\). With the kinematic lower pairs and the Universal joint, most assemblies can already be modeled.

KRAMER introduces a small yet powerful set of geometric constraints in [Kra92b]. With them the kinematic lower pairs and the Universal joint can be modeled.

The constraints of KRAMER all are defined using a single universal geometric object called marker. A marker consists of a point and two orthogonal axes, the \(x\) and \(z\) axis, see Figure 3.2. For instance, the pt-coi-pt constraint is defined between two markers requiring that their points must coincide. We chose the approach of modeling the constraint with the smallest geometric entities which they affect to avoid unnecessary ambiguity. Of course, both approaches are equivalent and the constraints can easily be transformed between them. For consistency we adapt the names of the constraints. For instance, parallel-z in KRAMER’s notation corresponds to vc-coi-vc in ours.

The constraint names used here are schematically built and similar to the ones used in [RF94]. They consist of the type of the two geometric elements involved and the relation between them. For instance, pt-coi-pt means that a point (pt) must coincide (coi) another point (pt). A pt-on-pl indicates that a point (pt) must lie in a plane (pl). The geometric elements currently supported by the ROMA-solver are listed in Table 3.3.

Geometric elements can be combined using the following relations:

- **coi** (coincidence): the two geometric elements must be equal. Coincidence can only be applied between elements of the same kind.
### Element Abbr. Description

<table>
<thead>
<tr>
<th>Element</th>
<th>Abbr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>pt</td>
<td>a point</td>
</tr>
<tr>
<td>vector</td>
<td>vc</td>
<td>a direction (the length has no semantic)</td>
</tr>
<tr>
<td>line</td>
<td>ln</td>
<td>an oriented line</td>
</tr>
<tr>
<td>plane</td>
<td>pl</td>
<td>an oriented plane</td>
</tr>
<tr>
<td>ray</td>
<td>ry</td>
<td>a ray</td>
</tr>
<tr>
<td>sphere</td>
<td>sp</td>
<td>a sphere</td>
</tr>
<tr>
<td>trihedral</td>
<td>th</td>
<td>a trihedral defines a 3D-rotation, Figure 3.2</td>
</tr>
<tr>
<td>frame</td>
<td>fr</td>
<td>a 3D-frame (coordinate system), Figure 3.2</td>
</tr>
</tbody>
</table>

**Figure 3.3:** *Geometric elements in 3D space.*

Note that some elements own an orientation (e.g. line, plane).
Example: pl-coi-pl enforces that a particular plane must be equal to another plane. This condition holds if the planes define the same point set and have the same orientation.

- **on:** the point set described by the first element must entirely be included in the point set of the second element. This relation does not consider the orientation of the elements.
  
  Example: pt-on-pl means that a point must lie in a plane.

- **col, ang, abv, maxang, minang:** these relation (collinear, angle, above, maximum-angle, minimum-angle) are currently only used for one constraint each. Their exact meaning is explaining at the respective constraint.

#### 3.1.1 Base Constraints

The base constraints are not further cut into smaller pieces. They build the core of the constraint language. Higher level constraints can be built by combining the base constraints, see Section 3.1.2 and 3.1.3.

The base constraints are all defined between two bodies. Hence, they are *binary constraints*. In the following, we attach an index to the geometric entities to indicate the body to which they belong. For instance, a `pt-coi-pt(p₁, p₂)` means that the bodies 1 and 2 are constrained such that a point `p₁`, attached at body 1, and a point `p₂`, attached at body 2, coincide. Note that the coordinates of the geometric entities are all local to the coordinate system of the body to which they belong.
3.1. Constraint Types

\(fr\text{-}coi\text{-}fr(fr_1, fr_2)\)
The frames \(fr_1\) and \(fr_2\) must be identical. This constraint reduces all translational and rotational mobility.

\(pt\text{-}coi\text{-}pt(pt_1, pt_2)\)
The points \(pt_1\) and \(pt_2\) must be coincident. This constraint fixes the relative translation of the two parts.

\(pt\text{-}on\text{-}ln(pt_1, ln_2)\)
The point \(pt_1\) must lie on the line \(ln_2\).

\(pt\text{-}on\text{-}pl(pt_1, pl_2)\)
The point \(pt_1\) must lie on the plane \(pl_2\).

\(th\text{-}coi\text{-}th(th_1, th_2)\)
The trihedrals \(th_1\) and \(th_2\) must be identical. This constraint removes all rotational mobility between the two parts.

\(vc\text{-}coi\text{-}vc(vc_1, vc_2)\)
The vectors \(vc_1\) and \(vc_2\) define the same direction. Note that the length of the vector is not considered for the restriction.

\(pt\text{-}on\text{-}sp(pt_1, sp_2)\)
The point \(pt_1\) must lie on the sphere \(sp_2\). In principle, this models a distance constraint. The advantage is that scaling can be introduced. The distance is scaled with the part to which the sphere is attached. If the radius of the sphere is 0, \(pt\text{-}on\text{-}sp\) corresponds to a \(pt\text{-}coi\text{-}pt\).

\(vc\text{-}ang\text{-}vc(vc_1, vc_2, \alpha)\)
The angle between the vectors \(vc_1\) and \(vc_2\) must be equal to \(\alpha\), where \(0 < \alpha < \pi\). The cases \(\alpha = 0\) and \(\alpha = \pi\) correspond to a \(vc\text{-}coi\text{-}vc\). In both cases, additional mobility is reduced.

Each of the previous constraints induces at least one algebraic equation. The following constraints correspond to algebraic inequalities and are used to select between multiple solutions.

\(pt\text{-}abv\text{-}pl(pt_1, pl_2)\)
The point \(pt_1\) must be on the plane \(pl_2\) or on its positive side defined by the orientation of the plane.

\(vc\text{-}maxang\text{-}vc(vc_1, vc_2, \alpha)\)
The angle between the vectors \(vc_1\) and \(vc_2\) must be in the closed interval \([0, \alpha]\), where \(0 < \alpha < \pi\).

\(vc\text{-}minang\text{-}vc(vc_1, vc_2, \alpha)\)
The angle between the vectors \(vc_1\) and \(vc_2\) must be in the closed interval \([\alpha, \pi]\), where \(0 < \alpha < \pi\).
3.1.2 Compound Constraints

Base constraints can be combined to higher level constraints. There are many possibilities to do that. However, the modeling should be made redundancy-free. Otherwise the redundancy-information of the entire mechanism is no longer correct. Examples for compound constraint are:

\( \text{ln-coi-ln}(ln_1, ln_2) \)

The "line coincides line" constraint requires that the two lines \( ln_1 \) and \( ln_2 \) define the same point set. The orientations of the lines are not considered. The \( \text{ln-coi-ln} \) is modeled by a \( \text{pt-on-ln}(pt_1, ln_2) \) where \( pt_1 \) is an arbitrary point on the line \( ln_1 \), and a \( \text{vc-coi-vc}(vc_1, vc_2) \), where \( vc_1 \) and \( vc_2 \) are vectors defining the directions of \( pl_1 \) and \( pl_2 \), respectively.

\( \text{pl-coi-pl}(pl_1, pl_2) \)

The "plane coincides plane" constraint requires that the two planes \( pl_1 \) and \( pl_2 \) define the same point set. The orientations of the planes are not considered. The \( \text{pl-coi-pl} \) is modeled by a \( \text{pt-on-pl}(pt_1, pl_2) \) where \( pt_1 \) is an arbitrary point on the plane \( pl_1 \), and a \( \text{vc-coi-vc}(vc_1, vc_2) \), where \( vc_1 \) and \( vc_2 \) are the normal vectors of \( pl_1 \) and \( pl_2 \), respectively.

\( \text{ry-coi-ry}(ry_1, ry_2) \)

The "ray coincides ray" constraint requires that the two rays \( ry_1 \) and \( ry_2 \) define the same point set. It is modeled by a \( \text{pt-coi-pt}(pt_1, pt_2) \) where \( pt_1 \) and \( pt_2 \) are the end points of the rays \( ry_1 \) and \( ry_2 \), respectively, and a \( \text{vc-coi-vc}(vc_1, vc_2) \), where \( vc_1 \) and \( vc_2 \) are vectors defining the directions of \( ry_1 \) and \( ry_2 \), respectively.

A \( \text{fr-coi-fr} \) is actually also a compound constraint. It can be modeled by a \( \text{pt-coi-pt} \) in combination with a \( \text{th-coi-th} \). However, it seems reasonable to model it as an own constraint such that it can be handled more efficiently.

3.1.3 Kinematic Constraints

The kinematic constraints supported by the ROMA-solver consist of the lower pairs and the Universal joint. The Screw joint is the less important of the lower pairs and is not supported in the first version. Its introduction requires a new base constraint \( \text{pt-on-hl} \) "point on helical" what can principally be done. However, often a Screw can sufficiently be modeled as a Revolute.
3.1. Constraint Types

The kinematic joints supported by the ROMA-solver are modeled as follows:

**Revolute** \((ry_1, ry_2)\)
The Revolute joint \((R)\) is modeled by a \(ry\)-\(coi\)-\(ry\) compound constraint, see Section 3.1.2.

**Prismatic** \((fr_1, fr_2)\)
The Prismatic joint \((P)\) is modeled by a \(pt\)-\(on\)-\(ln\) \((pt_1, ln_2)\), where \(pt_1\) is the zero of \(fr_1\) and \(ln_2\) is the z-axis of \(fr_2\), and a \(th\)-\(coi\)-\(th\) \((th_1, th_2)\), where \(th_1\) and \(th_2\) are the trihedrals of the two frames \(fr_1\) and \(fr_2\), respectively.

**Cylindrical** \((ln_1, ln_2)\)
The Cylindrical joint \((C)\) is modeled as a \(ln\)-\(coi\)-\(ln\) compound constraint, see Section 3.1.2.

**Planar** \((pl_1, pl_2)\)
The Planar joint \((F)\) is modeled as a \(pl\)-\(coi\)-\(pl\) compound constraint, see Section 3.1.2.

**Spherical** \((pt_1, pt_2)\)
The Spherical joint \((S)\) is simply a \(pt\)-\(coi\)-\(pt\).

**Universal** \((fr_1, fr_2)\)
The Universal joint \((U)\) is modeled by \(pt\)-\(coi\)-\(pt\), a \(vc\)-\(ang\)-\(vc\), and a \(vc\)-\(maxang\)-\(vc\) to avoid duality.

**Identical** \((fr_1, fr_2)\)
The Identical joint \((I)\) describes a rigid joint and corresponds to a \(fr\)-\(coi\)-\(fr\).

### 3.1.4 Driving Input Constraints

The model of a mechanism should reflect the actual mobility in reality. However, some mobility is often negligible, such as the rotational mobility of a screw. In the plug example there is no important effect when the screws are rotated. As soon as the screw is modeled to be as far in the hole as needed, the shell is rigid. The remaining mobility, however, may hinder the solver, because it represents and unnecessary freedom of choice.

In addition, we often have a mobility left which represents a function of the mechanism. In this case we want to be able to simulate the mechanism by driving through all of its possible states. However, such driving can only be done if we have a handle with which we can select...
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr-coi-fr(fr₁, fr₂)</td>
<td>6</td>
</tr>
<tr>
<td>pt-coi-pt(pt₁, pt₂)</td>
<td>3</td>
</tr>
<tr>
<td>pt-on-ln(pt₁, ln₂)</td>
<td>2</td>
</tr>
<tr>
<td>pt-on-pl(pt₁, pl₂)</td>
<td>1</td>
</tr>
<tr>
<td>th-coi-th(th₁, th₂)</td>
<td>3</td>
</tr>
<tr>
<td>vc-coi-vc(vc₁, vc₂)</td>
<td>2</td>
</tr>
<tr>
<td>vc-ang-vc(vc₁, vc₂, α)</td>
<td>1</td>
</tr>
<tr>
<td>pt-on-sp(pt₁, sp₂)</td>
<td>1</td>
</tr>
<tr>
<td>pt-abv-pl(pt₁, pl₂)</td>
<td>0</td>
</tr>
<tr>
<td>vc-maxang-vc(vc₁, vc₂, α)</td>
<td>0</td>
</tr>
<tr>
<td>vc-minang-vc(vc₁, vc₂, α)</td>
<td>0</td>
</tr>
<tr>
<td>Revolute(ry₁, ry₂)</td>
<td>5</td>
</tr>
<tr>
<td>Prismatic(fr₁, fr₂)</td>
<td>5</td>
</tr>
<tr>
<td>Cylindrical(ln₁, ln₂)</td>
<td>4</td>
</tr>
<tr>
<td>Planar(pl₁, pl₂)</td>
<td>3</td>
</tr>
<tr>
<td>Spherical(pt₁, pt₂)</td>
<td>3</td>
</tr>
<tr>
<td>Universal(fr₁, fr₂)</td>
<td>5</td>
</tr>
<tr>
<td>Identical(fr₁, fr₂)</td>
<td>6</td>
</tr>
</tbody>
</table>

**Figure 3.4:** Constraints and their restriction.
one current position and orientation. More formally, we need a variable and a legal interval for it. Then we can perform a simulation by stepping through the legal positions and orientation of the variable, in order to get the positions and orientations of the entire mechanism.

Driving input constraints are used for driving a simulation and for restriction negligible freedom. A driving input constraint is a normal constraint, with the difference that some or the entire mobility of the constraint is parameterized. For instance, a driving Revolute (DR) is a Revolute constraint where the remaining rotational mobility is parameterized by an angle $\alpha$. When the entire rotation is allowed, the interval for $\alpha$ is $0 < \alpha < 2\pi$.

Often we are interested in computing a single snapshot of the mechanism. For this purpose we can choose an arbitrary value out of the interval of the driving parameter. However, a driving Revolute with a fixed angle corresponds to a fr-coi-fr describing a rigid relation. We shall see in Chapter 7 that this leads to simplifications. However, a driving Revolute is more than a fr-coi-fr, because on the one hand we know that there is a Revolute joint in reality, and on the other hand, we know how to enumerate all different states of the mechanism.

In the plug example we have to deal with several screws. Each of these screws can be modeled by inserting a Revolute constraint between the screw and the shell. However, for esthetic reasons it might be desirable that the screws are rotated such that their slits are parallels. Therefore, for each screw a driving Revolute is inserted with the same value for the angle.

The most common driving input constraints are driving Revolute, which parameterizes a rotation around an axis, and driving Prismatic (DP), which parameterizes a translation along a line.

### 3.2 Constraint Graph

In our set-up only binary constraints are considered. Therefore, an assembly problem can be modeled as a constraint graph. For each body there is a node in the graph. A constraint between two bodies is indicated by an edge between the corresponding nodes. To avoid parallel edges—more than one constraint between the same nodes—we combine all constraints between the same pair of nodes into a single compound constraint.
The constraint graph of the plug is shown in Figure 3.5. For each of the 13 parts we have a node in the graph. The constraints are chosen such that they reflect the situation in reality. The constraints are kinematic joints, see Section 3.1.3. Screws are modeled by a driving Revolute (DR).

### 3.3 Degrees of Freedom

In the previous section, we introduced a constraint list to support assembly problems. In Table 3.4, they are listed in conjunction with the number of degrees of freedom (DOF) which they reduce when applied solely. In this section, we are concerned with DOF of combination of constraints.

In general, the possible positions and orientations of a single body in 3D-space can be expressed in terms of the Euclidean group $SE(3)$, which is the semi-direct product $R^3 \times SO(3, R)$. $R^3$ is the translational part and $SO(3, R)$ is the special orthogonal group, representing all oriented orthonormal frames in 3D. To chose a single element of this group we need to specify 6 real numbers. Hence, a body has 6 DOF in 3D-space. They can be split into 3 translational DOF ($TDOF$), for choosing an element of $R^3$; and 3 rotational DOF ($RDOF$) to choose in $SO(3, R)$.

The simplest combination of bodies is a pair. This set-up is called 2-body-problem. Initially each body has 6 DOF. Thus, the whole system
has 12 DOF. These DOF can be split into 6 DOF of the whole system and 6 internal DOF between the parts. Intuitively that means that we can fix one part in space and only look at the relative mobility of the second parts with respect to the first. In kinematics the fixed part is said to be grounded.

In cases without redundancy the overall restriction of a combination of constraints is the sum of the restrictions of the constraints. Thus, a \( \text{pt-coi-pt} \) (3 DOF) in combination with a \( \text{vc-coi-vc} \) (2 DOF) restricts 5 DOF. This combination of constraints corresponds to the modeling of the Revolute joint.

If the constraints are redundant, we must subtract the degree of redundancy from the degree of restriction. For instance, a \( \text{pt-coi-pt} \) constraint removes 3 DOF. If we combine it with another \( \text{pt-coi-pt} \), the final situation is not rigid but has a single remaining DOF: a rotation along the axis through both points. The outcome is explained by the fact that one degree of restriction is redundant. Thus, there are only 5 real restrictions and 1 redundancy. The redundancy corresponds to the fact that the distances between the points at either body must be equal. If they are not equal, the system has no solution.

Often we need to solve problems containing redundancies: either because they are unwanted and will be removed, or because they are intended to obtain greater stiffness (as in the plug example). For these cases solving approaches based on counting DOF are only an estimate of the real restriction.

A more complex structure is a loop. A loop consists of \( n \) parts, say 1 to \( n \). There are geometric constraints between each consecutive two parts and also from part \( n \) to part 1. A loop owns 6\( n \) DOF which can be split into 6 of the whole system with respect to the environment and 6(\( n - 1 \)) internal DOF.

Consider a general system with \( n \) bodies in 3D space arbitrarily restricted by given geometric constraints. Again one body can be grounded. Thus, the whole system has 6(\( n - 1 \)) DOF which are restricted by the constraints. If the system is redundancy-free, the mobility can be determined according to the Kutzbach-Grübler criterion:

\[
f = 6(n - 1) - \sum_{i=1}^{m} f_{J_i}
\]

where \( n \) is the number of bodies, \( m \) the number of joints, and \( f_{J_i} \) the number of DOF restricted by the joint \( J_i \). The result indicates, whether
the mechanism is well-constrained \((f = 0)\), under-constrained \((f > 0)\) or over-constrained \((f < 0)\). We distinguish between consistently over-constrained if the restriction are not contradictory, and inconsistently over-constrained when no solution is possible. A system is rigid if it is well-constrained or consistently over-constrained.

Unfortunately, the classification is not always possible. A system can be over-constrained and under-constrained at the same time. Furthermore, there are systems which own solution branches which do not have the same DOF. For instance, a system consisting of six sticks arranged as a loop such that between each two consecutive sticks there is an angle of 90 degrees. This system has a single, well-constrained solution, Figure 3.6 on the left, and a topologically unconnected solution branch which is under-constrained, Figure 3.6 on the right.

Also in the case of redundancy the Kutzbach-Grübler criterion does not hold anymore. In the plug example we have 13 parts. Hence, there are \(12 \times 6 = 72\) DOF. The total restriction of the 18 constraints is

\[3 \times (6 + 4 + 4) + (6 + 4 + 3) + 5 + (6 + 6 + 4 + 4 + 4) = 84.\]

Consequently, it should be rigid. However, the two blocks with the parts 1 to 9 and the parts 10 to 13 have 1 remaining DOF in between. Hence, there are 72 DOF and 84 degrees of restrictions, of which 71 are real and 13 are redundant restrictions.

Under the same simplified circumstances as the Kutzbach-Grübler theorem holds the size of the largest rigid loop is bounded. To underline that this is only the case with simplified analysis, Kramer uses the term topologically rigid, meaning that the values of the parameters are not taken into account. Hence, no redundancy, degeneracy, etc. occurs.
Theorem 3.1 The largest topologically rigid loop in 3D-space has 6 bodies.

Proof A loop consists of $n$ parts and $n$ constraints. Its initial internal mobility is $6(n - 1)$. If the loop is supposed to be rigid this number must be equal or larger than the number of DOF restricted by the constraints. We get larger $n$ when the constraints restrict more DOF. Thus, we assume that each constraint restricts 5 DOF. Then we get

$$6(n - 1) - 5n \leq 0.$$ 

Hence, we $n = 6$ the size of the largest possible loop.

Analogous the largest topologically rigid loop in 2D consists of 3 parts and 3 constraints.

3.4 Solving Approaches

Geometric constraint solvers can be categorized into two main classes: algebraic solvers and special assembly solvers. Algebraic solvers work on the algebraic abstraction of the problem, whereas special assembly solvers take special properties of assemblies into account.

HOFFMANN introduces another distinction between generic solvers and instance solvers [Hof]. Generic solvers analyze the problem on the abstract level without considering numerical data whereas instance solvers use the given numeric values assigned to the geometric entities in order to analyze the situation.

3.4.1 Algebraic Methods

Assembly constraints correspond to algebraic equalities and inequalities. They can be treated by algebraic solvers. However, they must be translated into equations, a step which has some drawbacks.

The variables of the equation system describe the position and orientation of the coordinate systems of the parts with respect to a reference system. A coordinate system can be described by a transformation matrix or by a quaternion. Both representations are parameterized by 6 variables corresponding to the 6 initial DOF of the part. One possible parameterization is based on Euler angles. The three positional
variables $x$, $y$, and $z$ indicate the position of the zero of the coordinate system with respect to the reference system. The Euler angles $\theta$ (rotation about $z$-axis), $\phi$ (rotation about the new direction of the $y$-axis after $\theta$ rotation), and $\psi$ (rotation about the $z$-axis after $\theta$ and $\phi$ rotation) are used to describe the rotation of the system.

The transformation matrix in homogeneous coordinates is then:

$$
T = 
\begin{bmatrix}
\cos \phi \cos \theta \cos \psi & -\cos \phi \cos \theta \sin \psi & \cos \phi \sin \theta & x \\
-\sin \phi \sin \psi & -\sin \phi \cos \psi & \sin \phi \sin \theta & y \\
\sin \phi \cos \theta \cos \psi & -\sin \phi \cos \theta \sin \psi & \sin \theta \sin \psi & \cos \theta \sin \psi & z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Given $T_1$ and $T_2$, the transformation matrices of part 1 and 2, and given two points $pt_1$ and $pt_2$, a $pt$-coi-$pt$ ($pt_1$, $pt_2$) constraint implies the equation:

$$
T_1 \cdot pt_1 = T_2 \cdot pt_2
$$

stating that the global positions of point $pt_1$ and point $pt_2$ must be equal. When one part is assumed to be fixed, the condition is translated into 3 equations in the variables $x$, $y$, $z$, $\theta$, $\phi$, and $\psi$ of the other body, which correspond to the 3 DOF which a $pt$-coi-$pt$ restricts.

Each DOF restricted by a constraint corresponds to an equation. A well-constrained system of $n$ bodies in 3D space is mapped into a system with $6(n-1)$ variables and $6(n-1)$ equations.

The engineer expects that the assembly can be entered in terms of constraints and parts. Therefore, when an algebraic solver is used, the translation into equations must be done automatically. However, in this step the structure of the initial problem is lost, such that it is difficult to use information coming from the original problem. Furthermore, the results of the solver cannot easily be mapped back to the initial problems. For instance, it is difficult to find out which constraint combination caused a possible inconsistency.

Once modeled the systems of equations can be attacked by an general-purpose algebraic solver. Starting from TARSKI's decidability result [Tar48], many algorithm where invented to find solutions to equation
systems. The approaches can be split into two classes: symbolic methods and numeric methods.

Symbolic algebraic solvers treat the systems of equations on an abstract level without considering numerical data. The concrete values of the constraint parameters are only used at the end to determine a single assignment. The most common approach is by Buchberger based on Gröbner Bases [Buc65, RF94, RF96, Lak97] to solve polynomial equations in doubly exponential time. This approach is algebraically closed for complex numbers but not for reals. Computing symbolic solutions is still NP-hard if the equations are derived from geometric constraints [LP90]. Thus, we do not expect a solution to be found in feasible time. Especially if we consider that the number of equations is approximately 6 times as high as the number of bodies.

One possible way to work around the drawbacks of symbolic methods is to use numeric iterative methods to approximate a solution. Variants of this approach are the Newton-Raphson method and are instance solvers. They start with an (inconsistent) instantiation, which is refined step-wise in order to get a good approximation of a single solution.

Another numeric approach is based on interval techniques [Hyv92, Lho93, Fal94, Ben95, VMD97]. Like the Newton-Raphson method they are also instance solvers. The idea is to store intervals for each variable in which at least one solution exists. During the computation these intervals are iteratively refined by deleting sub-intervals for which it can be proven (locally) that they do not contain a solution.

The problem with both numeric methods is that in case of many solutions, one can in general not steer the solving to a desired solution. Furthermore, both approaches still need a lot of time even for simple problems. For instance, Numerica, one of the currently best interval solvers, needs 243 seconds to solve an inverse kinematic problem involving 6 bodies, [VMD97]. For use in an interactive assembly solver this time is not acceptable.

### 3.4.2 Special Assembly Solvers

As we have seen in the previous section algebraic approaches cannot be used in an efficient assembly module. In particular, if we consider that we need more information than just a single assignment. Special purpose solvers try to exploit more information of the assembly than algebraic methods. Some of them are presented in the following.
KRAMER introduced the degree of freedom analysis in [Kra92a]. Degree of freedom analysis satisfies constraints incrementally by performing appropriate actions on the bodies. Whenever a constraint is satisfied by an action, the constraint information is transformed into remaining mobility of the constrained part. This remaining mobility is considered when choosing for actions required by later constraints.

For instance, given two parts with three pt-coi-pt constraints between them. We assume that one part is grounded. The first pt-coi-pt constraint is satisfied by translating the non-grounded body such that the two points coincide. Then this part gets a stamp saying that there is no translational DOF and 3 rotational DOF around the point. The second pt-coi-pt is analyzed. An action must be chosen such that the first constraint remains satisfied. The solution is a rotation around the point such that afterwards both point pairs coincide. The new stamp for the non-grounded part is: 0 TDOF and 1 RDOF. The third pt-coi-pt is satisfied accordingly by a translation around the vector between the points. At the end the non-grounded part has no TDOF and no RDOF.

There are a lot of routines needed to determine the actions and the next stamp. Almost for each combination of constraint type and stamp, a special routine is needed. In [Kra92b], detailed Lisp-algorithms for the constraints presented in Section 3.1 are given. However, some combinations of stamp and constraint type are not treated. For instance, a vc-coi-vc constraint is only applicable if there are only 2 RDOF left.

Furthermore, the mapping onto invariants poses problems in the case of difficult constraints. For instance, a pt-on-sp implies 2 TDOF on the sphere and 3 RDOF. When the constraint set is known in advance, one can work around this problem by first considering other constraints and hoping for less remaining DOF. For instance, in combination with a th-coi-th, a pt-on-sp is easy because the rotation problem vanishes.

With the above described procedure only 2-body problems can be handled. All assembly problems are solvable, where there is a sequence, such that each body can be rigidly constrained to the parts preceding it in the sequence. This sequence need not to exist even if the entire mechanism is rigid, especially in kinematic loops. KRAMER provides a scheme to treat loops. All closed constraint paths in the constraint graph are enumerated and checked for rigidity. The largest topologically rigid loop in 3D is of size 6, see Section 3.3, such that all rigid loops can be found efficiently. In the case of rigidity special routines are used to compute the position and orientation of bodies in the loop. After that
they are replaced by a so-called *macro-geom*. The process is iterated until no progress can be made anymore.

With KRAMER's degree of freedom analysis many assembly problems can be solved efficiently. It was the first approach to achieve this goal. However, his approach suffers from the number of solving routines to cover the special cases. Loop detection is also rather incomplete. Furthermore, there is no indication how information like redundancy or numerical instability can be localized properly. In addition, the mapping of constraints to remaining DOF loses information which might be used for later solving steps, particularly in on-line algorithms, where the final set of constraints is not known in advance.

There are also other special approaches for geometric constraints. However, almost all of them are only concerned with finding a single solution for well-constrained situations. Mostly the approaches are only analyzed for 2D constraints.

The first systems aiming at interactively supporting design in 2D was the famous *Sketchpad* system by SUTHERLAND [Sut63]. The base object points, lines, and circles are connected by constraints like parallel, equal length, etc. The drawing could be interactively manipulated.

BRÜDERLIN solves 2D ruler-compass constructions by a rule-based approach [Brü87, Brü93]. He introduces rewrite-rules directly derived from axioms of Euclidean geometry. His system is complete and terminates because it is locally confluent.

In [SAK90] SUZUKI, ANDO, and KIMURA deal with dimensional geometric constraints in 2D similar to those in the Sketchpad system. They discuss how to model drawings in their framework and use geometric reasoning for solving.

In [BFH+95, Fud95] BOUMA, FUDOS, HOFFMANN, CAI, and PAIGE presented a solver that treats geometric problems involving points, lines, circles with prescribed radii, arcs, segments, and rays. The constraints include distance, angle, tangency, concentricity, perpendicularity, and parallelism. Solving is done by finding rigid sub-clusters which can be combined in a suitable way to get a global solution. The solver works for 2D and is designed for use in the sketcher of a commercial CAD system. In [Fud96] it is proven to act correctly.

OWEN analyses simpler spatial dependencies systematically [Owe96]. The results can be used to practically usable algorithms embedded in a commercial CAD-system.
Latham and Middleditch provide algorithms to detect subgraphs which are over-constrained or under-constrained [LM96]. The subgraphs are solved independently. It was shown in [HLS97] that finding such subgraphs is generally NP-complete. However, some heuristics are provided.

In [BNT98] Bliek, Neveu, and Trombettoni present an approach to deal with over- and under-constrained system based on graph decomposition. The authors extract well-constrained parts which can be attacked by interval methods, for instance with Numerica. The under and over-constrained parts are made well-constraint by (manually) adding variables and equations, respectively. The approach seems to work fine if only a single solution is needed. Qualitative information, however, cannot be extracted because a numerical solver is used.

In [Tro98] Trombettoni uses constraint propagation techniques to solve geometric constrain satisfaction problems. His approach—general-PDOF—is based on adding priorities to the constraints and removing unimportant constraints until the directed graph is acyclic. Then the system can be solved easily by propagating values along the direction of the edges. The drawback is that general-PDOF does also not deliver qualitative information.

3.5 Summary

There is a trade-off between the expressiveness of the constraint language and the solving. Therefore, it is important to define the supported constraint set first. We have seen that linear constraints, ruler-compass constraints, and kinematic lower pairs increase complexity in this order. Assembly constraints comprise kinematic joints and are known to be NP-hard. Thus, we do not expect that we can always find a solution efficiently.

Solving approaches can be classified into two classes: general purpose algebraic solvers and special assembly solvers. The algebraic approaches allow arbitrary algebraic constraints. However, solving is often slow even for small problems. In addition, translating from and into constraints, the language in which the engineer models the problems and expects the result, poses problems. On the one hand, the solver cannot make use of special structure. On the other hand, the results cannot easily be mapped back to the initial constraints.
3.5. Summary

Special purpose solvers try to work around the drawbacks of algebraic approaches by using special properties of assemblies. However, most of them only compute one solution or deal with a simpler set-up. The currently best approach was published by Kramer. His degree of freedom analysis is able to handle many kinematic problems in 3D efficiently. However, his approach is rather incomplete for loops and does not always provide enough information for practical assembly modules.
Chapter 4

The ROMA-Design

In the previous chapter, we have compared and evaluated several solving strategies. In this chapter, we present the design of the ROMA-solver. Its strategy is based on propagating new restriction in the constraint graph. The new restriction is generated by combining restriction of two constraints in series. The process is iterated as long as new restriction is still derived. This method is known as consistency enforcement. It is analyzed in more detail in the next chapter.

The solving approach is limited in several ways. The constraint representation is not complete, not everything can be computed, and even with perfect geometric routines we do not expect that the problem is always solved after having enforced the (limited) consistency.

Despite its incompleteness the ROMA-solver is successful in solving assemblies because it solves assemblies similar to a human constructor. Hence, there is a connection between constructibility and solvability. In addition, our approach is scalable. That is, with additional implementation effort, the ROMA-solver can be extended to cover more cases.

In Section 4.1, we give a sketch of the solving mechanism of the ROMA-solver. In Section 4.2 we compare constructibility and solvability of assemblies. Then, in Section 4.3, we show how we represent constraints and discuss the limitation of our approach. In Section 4.4, the constraint data structure and the applicable operations are introduced. Among those the most important are Add and Cat. Finally, in Section 4.5, the architecture of the ROMA-solver software is described.
4.1 The Solving Approach

In the following, we sketch out the basic idea of the solving process. For that purpose we first introduce the saber saw, shown in Figure 4.1. The example is discussed in more detail in Section 8.1.3.

The mechanism consist of a housing built of the bodies 1, 2, 3, 4, 5, and 6. Into that housing the mechanism is mounted. The bearing, body 7, is the beginning of the kinematic chain of the mechanism. It is driven by an engine. The resulting mobility is modeled as driving Revolute. Body 11 is the saw rod, which is designated to oscillate along an axis parallel to the axis of the engine. For each concrete rotation of the engine axis the saw rod has a fixed location. The other parts serve for stability, as counter weight, or to transform the rotational moment into an oscillating one.

In total the constraint graph of the saber saw consists of 12 bodies and 18 constraints. The mechanism has 1 degree of freedom, parameterized by the driving input constraint.

The answers to the following questions are of interest to the engineer:

- What are the locations of the bodies given a fixed value for the driving input constraint?

- Is the mechanism rigid?
4.1. The Solving Approach

- Are there redundant constraints. If yes which?
- Are there numerical problems and does degeneracy occur?

Before presenting the solving approach of the ROMA-solver, some other solving approaches are discussed briefly.

Basically, the assembly could be transformed into a system of algebraic equations, which can be solved by an algebraic solver. However, that requires to solve a system with

\[(12 - 1) \times 6 = 66\]

variables and

\[3 \cdot 6 + 26 + 1 \cdot 6 + 6 \cdot 5 + 8 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 89\]

(non-linear) equations\(^1\). The number of equations and the number of variables can be reduced by parameterizing the locations of some bodies relative to those of others. However, smart transformations already require insight into the problem and are not an easy task. In addition, the resulting system is too complex for symbolic solvers. Even a numeric solver would still need a long time to solve such a system to an adequate precision. Moreover, a numeric solver only reports a single solution and no information about remaining mobility, redundance, degeneracy, and numerical instability.

The advantage of Kramer's approach is that it works with the geometric abstraction. The engineer directly enters the constraint graph as given in Figure 4.1. No transformation—neither manual nor automatic—is needed. Solving starts by examining rigid 2-body problems. Those consist of the pairs of bodies that can be rigidly united by only looking at 2 bodies. In this phase, called action analysis, the parts 1, 2, 3, 4, 5, 6, and 7 are united to a macro-geom. Then loops are searched for in the remaining graph using degree of freedom analysis. The loops found are solved by special loop routines. A solved loop is mapped onto one body and the new problem is reconsidered. Depending on the power of the special routines the problem is eventually solved.

After action analysis a Ppt-on-1nR loop occurs between bodies 7, 9, and 8. To solve it a special routine Ppt-on-1nR is required. Unfortunately, this loop occurs rarely in other mechanisms if at all. One

---

\(^1\)The equations originate from the 3 Identical, 1 driving Revolute, 6 Revolute, 8 Cylindrical, 1 pt-on-ln, and 1 pt-on-pl constraints.
The weakness of Kramer’s approach is that a lot of special routines are required. In the prototype described in [Kra92b] the loop of the saber saw is not included.

The approach by Bouma, Fudos, Hoffmann, Cai, and Paige [BFH+95] is based on finding subgraphs which are rigid according to the Kutzbach-Grübler criterion. For instance, the subgraph induced by the bodies 1, 2, 3, and 4 might be rigid. It consists of \((4 - 1) \cdot 6 = 18\) degrees of freedom of which \(6 + 6 + 4 + 4 = 20\) are removed by two identical and two Cylindrical joints. Such a loop is solved by numerical means. Alike Kramer’s approach the rigid subgraph is shrunk into a single body and the problem is reevaluated until it is completely solved.

There are three drawbacks to this approach. First, the criterion for rigidity used to find the solvable loops is not always exact. The Kutzbach-Grübler formula only holds if there is no redundancy present. In practical assemblies this need not to be the case. Often redundancy is part of the problem for some reasons. Second, finding minimal rigid subgraphs is an NP-complete problem as shown in [HLS97]. Third, the numerical method used at the end is again too less informative for practical assemblies.

In our approach search for possibly rigid loops is avoided. On the contrary, we solve loops by generating new (redundant) constraints and combine them with existing restriction. In this way, the information about restriction is spread in the constraint graph. Eventually there is enough information bundled at one spot to solve the loop. In the framework of constraint satisfaction theory this mechanism is called consistency enforcement.

As a first example consider a loop with 3 bodies, see Figure 4.2 on the left. Look at the restriction between body 1 and body 2. There is a direct Revolute joint between the bodies and a restriction over body

![Figure 4.2: A 3-loop and 4-loop.](image)
3 consisting of another two Revolute joints. Now, instead of applying a special routine for a RRR-loop, we compute the restriction of two Revolute in series. This restriction is expressed again as a constraint. Then we combined it with the direct Revolute constraint. The result is the overall restriction between the bodies 1 and 2.

This is the atomic solving routine of the ROMA-solver: computing restriction of two constraints in series and combining them with the direct restriction. With this generic scheme also loops larger than 3 can be solved as shown in the following.

An example with 4 bodies is solved similarly, see Figure 4.2 on the right. We look at the restriction between the parts 1 and 3. There is not yet a direct constraint between the parts. But there are two paths with two constraints each. We compute the restriction implied by each path and combine them. At the end we know the overall restriction between 1 and 3. Note that this example can also be solved differently by computing the overall restriction between bodies 2 and 4 or between bodies 1 and 2 etc.

In general, the routine for computing the restriction of two constraints in series is applied for each path of length 2 and the result is added to the graph. Whenever there is a change in some constraints, the paths going through the changed constraint is reconsidered. This is done as long as there no new restriction is generated.

In the saber saw example we would simply take any two constraints in series, compute the new restriction, and add it to the graph as new constraint. For instance, we compute the restriction of the chain 7, 9, 11 and add a new constraint between 7 and 11. Then we could compute the restriction given by 7, 10, 12, combine that with the constraint between 12 and 11 and get another constraint between 7 and 11. Eventually no new restriction can be generated. Then the solving process stops. As a result we have a new description of the initial problem with more explicit information with which the questions mentioned before might be answered.

In constraint satisfaction theory this scheme is called consistency enforcement. At the end the problem is strongly 3-consistent, see Section 5.2.

For this approach two geometric routines are needed: one to generate restriction of two constraints in series and one to generate the combined restriction of two constraints in parallel. With these two rou-
tines loops can theoretically be solved as described previously. The routines for parallel constraints is called \texttt{Add} and is described in more detail in Section 4.4.2. The routine for constraints in series is called \texttt{Cat} and is described in Section 4.4.3.

Our approach also needs special routines but not so many as in \textsc{Kramer}'s approach. We only need a routine for each combination of two allowed constraints, whereas \textsc{Kramer}'s needs all combination of six constraints. In addition, we do not rely on the inaccurate analysis with Kutzbach-Grübler to search for places where the routines are applied. The ROMA-solver simply generates each possible restriction. Note that, nevertheless, the ROMA-solver only needs time cubic in the number of bodies in the worst case.

For our approach to work some points need to be analyzed.

- Is the restriction between any two bodies representable by our constraint language?
- If it is representable, can it be computed?
- Is it sufficient to look only at each path of length 2 to solve the problem?

All questions are answered with no. No, we cannot represent all restrictions implied by our basic constraints set. No, we cannot compute everything. No, the problem need not to be solved entirely. Nevertheless, the approach works fine for almost all relevant cases. The saber saw for instance can entirely be solved by this approach. In the following, we argue why.

4.2 Constructing and Solving

In general the assembly is a hard problem. Nevertheless, our experience shows that the situations occurring in practice can in most cases be covered by the ROMA-solver.

One of the fundamental reasons why the ROMA-solver is working this satisfactorily is because the problem is solved on a high abstraction level. Instead of modeling geometric constraints as algebraic equations, we support kinematic joints directly. Thus, the constraints have a clear intuitive semantic to the engineer and do not just represent incomprehensible equations. Furthermore, when modeled as algebraic system,
4.2. Constructing and Solving

information about the geometric characteristic of the problem is lost. The ROMA-solver, however, can use this structural information.

The solving approach is similar to how an engineer constructs a mechanism. Local subproblems are considered and the results are combined with results of other small problems. The saber saw is a good example to illustrate this principle. Basically, it consists of four superimposed module. Each of the modules comprises a basic construction. When they are put together, the entire constraint graph might become difficult. However, still the modular structure can be exploited.

Thus, the ROMA-solver can be seen as a program that simulates a human constructor. Hence, there is a connection between constructibility and solvability. Cases that are easy to construct are also easy to solve. The other way round, non-solvable cases often correspond to cases that are not easy to construct. Mechanisms, however, are typically clearly constructed.

But contrary to Kramér, where the idea of emulating a human constructor was also used, the granularity of our system is adapted more to the problem. Instead of having a special routine for each loop, we generate new constraints out of pairs of constraints. Hence, fewer routines are needed without loss of functionality. Moreover, our approach is more flexible because. Contrary to Kramér the results of our computations are again represented as constraints and not as a number of DOF which is in fact too restrictive.

For algebraic solvers difficult cases need not necessarily correspond to difficult constructions. In the ROMA-solver this connection is established. Thus, it is easier to argument for the vendor of a CAD/CAD system if there that the solver covers the human solvable cases.

In addition, in our approach it is much easier to decide for which cases a routine should be provided because the impact can directly be estimated. In an algebraic system the question would for instance be whether quadratic equations are handled or not. The implication of that decision are not immediately clear. Whereas the effects of supporting a Revolute and Prismatic in series are more obvious.

After these general remarks about our approach we describe in the next section how it is actually realized. First, we discuss how restrictions are represented as constraints and how it is computed. The implementation of the geometric routines is not part of this thesis and is therefore only described from a client's point of view.
4.3 Representing Constraints

In general, an assembly constraint could be any restriction imposed on the relative location of two bodies. The question is how to represent the restriction between bodies. On the one hand, the representation should be powerful enough such that the engineer can model his intent and such that the intermediate restriction generated during the solving can be represented. On the other hand, the solver must still be able to compute the result of combinations of constraints efficiently.

Basically, we could allow any algebraic equation involving a parameterization of the relative location. However, then the solver must deal with all of this constraints and it would be a general equation solver.

In the ROMA-solver we restrict ourselves to representing constraints by a conjunction of base constraints as given in Section 3.1.1. In addition, we allow a disjunction of constraint combination in order to deal with discrete DOF. In general, a constraint is given as

\[ \bigvee_i \bigwedge_j \text{constr}_{ij} \]

where each \( \text{constr}_{ij} \) is an instantiation of a base constraint given in Section 3.1.1. For instance, the following expressions are allowed constraints:

- \( \text{pt-coi-pt}(\cdot, \cdot) \)
- \( \text{pt-coi-pt}(\cdot, \cdot) \lor \text{pt-on-sp}(\cdot, \cdot) \)
- \( \text{pt-on-pl}(\cdot, \cdot) \land \text{pt-on-ln}(\cdot, \cdot) \)
- \( (\text{pt-coi-pt}(\cdot, \cdot) \land \text{th-coi-th}(\cdot, \cdot)) \lor \text{pt-coi-pt}(\cdot, \cdot). \)

The constraint representation is closed under the operation Add combining two constraints in parallel. If we combine the two constraints

\[ \bigvee_{i=1}^{a} \bigwedge_{j=1}^{b_{ij}} \text{constr}_{ij}, \bigvee_{i=1}^{a'} \bigwedge_{j=1}^{b'_{ij}} \text{constr'}_{ij} \]

the result is

\[ (\bigvee_{i=1}^{a} \bigwedge_{j=1}^{b_{ij}} \text{constr}_{ij}) \land (\bigvee_{i=1}^{a'} \bigwedge_{j=1}^{b'_{ij}} \text{constr'}_{ij}) \]
4.3. Representing Constraints

which can be rewritten as

\[ \bigvee_{i=1}^{a'} \bigvee_{i'=1}^{a} \left( \bigwedge_{j=1}^{b_i} \text{constr}_{ij} \land \bigwedge_{j=1}^{b'_{i'}} \text{constr}'_{ij} \right) \]

to be in the required form \textit{disjunction of conjunction of base constraints}. For instance:

\[ ((a \land b) \lor (c \land d)) \land ((e \land f) \lor (g \land h)) \equiv (a \land b \land e \land f) \lor (a \land b \land g \land h) \lor (c \land d \land e \land f) \lor (c \land d \land g \land h) \]

However, the representation is not closed under the operation \textit{Cat}. For instance, assume that we have three parts 1, 2, and 3 in 3D-space. Between 1 and 2 there is a \textit{pt-on-sp} and between 2 and 3 is second \textit{pt-on-sp}. Then the allowed location for 3 relative to 1 is given as the interior of a large sphere whose radius is the sum of the radii of the input spheres minus the interior of a sphere whose radius is the difference of the radii of the input spheres. This restriction is not representable by a disjunction of conjunction of our base constraints. Nevertheless, most constraint combinations can be computed in series. For instance, two \textit{Revolute}, which is one of the joints that is most often used, in series can always be computed.

It is more powerful to represent mobility by means of constraints than by mapping it on remaining DOF as in KRAMER’s approach. With that approach only manifolds can be represented, whereas with our approach, disjunctions of manifolds of different dimension can be represented (variety). Furthermore, we are more flexible since it is not necessary to map the information onto another data type. The information is directly stored as constraints which suits the solver and the user.

Practical tests have shown that the result of most assembly situations in practice can in fact be represented. In addition, there is often a solving sequence passing intermediate constraints that are also representable. The ROMA-solver tries all of them simultaneously such that it will find the solvable sequence if it exists.

Another advantage of our design is that if the representation seems to be too restrictive, it can be enhanced without disturbing the consistency enforcement strategy. A first enhancement is to add new constraint types. Note, however, that enhancing the constraint type set
\begin{tabular}{|l|l|}
\hline
List$<$List$<$BaseConstr$>>$ & c representation of this \\
int red & redundancy information \\
int deg & degeneracy information \\
int num & number of numerical instabilities \\
List$<$Constraint$*>$ & ilist constraints that have influenced this \\
\hline
Add & combine two constraints in parallel \\
Cat & combine two constraints in series \\
Satisfy & satisfy a constraint \\
AreEqual? & test for equivalence \\
IsRigid? & test for rigidity \\
IsIllegal? & test for illegal \\
GetMobility & return the remaining DOF \\
GetPriority & estimate the remaining DOF \\
\hline
\end{tabular}

\textbf{Figure 4.3: Data fields and routines of constraint.}

does also mean that the new types need also be supported by Add and Cat. The current constraint type set seems to be a good compromise between functionality and computability. Nevertheless, it is still interesting from the commercial point of view to be able to adapt the functionality to special clients. The ROMA-solver provides the CAD/CAM vendor with this possibility.

A way out of the computational restriction would be to add new routines to handled special combinations of constraints. The problem, however, often is that some problems do not have a closed algebraic solution. For instance, the Steward platform cannot be solved analytically. In such cases it is a good idea to take numeric routines to generate at least one solution.

In the next sections, we describe how the constraint is realized as a data structure and which routines can be applied on that structure.

\section{4.4 The Constraint Data Structure}

Basically, the ROMA-solver consists of two parts: a global solver and a local solver. The global solving includes routines to steer where to apply the two local routines Cat and Add. The global solver is described in more detail throughout the next chapters. The geometric solving routines—the local solver—deal with small subproblems of size 2 and
3. All of this routines are methods of the class constraint. Thus, this class corresponds to the interface between the global and the local solver.

In this section, we describe the constraint data structure. An overview is shown in Figure 4.3. The structure consists of data fields (upper part) and operations applicable on constraints. First, we describe the data fields, then the routines Add and Cat, and finally the other routines.

4.4.1 The Data Fields

In the previous section, we have seen that a constraint is represented as disjunction of conjunction of base constraints. This information is stored in the constraint data structure as a list of list of base constraints. The outer list corresponds to the disjunction, the inner lists to the conjunctions. For instance, the constraint

\[((a \land b) \lor (c \land d) \lor e)\]

is represented as a list of 3 lists, the first of which is a list including a and b, the second including c and d, and the third including e.

The additional information on redundancy, on degeneracy, and on the number of numerical problems is stored explicitly in red, deg, and num, respectively. Each of this fields is an integer indicated the degree of the respective information. The results are accumulated during the two routines that manipulates constraints, namely Add and Cat. Note that the remaining mobility is not stored explicitly. This information is implicitly included in the representation of the constraint and can be dynamically compute by the routine GetMobility, see Section 4.4.4.

In order to track the origin of new constraints during Cat we store references to other constraints in ilist. Whenever a new constraint is generated by Cat, the reference to the two input constraints is stored in that list. Now, if we look at a constraint and get some information, we know the set of possible sources of the particular information.

4.4.2 The Procedure Add

The procedure Add is used to compute the combined restriction of two constraints in parallel. In the context of the assembly this means solving of 2-body-problems. The set-up is the following. Given two parts with two constraints between them, compute the combined constraint, see
Figure 4.4. The case of more than two constraints between the parts is computed by an iterative invocation of Add. More details about the implementation of Add can be found in [OMSE99].

For instance, consider the situation in Figure 4.5 with two bodies and two pt-coi-pt constraints in parallel. It is specified as follows:

\[ \text{pt-coi-pt}(pt(0, 0, 0), pt(0, 6, 4)) \land \text{pt-coi-pt}(pt(3, 4, 5), pt(3, 1, 0)) \].

The points \( pt(0, 0, 0) \) and \( pt(3, 4, 5) \) are given locally with respect to the coordinate system of the first body, whereas \( pt(0, 6, 4) \) and \( pt(3, 1, 0) \) are locally with respect to the coordinate system of the second body.

Two point coincidences between the same two bodies can only be satisfied if the distance between the point pair on each body is equal. Hence, this is checked first. The distance between the point pair on the first body is

\[ \sqrt{(3-0)^2 + (4-0)^2 + (5-0)^2} = \sqrt{9 + 16 + 25} = \sqrt{50}. \]

The one on the second body is

\[ \sqrt{(3-0)^2 + (1-6)^2 + (0-4)^2} = \sqrt{9 + 25 + 16} = \sqrt{50}. \]
They are equal. Thus, the constraints can be satisfied simultaneously.

The check that was performed corresponds to a redundancy. That is, partial information of the second constraint must fit partial information of the first constraint. If this is not the case, then the situation is illegal and overall solving can be stopped immediately.

In our example the check passes and a degree of redundancy is detected. For further computations we are interesting in having an explicit, redundancy-free representation of the two constraints as well as explicit information about the redundancy. This can be done with a pt-coi-pt and an additional constraint requiring that the vector between the two point pairs need to be equal. This corresponds to a vc-coi-vc. The result can be represented without redundancy as:

\[
\text{pt-coi-pt}(\text{pt}(0, 0, 0), \text{pt}(0, 6, 4)) \land \\
\text{vc-coi-vc}(\text{vc}(3, 4, 5), \text{vc}(3, -5, -4)).
\]

Note that the length of the two vectors does not have a semantic, only the direction is relevant. In the example, however, also the lengths of the two vectors need to be equal. However, that has already been tested when the distance between the points was checked.

The final situation owns 1 DOF: a rotation about the vectors. A pt-coi-pt removes 3 DOF, whereas a vc-coi-vc removes 2 DOF. This sums up to the expected 5 DOF that have been removed.

There is another case occurring when two pt-coi-pt are in parallel. If the two constraints had been identical, meaning that the coordinates of the point pairs had the same value, then the redundancy-free result would be one of the input constraints with additional 3 degrees of redundancy.

Due to the fact that constraints are represented using lists of base constraints, the result of Add can always be given perfectly as shown in Section 4.3. For instance, the combination of 6 pt-on-pl removes 6 DOF. It can be stored easily. However, is the result rigid? Does it have finitely many solutions or no solution at all? There is no routine known to check that. Thus, the 6 pt-on-pl are stored until possibly new constraint are added which may simplify the computations. In this way, information is preserved. However, some answer may not be given.

An even worse example is the Steward platform already mentioned in Section 2.2.3, see Figure 2.4. The Steward platform consists of two bodies with 6 pt-on-sp between them. Theoretically each pt-on-sp removes 1 DOF, such that the result should be rigid. However, depending
on the distances and the coordinates of the points, there can be up to 40 discrete solutions even if the points attached at one body are coplanar [WH98]. Up to now no analytical solution to the Steward platform is known.

Beside computing the combined restriction Add tries to discover and extract redundancy, degeneracy, and numerical instability. This information is accumulated during the computation and stored in the respective fields of the constraint data structure.

In the previously mentioned examples we have already seen how redundancy is extracted out of constraints. When two constraints in parallel already own a redundancy of \( red_1 \) and \( red_2 \), the combined constrained has a redundancy of \( red_1 + red_2 \) plus the redundancy extracted of the base constraints. The values for degeneracy and numerical instability are computed also as sum of the input fields. Note that the remaining mobility is computed dynamically by the GetMobility routine and not stored in the data structure. The list of the constraints that influenced the result, ilist, is simply the union of the ilist of the input constraints plus the reference to both input constraints.

As we have seen in the examples the technique to solve a 2-body-problem is based on analyzing the situation with certain geometric rules or theorems. These are used to simplify the situation. In the current version of the ROMA-solver the Add routine is implemented using term-rewriting techniques. These rules describe under which preconditions which output constraint are generated. There is a rule for almost all combinations of two base constraints. Note that these rules do not only work on the symbolic level, but need to consider the actual coordinates of the geometric entities to decide what output is generated. Hence, the ROMA-solver is an instance-solver.

In Kramer's approach also rules to handle 2-body-problems are used, see [Kra92b]. Unfortunately, the constraints are represented not by a list of base constraints but by the remaining DOF, split into RDOF and TDOF. This, however, is a too strong simplification for certain cases and hinders solving. Furthermore, the rules of the ROMA-solver are able to combine parts of a constraint with the part of another constraint instead of treating the constraints as whole. This concept saves resources during implementation. For instance, the rule to combine the two pt-coi-pt can also be applied if there are other base constraints present between the parts.
4.4. The Constraint Data Structure

Figure 4.6: Procedure Cat.

Figure 4.7: Two pt-coi-pt constraints in series.

4.4.3 The Procedure Cat

The Cat routines handles two constraints in series. Given three parts 1, 2, and 3 with a constraint between 1 and 2 as well as one between 2 and 3, compute the restriction between 1 and 3, see Figure 4.6. The concepts behind Cat are in more detail described in [OMSE98a].

Alike Add, Cat is also not perfect. However, the computation of the result is not the problem but the representation is. As shown in Section 4.3, the constraint representation used is not closed under Cat. In the case where Cat cannot represent the result, it always returns a subset of the actual restriction. In this way, we do not lose solutions, which is important for the global solving algorithm.

Two pt-coi-pt in series can be handled perfectly. Consider for instance the example shown in Figure 4.7. It is given by

\[
\text{pt-coi-pt}(pt(1,0,0), pt(0,2,0)) \\
\text{pt-coi-pt}(pt(2,0,1), pt(0,0,0))
\]

The point \(pt(1,0,0)\) is attached at body 1, the points \(pt(0,2,0)\) and \(pt(2,0,1)\) are attached at body 2 and the point \(pt(0,0,0)\) is attached at body 3. We are now interested in computing the restriction between body 1 and 3.
In the example the relative mobility of body 3 with respect to body 1 is a translation on a sphere plus an arbitrary rotation. The radius of the sphere is given by the distance of the 2 points attached at body 2. Hence, the result is

\[ \text{pt-on-sp}(pt(1,0,0), \text{sp}(pt(0,0,0), r)) \]

where \( r \) is the radius of the sphere given by

\[ r = \sqrt{(2-0)^2 + (0-2)^2 + (1-0)^2} = \sqrt{4 + 4 + 1} = \sqrt{9} = 3. \]

There is a special case to be distinguished. If the two points attached at body 2 coincide, then the radius \( r \) would be 0. In that case the \text{pt-on-sp} collapses to a simple coincidence of points represented as \( \text{pt-coi-pt}(pt(1,0,0), pt(0,0,0)) \).

A more complex case are two Prismatic joints in series, see Figure 4.8. This situation is specified by

\[ \text{Prismatic}(fr_1, fr'_2) \]
\[ \text{Prismatic}(fr''_2, fr_3) \]

The shift directions are given by the \( x \)-axes of the frames, see Section 3.1.3. Two cases must be distinguished. If the directions of the constraints at the middle body are collinear, then the result is \( \text{Prismatic}(fr_1, fr_3) \), see Figure 4.8 on the left. Otherwise the result is

\[ \text{th-coi-th}(th_1, th_3), \text{vc-ang-vc}(vc_1, vc_3, \alpha) \]

The geometric entities are constructed as follows: \( th_1 \) and \( th_3 \) are the trihedrals of \( fr'_2 \) and \( fr''_2 \) expressed locally to body 1 and 3, respectively, \( vc_1 \) and \( vc_3 \) are the \( x \)-axes of \( fr'_2 \) and \( fr''_2 \), also locally to body 1 and 3, and \( \alpha \) is the angle between the \( x \)-axes of \( fr'_2 \) and \( fr''_2 \).

A like \text{Add}, \text{Cat} is also realized using rules. Whenever a rule generates
4.4. The Constraint Data Structure

a base constraint, this constraint is stored in the result. The input constraints used are marked to prohibit double usage. When more than one rule generates a result, those are combined by Add to obtain a normalized representation.

The accumulation of the additional information fields for Cat is also straightforward. Note, however, that contrary to Add, where the result replaces the input, in Cat a new constraint is added to the constraint graph. This constraint is redundant because it is entirely derived by already existing constraints. This must be taken into account when computing the information fields.

New redundancy cannot occur during Cat such that principally red is 0. However, as each basic constraint detected by Cat is entered into the constraint graph and we only want to compute redundancy of the initial situation, we need to compensate. Hence, in each basic constraint we return red to be minus the remaining mobility of the constraint as stated in Table 3.4. This is done for each basic constraint detected during Cat. At the end of Cat the resulting constraints are normalized with Add, where the red fields are summed up.

Whenever there is numerical instability detected, we stored that in the fields of the resulting constraint. The numerical instability of the input is not added as the constraints remain in the graph. Degeneracy cannot occur during Cat. Hence, deg is always 0 in the resulting constraint. The list of influencing constraints ilist is simply the ilist of the two initial constraints plus the reference to the initial constraint themselves.

4.4.4 The Queries

In order to be able to work with the constraints the global solver and also its client need to have access to the data of the constraint data. This is possible by the following routines. Note that these routines are simple. However, they require that combined constraints have been normalized using Add.

Satisfy

This routine satisfies a constraint by bringing the two involved bodies into an allowed relative location. This is possible as long as the representation of the constraint is feasible. Often there are many possible
transformation to do that. Satisfy chooses one of them which is easy to compute and which involves as small rotations as possible.

Note that satisfying can be done by either moving the first or the second body. Later, however, we use this routine to instantiate solutions. Then we need to steer which body is changed. We assume from now on that Satisfy always moves the second body and leaves the first body at its location.

For instance, to satisfy the constraint
\[
\text{pt-coi-pt}(pt(1,0,0), pt(-1,-2,-3))
\]
first the global position of the two points is determined. Then the vector between them is computed and the second body is moved along the vector until the two points coincide. There is no rotation involved.

\textbf{AreEqual}?

\textbf{AreEqual} checks whether two constraints are equivalent. Equivalence means that the constraints representation is equivalent, not that they define the same solution set. That problem would be hard to decide. Moreover, in order to be equivalent the constraints must hold the same redundancy, degeneracy and numerical problems. It is not checked whether two different representations define the same solution space.

\textbf{IsRigid}?

The routine \textbf{IsRigid}? checks whether the constraint is a \texttt{fr-coi-fr}. There is not much work done in this routine. However, as we have seen in Section 4.4.2, a precondition is that the Add routine could handle the combination of base constraints. This is not always the case. Hence, it might be the case that \textbf{IsRigid}? returns false, although the constraint represents a rigid constraint. If \textbf{IsRigid}? returns true, then the constraint is definitely rigid. As we shall see later, the latter case is important for the algorithms.

\textbf{IsIllegal}?

This routine returns whether the constraint is illegal. This is done by looking for an \texttt{illegal} in the constraint representation. The same limitations as for \textbf{IsRigid}? also apply here. The positive result is definite, the negative one might be uncertain.
4.5. The Architecture

GetMobility

This routine generates the information on remaining mobility, redundancy, degeneracy, and numerical instability of the constraint, much like Add. But whereas Add only tries to find a redundancy-free representation and to detect rigid constraints, GetMobility goes further and tries to quantify the remaining DOF using (expensive) numeric methods. Contrary to the other queries, GetMobility is a costly routine and is therefore not used during the solving process, but only at the end for presentation of the results.

GetPriority

GetPriority computes an estimate of the restriction of the constraint. It is used during solving instead of the more expensive GetMobility routine. The remaining DOF can be estimated by adding up the DOF of the base constraint contained in the constraint. This sum corresponds to the value of the Kutzbach-Grübler criterion, see Section 3.3.

Contrary to the approach of BOUMA ET AL the number obtained is not used to decide whether a loop is expected to be rigid, but only to sort the paths to be considered. Thus, an inaccuracy during the routines does only changes the running time but not the solvability itself.

4.5 The Architecture

The ROMA solver-software is a ready-to-use solver for the assembly problem. First, it was designed as a prototype. In the meantime it has reached the level of professionalism to be released in the current version of the Swiss Precision Engineer CAD/CAM-system.

The ROMA-solver consists of several levels of abstraction, which are described briefly in the following, see Figure 4.9. All in all the ROMA-solver consists of about 60’000 lines of code, entirely written in C++. It is designed to work with the Gnu and Sun Compiler under Unix and with Visual-C++ under Windows.

To its client—the CAD/CAM-system of Precisionsoft—the ROMA-solver simply provides a constraint graph data structure. Bodies and constraints can be added, removed, and changed in various ways. The internal representation of the constraint graph is not influenced by the
editing sequence. Thus, *variational design* is supported, which is an important requirement to a modern CAD/CAM-system. There is also a function `Solve` to solve the current constraint graph, see Section 5.3. Furthermore, there are several routines to inquire the computed result, such as the remaining mobility between two parts, the rigidity components, the redundancy, etc. These routines are described in more detail in Section 5.4.

The interface between the global and the local solver is based on the data structure `constraint` as described in Section 4.4. The local solver basically consists of implementing the abstract data type `constraint`. The main routines are `Add` and `Cat` each consisting of numerous rules to cover the relevant cases.

As seen in Section 4.4.2 and 4.4.3, `Add` and `Cat` both comprise two steps. The first step is to analyze the situation. The second step is to compute the geometric entities needed to represent the result. In both steps basic geometric routines are used. These routines are encapsulated in a separate layer providing geometric objects and functions. The objects are the entities used to describe constraints, see Table 3.3 plus entities occurring during solving, for instance circles. On those objects various operations can be performed. For instance, intersection of two objects, test of relative position of objects, construction of objects by other objects, and so on.
The geometric layer provides two different kinds of geometric objects. One with absolute coordinates and one with coordinates relative to a coordinate system. Those objects are transformed whenever their coordinate system is moved. The local solver uses both kinds, whereas in the global part and in the interface to the CAD/CAM-system, only the relative objects are visible.

The next layer handles coordinate systems. A coordinate system is an item which can arbitrarily be placed into 3D-space. Its current position and orientation is represented by a frame, see Table 3.3. To each body in the constraint graph a coordinate system is attached. The constraints are expressed by geometric entities relative to the rigid bodies. Thus, they are automatically moved whenever their body is moved. Furthermore, if two bodies are rigid with respect to each other, their coordinate systems are coupled. Transforming either part then affects the geometric elements of both parts. This functionality is realized by a new extended implementation of the Union/Find data structure, discussed in Section 7.6.

For both the geometric layer and the coordinate system layer linear algebra routines are used. They are realized in a separate layer. There are vectors and matrices realized with homogeneous coordinates. Multiplication, addition, subtraction, test for collinearity, perpendicularity and various constructors are implemented. The routines to deal with numeric errors are also located here.

4.6 Summary

In this chapter, we have presented a new scheme to solve assembly problems. It is based on generating restriction of 2 constraints in series with the procedure \texttt{Cat} and adding them to the problem with \texttt{Add}. The approach does not require search for solvable loops and can be implemented with fewer special routines compared to Kramer's approach.

The approach has several limitations. First, the constraint representation is not closed under \texttt{Cat}. Second, not all cases of parallel constraint can be computed. Third, the solving need not to be finished after having generated all new information.

However, due to the fact that solving resembles constructing, the limitation are not severe in assembly problems. In most cases the solution can still be computed. In addition, the approach is well scalable:
new constraint types might be added, or new rules to deal with special cases could be implemented as soon as they are known and necessary.

The ROMA-software is organized in several levels and owns thin interfaces such that the implementation of the modules can be done independently. In particular, the interface between the global solver and the geometric routines only comprises the constraint data structure.

In the following chapter, we analyze the solving process. For this purpose we formalize it using constraint satisfaction theory. Among others things we shall answer the question whether the solving is finished after having applied Cat and Add in the graph.
Chapter 5

The ROMA-Solver

In the previous chapter, we have presented the basic solving scheme based on the procedures Add and Cat. In this chapter, we formalize this process by using constraint satisfaction theory.

In the terminology of constraint satisfaction theory the strategy of the ROMA-solver is called consistency enforcement, more exact enforcement of strong 3-consistency. After having applied Cat until there is no new restriction derived, the assembly might be solved completely. There is a test found by FREUDE to decide whether or not the problem is solved. However, this need not to be the case. If the result is not entirely solved, the computed output can nevertheless be used. We show how to deal with such situations. In addition, we show how to explore the final situation to gain insight into the mechanism.

In constraint satisfaction theory variables are usually assumed to have finite domains. Only finitely many values can be assigned to a variable. In the assembly, however, we need to deal with uncountable domains, namely with $SE(3)$. Thus, we need to adapt the solving algorithms.

This chapter is organized as follows. In Section 5.1 we give a short introduction to constraint satisfaction theory. Then, in Section 5.2, the concept of consistency is explained. In Section 5.3, we show how to enforce strong 3-consistency in assemblies. In Section 5.4, the data structure obtained after consistency enforcement is analyzed.
5.1 Constraint Satisfaction Problems

A constraint satisfaction problem is a general way of stating problems which involve assigning values to variables such that a given set of constraints is satisfied. In our case we want to assign locations to bodies such that the geometric relations between the bodies are satisfied. The global solver only knows the concept of constraint, thereby abstracting from the fact that the constraints are geometric. This abstraction leads us to the notion of constraint satisfaction problem. Comprehensive introductions to those problems can be found in [Kum92, Tsa93, MTU94, May94, Fre95, VS96, MS98]. In the following, we introduce the basics of this theory.

A constraint satisfaction problem (CSP) is described by a triple \((\mathcal{X}, \mathcal{D}, \mathcal{C})\) where

- \(\mathcal{X}\) is a set of variables \(\{1, 2, \ldots, n\}\)
- \(\mathcal{D}\) is a set of domains \(\{D_1, D_2, \ldots, D_n\}\).
- \(\mathcal{C}\) is a set of constraints \(\{C_1, C_2, \ldots, C_m\}\).

The domain \(D_i\) is the set of all values that can be assigned to the variable \(i\). Principally, the domains can be finite or infinite. In classical CSP theory often finite domains are assumed. Then the size of the largest domain is written as \(d\).

In the assembly the domains are all locations of coordinate systems in 3D-space, represented by the Euclidean group \(SE(3)\). This domain is non-finite, even uncountable. Therefore, we must adapt some of the algorithms from classical constraint theory to work in this framework.

A constraint consists of two parts. The constraint scope is a \(k\)-tuple of variables \((v_1, \ldots, v_k)\) to which the constraint applies. The constraint relation is a relation of arity \(k\), a subset of \(D_{v_1} \times D_{v_2} \times \cdots \times D_{v_k}\). The constraint relation indicates the allowed combinations of simultaneous values for the corresponding variables in the constraint scope.

A binary CSP is a CSP where each constraint scope is binary\(^1\). The topology of a binary CSP \(\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})\) is modeled as an undirected graph \(G = (V, E)\) where the vertices \(V\) represent the variables \(\mathcal{X}\) and there is an edge \(\{i, j\}\) in \(E\) if there is a (non-trivial) constraint with

\(^1\)From now on a CSP is assumed to be binary if not stated otherwise.
5.2. Consistency

a scope \((i, j)\) in \(\mathcal{C}\). This graph is called the constraint graph and was introduced informally in Section 3.2.

Theoretically constraints with the same scope are allowed in \(\mathcal{C}\). However, we combine them to avoid parallel edges in the graph. We use the notation \(C_{ij}\) for the combined constraint with scope \((i, j)\) and \(R_{ij}\) for the intersection of their relations.

We use the terms "vertex" ("node") and "edge" whenever treating graph related topics, and "variable" and "constraint" for CSP topics. Furthermore, we use \(V\) for \(X\) and \(E\) for \(C\) when there is no possibility of confusion.

A relation \(R_{ij}\) is universal if it is equal to \(D_i \times D_j\), in this case \(C_{ij} \notin \mathcal{C}\). \(R_{ij}\) is defined for all \(i, j \in X\) whereas \(C_{ij}\) exists in \(\mathcal{C}\) if and only if there is a real restriction on \(i\) and \(j\). The constraints are undirected. Hence, \(C_{ij} = C_{ji}\) and \(R_{ij} = \{(a, b) | (b, a) \in R_{ji}\}\).

An important concept for constraint solving algorithm is the join of two relations \(R_{ij}\) and \(R_{jk}\), defined as

\[
R_{ij} \times R_{jk} = \{(a, c) \in D_i \times D_k | \exists b \in D_j : (a, b) \in R_{ij} \land (b, c) \in R_{jk}\}.
\]

Note that contrary to the notion in database theory we directly perform the projection on the outer pair. Let \(P = (v_1, v_2, \ldots, v_p)\) be a path. The join over the path \(P\) is denoted \(|\times|_P\) and is defined as follows:

\[
|\times|_P = R_{v_1v_2} |\times| R_{v_2v_3} |\times| \cdots |\times| R_{v_{p-1}v_p}.
\]

An assignment or instantiation of a \(k\)-tuple of values \((a_1, a_2, \ldots, a_k)\) for \(k\) variables \((v_1, v_2, \ldots, v_k)\) means that the value \(a_1\) is assigned to the variable \(v_1, a_2\) to \(v_2\) etc. simultaneously.

An assignment for a set of variables \(V'\) is allowed if it satisfies all constraints in the subgraph induced by \(V'\). An allowed assignment for all variables is called a solution. The set of all solutions of a CSP \(\mathcal{P}\) is written \(\text{sol}(\mathcal{P})\). Two CSPs \(\mathcal{P}\) and \(\mathcal{P}'\) are equivalent if they describe exactly the same solution set, that is \(\text{sol}(\mathcal{P}) = \text{sol}(\mathcal{P}')\).

### 5.2 Consistency

Consistency describes whether allowed assignments can be extended to larger allowed assignments. Or the other way around: all instantiations
Figure 5.1: A 2-coloring problem.

which cannot be extended to larger allowed instantiations are excluded from the relation between the variables.

We start with some basic definitions. Let \( V' \subseteq V \) be a set of variables and \( i \notin V' \) be another variable. If a certain allowed assignment for \( V' \) can be extended to an allowed assignment for \( V' \cup \{i\} \) by assigning \( a \in D_i \) to \( i \). Then \( a \) is called a support for this assignment. We also say that the assignment is supported in \( i \).

A CSP is \( k \)-consistent if each allowed assignment for \( k-1 \) variables has at least one support in every other variable. Hence, consistency describes the fact that allowed assignments can be extended to larger allowed assignments. Arc-consistent means 2-consistent, path-consistent means 3-consistent. A CSP is strongly \( k \)-consistent if it is \( k' \)-consistent for each \( 1 < k' < k \). A CSP is directional \( k \)-consistent with respect to a given variable ordering if each allowed assignment for \( k-1 \) variables has at least one support in every other variable succeeding those \( k-1 \) variables in the ordering. The \( k \)-consistent closure of a CSP \( \mathcal{P} \) is a CSP which is equivalent and strongly \( k \)-consistent. For each CSP and for each \( k \) the \( k \)-consistent closure exists and is unique, see [Mon74].

Consider the CSP in Figure 5.1. It consist of 3 variables, each of which has a domain \( \{a, b\} \). The constraints require inequality. Therefore, the CSP is a 2-coloring problem. We see that each value for each variable has a support in each other variable. For instance, the value \( a \) for 1 has \( b \) as support in 2, \( b \) for 1 is supported by \( a \) in 2, the rest follows by symmetry. Thus, the CSP is 2-consistent. However, the CSP is not 3-consistent because the assignment \( (a, b) \) for \( (1, 2) \) cannot be extended to a larger allowed assignment for \( (1, 2, 3) \).

A CSP which is \( k \)-consistent needs not be \( (k-1) \)-consistent as Freuder points out in [Fre82]. A counter example is shown in Fig-
5.2. Consistency

Figure 5.2: A CSP which is 3-consistent but not 2-consistent [Fre82].

ure 5.2. This CSP is 3-consistent. Each allowed pair of values is supported. For instance, the pair \((a, b)\) for \((1, 2)\) is supported in 3 by the value \(a\). The pair \((a, a)\) for \((1, 3)\) is supported by \(b\) in 2. However, the CSP is not 2-consistent. The value \(a\) of 2 is neither supported in 1 nor in 3.

A CSP which is not \(k\)-consistent can be transformed into an equivalent one which is \(k\)-consistent. This procedure is called \(k\)-consistency enforcement or simply consistency enforcement. During that procedure allowed but unsupported \((k - 1)\)-tuples of values are searched for. These \((k - 1)\)-tuples are removed from the relation between the variables.

The CSP in Figure 5.2 for instance can be made 2-consistent by removing the value \(a\) of the domain of 2. Then each value of each variable in supported in each other variable. The CSP is also 3-consistent because each allowed pair is still supported in each other variable.

The history of consistency enforcement algorithms is long. In particular, algorithms for arc and path-consistency underwent through many revisions. An algorithm to make a finite CSP arc-consistent marks the beginning of CSP research, namely the famous Waltz-Filtering [Wal72, Wal75]. McKWORTH introduced the notion of consistency in [Mac77] and started to number the algorithms from AC-1 (corresponding to Waltz-Filtering) and PC-1 for the first arc and path-consistency enforcement algorithms, respectively. The latest results are AC-6 and PC-5 both of which are optimal with respect to space and time complexity. That is they reach the theoretical lower bounds.

AC-5, invented by VAN HENTENRYCK, DEVILLE, and TENG, is a generic algorithm, which can be instantiated for particular subclasses. The authors show that for those classes—for instance functional, anti-functional, and monotonic constraints, see Section 7.2—arc-consistency
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC-1</td>
<td>$O(dn)$</td>
<td>$O(dn)$</td>
<td>[Mac77]</td>
</tr>
<tr>
<td>AC-1</td>
<td>$O(d^3nm)$</td>
<td>$O(dn + m)$</td>
<td>[Mac77]</td>
</tr>
<tr>
<td>AC-3</td>
<td>$O(d^3m)$</td>
<td>$O(dn + m)$</td>
<td>[MF85]</td>
</tr>
<tr>
<td>AC-4</td>
<td>$O(d^2m)$</td>
<td>$O(d^2m)$</td>
<td>[MM88]</td>
</tr>
<tr>
<td>AC-5(^2)</td>
<td>$O(dm)$</td>
<td>$O(d^2m)$</td>
<td>[VDT92]</td>
</tr>
<tr>
<td>AC-6</td>
<td>$O(d^2m)$</td>
<td>$O(dm)$</td>
<td>[Bes94]</td>
</tr>
<tr>
<td>RPC</td>
<td>$O(dm(d + n))$</td>
<td>$O(dm(d + n))$</td>
<td>[Ber95]</td>
</tr>
<tr>
<td>PC-1</td>
<td>$O(d^5n^5)$</td>
<td>$O(d^2n^3)$</td>
<td>[Mac77]</td>
</tr>
<tr>
<td>PC-2</td>
<td>$O(d^5n^3)$</td>
<td>$O(n^3 + d^2n^2)$</td>
<td>[Mac77]</td>
</tr>
<tr>
<td>PC-3</td>
<td>$O(d^3n^3)$</td>
<td>$O(d^3n^3)$</td>
<td>[MH86]</td>
</tr>
<tr>
<td>PC-4</td>
<td>$O(d^3n^3)$</td>
<td>$O(d^3n^3)$</td>
<td>[HL88]</td>
</tr>
<tr>
<td>PC-5</td>
<td>$O(d^3n^3)$</td>
<td>$O(d^2n^3)$</td>
<td>[Sin95]</td>
</tr>
<tr>
<td>k-consistency</td>
<td>$O(d^kn^k)$</td>
<td>$O(d^{k-1}n^k)$</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.3:** Complexity of consistency algorithms for finite domains.

can be computed faster. RPC [Ber95] is a compromise between arc and path-consistency. PC-4 is the corrected version of PC-3 where some minor mistakes were made. An overview of arc and path-consistency algorithms is given in Table 5.3. Implementations of some of these algorithms can be found in [Nad88] and [Tsa93].

In the case of finite domains the implementation of arc-consistency enforcement is straightforward. As an example we introduce AC-3, see Algorithm 5.1. The routine consists of a loop—Lines 5-11—which considers all ordered pairs of variables. For each of these pairs $(i,j)$ the values of $D_i$ that have no support in $j$ are deleted, Line 7. If the domain $D_i$ is changed, all pairs $(h,i)$ need to be reconsidered, Line 9 except the pair $(j,i)$ because $D_i$ was reduced because of $j$. This will not, in turn, cause $D_j$ to be reduced. The search for a support is encapsulated in a routine called `Revise`, see Algorithm 5.2, which searches for unsupported values and deletes them.

Assembly solvers must deal with uncountable domains. Removing values of such domains is not as simple as in the finite case. Hence,

\(^2\)Only for special constraints, for instance: functional, anti-functional, monotonic.
AC-3 for finite domains

(1) procedure AC-3
(2) // enforce arc-consistency in finite CSPs
(3) begin
(4) Q ← \{(i, j) \mid C_{ij} \in \mathcal{C}\}
(5) while Q ≠ ∅ do
(6) (i, j) ← pop(Q)
(7) if Revise(i, j) then
(8)  // Di was reduced, reconsider relevant pairs
(9)  Q ← Q \cup \{(h, i) \mid h ≠ i, h ≠ j, C_{hi} \in \mathcal{C}\}
(10) end if
(11) end while
(12) end

Algorithm 5.1

Revise for finite domains

(1) procedure Revise(i, j)
(2) // remove values of Di not supported by j
(3) begin
(4) deleted ← false
(5) forall a ∈ Di do
(6)  if not HasSupport(i, a, j) then
(7)    Di ← Di \{a\}
(8)    deleted ← true
(9)  end if
(10) end forall
(11) return deleted
(12) end

Algorithm 5.2
Finding supports in finite domains

(1) procedure HasSupport(i, a, j)
(2) // return whether a for i is supported in j
(3) begin
(4) forall b ∈ D_j do
(5) if (a, b) ∈ R_{ij} then return true end if
(6) end forall
(7) return false
(8) end

Algorithm 5.3

we need a different Revise routine. The skeleton of the consistency enforcement algorithm, however, is the same.

The lower bound of the running time for AC-3 is Ω(d^2m) because for each of the 2m ordered pairs, Revise needs to be called once to assure that the needed supports are there.

The worst-case running time of AC-3 is O(d^3m). Each execution of the loop needs time O(d^2) for Revise. The question is how often it is called. At the beginning, for each of the constraints two pairs are inserted into the queue. A pair is reentered into the queue if the domain of either of the participating variables is restricted. The domain size is d. Thus, this can happen only 2d times. Therefore, each of the 2m ordered pairs is considered at most 2d + 1 times and the worst-case time complexity is O(2m · (2d + 1) · d^2) = O(d^3m).

The enforcement of k-consistencies is implemented similar to AC-3. Basically, there is a loop to check whether all the allowed assignments for (k − 1)-tuple of variables are supported. If an unsupported (k − 1)-tuple of values is deleted, then all items (k − 1)-tuples of variables that might be influenced need to be rechecked. Traditionally the procedure that checks a specific (k − 1)-tuple of variables for supports is called Revise independent of the size of k. Thus, for 3-consistency we need a Revise which checks a pair of variables against each other and for k-consistency, there is a version of Revise to check a (k − 1)-tuple against each other variable.

As can be seen in Table 5.3, AC-3 is not optimal for finite domains, neither with respect to time nor to space complexity. AC-4 achieves optimal time complexity by storing the supports for values such that
not everything needs to be rechecked. This, however, increases the
space complexity and many people prefer AC-3 to AC-4 as reported in
[Wal93]. AC-6 improves on this drawback by not storing each support
but only the last one. The remaining are computed on the fly. In this
way, the (amortized) running time is not affected thereby reducing space
complexity. Still, AC-3 is by far the most popular algorithm because it
is simple.

5.3 Enforcing Consistency in Assemblies

The first stage of the ROMA-solver is to enforce strong 3-consistency.
This yields a CSP which can be examined by suitable algorithms as
shown in the next sections. For strong 3-consistency it need to be
checked whether each value in the domain of each variable has a sup¬
port in each other variable (arc-consistency) and whether each allowed
assignment of each pair of variables is supported by each third variable
(path-consistency).

Unfortunately, after enforcement of strong 3-consistency the prob¬
lem is not always solved, even with perfect local routines. As a first
solution to this limitation the level of enforced consistency could be in¬
creased. However, this is not suitable because of two reasons. First,
the result of $k$-consistency requires the concept of constraints between
$(k-1)$ bodies. This extension dramatically increases the time needed for
implementation and requires routines like Add and Cat for up to $(k-1)$-
tuples. Second, if the results were represented by constraints involving
more than two bodies, the user would be over-charged. Assemblies are
typically based on binary constraints and binary information.

In assemblies arc-consistency is automatically implied. Initially the
domains of the variables are $SE(3)$. Provided that global solutions exist
assembly constraints do not restrict these domains. When we have a
legal combination of bodies, the whole assembly can always be placed
such that any variables has a particular position and orientation. When
there are no solutions at all, each domain is empty. This, however, can
be stored globally.

Nevertheless, combining parallel basic constraints by Add in a nor¬
malized way can be seen as replacement of arc-consistency. Alike arc-
consistency needs to be performed on each variable with respect to each
other variable, Add needs to be called for each pair.
**ROMA-Revise**

1. procedure RomaRevise(i, j, k)
2. // make $C_{ik}$ path-consistent with respect to $(i, j, k)$
3. begin
4. $C' \leftarrow \text{Add}(C_{ik}, \text{Cat}(C_{ij}, C_{jk}))$
5. if not AreEqual?($C'$, $C_{ik}$) then
6. $C_{ik} \leftarrow C'$
7. return true
8. else
9. return false
10. end if
11. end

---

**Algorithm 5.4**

For path-consistency we need a ternary Revise. This procedure takes a pair of variables $(i, k)$ and removes all values from $R_{ik}$ that are not supported by $j$. The procedure returns true if there is a change in $R_{ik}$ and false otherwise. Note that changes in consistency enforcement are always caused by more restriction not by less.

In the ROMA-solver the ternary Revise is implemented with the help of Cat, see Algorithm 5.4. As seen in Section 4.4.3, Cat computes the restriction implied by two constraints in series. In other words, the join over the two relations. The result of Cat$(i, j, k)$ contains only relative locations for $i$ and $k$ that are allowed by $j$. Hence, it is path-consistent with respect to the path $(i, j, k)$. The result is added to the already existing $C_{ij}$, Line 4. In this way, it is ensured that the existing restriction is not lost, and simultaneously the constraint representation is normalized.

The skeleton of the consistency enforcement algorithm is shown in Algorithm 5.5. It is based on PC-2, see Table 5.3. The later path-consistency algorithms like PC-3, PC-4, and PC-5 only differ in the implementation of the ternary Revise for the finite case.

Basically, the solving procedure consist of a loop dealing with all not yet considered triples. These triples are stored in a queue $Q$. If a triple $(i, j, k)$ is in $Q$, this means that we need to check whether all values in $C_{ik}$ have a support in $j$. In other words, RomaRevise$(i, j, k)$ must be called.
5.3. Enforcing Consistency in Assemblies

ROMA-Enforce

(1) procedure RomaEnforce
(2) // precondition: Add has been applied on each constraint
(3) // enforce strong 3-consistency by using Add & Cat
(4) begin
(5) \[ Q \leftarrow \{(i, j, k) \mid C_{ij}, C_{jk} \in C\} \]
(6) while \( Q \neq \emptyset \) do
(7) \( (i, j, k) \leftarrow \text{pop}(Q) \)
(8) if RomaRevise\((i, j, k)\) then
(9) RomaRefresh\((i, k, Q)\)
(10) end if
(11) end while
(12) end

Algorithm 5.5

ROMA-Refresh

(1) procedure RomaRefresh\((i, k, Q)\)
(2) // insert into \( Q \) all triples that need to be
(3) // reconsidered if \( C_{ik} \) was changed
(4) begin
(5) \[ Q \leftarrow Q \cup \{(i, k, m) \mid C_{km} \in C, m \neq i\} \]
(6) \[ Q \leftarrow Q \cup \{(m, i, k) \mid C_{im} \in C, m \neq k\} \]
(7) end

Algorithm 5.6

At the beginning each pair of variables needs to be checked against each other variable. Values in \( R_{ik} \) can only be unsupported with respect to variables that are connected to both \( i \) and \( j \). Hence, all \((i, j, k)\) where \( C_{ij} \) and \( C_{jk} \) exist are put into the queue, see Line 5.

Now we take an arbitrary triple \((i, j, k)\) out of the queue, Line 7, and make \( C_{ik} \) path-consistent with respect to the path \((i, j, k)\), Line 8. In the case of a change certain triples need to be rechecked. These triples are generated by RomaRefresh, see Algorithm 5.6. The triples comprise all path of length 2 in the constraint graph that contain the constraint \( C_{ik} \). Hence, all \((i, k, m)\) where \( C_{km} \) exists as well as all \((m, i, k)\) where \( C_{im} \) exists are put into the queue, see Lines 5 and 6.
The Algorithm 5.5 stops as soon as the queue is empty. Then we know that each constraint $C_{ik}$ only allows value combinations which are consistent with respect to each path $(i, j, k)$. A theorem of Montanari [Mon74] says that if all paths of length 2 are path-consistent with respect to all other variables, then the CSP is strongly path-consistent. Hence, provided that Add and Cat are perfect, the constraint graph is strongly 3-consistent.

A change of a relation, Line 8, always comprises the reduction of an entire DOF. Thus, it is ensured that Revise is only executed a number of times polynomial in the number of variables. The routine Revise itself needs constant time because the number of rules to be checked is constant. Hence, the worst case running time of Roma-Solve is $O(n^3)$.

As mentioned in Section 4.4.3, the Cat routine is not complete. That is, it can not always detect the full restriction implied on the outer pair of bodies. However, the restriction reported is always a subset of the actual restriction. More restriction than the actual restriction is not detected. Thus, the constraint graph obtained at the end is still equivalent to the initial problem. Furthermore, we know that no call of Add and Cat yields more restriction. Hence, we have enforced strong 3-consistency implied by our version of the ternary Revise.

In the current implementation Cat covers the cases which occur in normal assemblies such that in most cases the result is really strongly 3-consistent. If there are cases missing, the solving routines for those combinations may be included into the Cat routine without disturbing the overall solving. In addition, the special cases can directly be used orthogonally with other constraints.

Consistency enforcement in conjunction with local geometry routines provides us with a nice possibility to be able to scale the functionality of the solver. On the one hand, this is an advantage during development of the solver, since it can already be tested without full functionality. On the other hand, we are flexible to react on special demands coming from customers.

### 5.4 Interpretation of the Result

During the first stage the constraint graph is reformulated to an equivalent CSP which in most cases is strongly 3-consistent. However, strong 3-consistency may not be enough to entirely solve the problem. In this
section, we discuss under which circumstances it is solved and what can be done otherwise.

The answer to this question was given by Freuder [Fre82]. He showed the connection between the degree of consistency and a graph parameter, the width, which can be determined efficiently. The width was introduced by Freuder [Fre82]. However, the corresponding notion dates back to Szekeres and Wilf [SW68] and Matula [Mat68]. It is defined threefold as follows.

The **width of a vertex** $v$ with respect to a given vertex ordering is the number of adjacent vertices of $v$ preceding it in the ordering. The **width of an ordering** is the maximum width of all vertices with respect to it. The **width of a graph** is the minimum width of all its orderings. The width and linkage of a graph are equivalent [MMI72, Fre82, KT96].

Note that a graph with cycles has at least a width of 2 because one vertex of the cycle must be the last in the ordering. That vertex has at least 2 adjacent vertices in front. The width of a non-empty tree is 1 because each vertex has exactly one predecessor except the root, which has none. Likewise, the width of a non-empty forest is also 1. The width of a complete graph with $n$ vertices is $n - 1$ because in either ordering the final vertex has a width of $n - 1$.

The width of an arbitrary graph can be determined in time $O(n + m)$ as follows, [Fre82]. Start with $k = 0$. Remove every vertex with a degree smaller than $k$. If the graph is empty, $k$ is the width. Otherwise increase $k$ and continue. Eventually the graph will be empty. Then the current $k$ is the width of the graph. The linear running time is achieved if the vertices are organized in a list of lists of vertices with the same degree.

By the same algorithm a width-optimal ordering can be determined. Simply take the removal sequence from back to front.

As an example consider the graph in Figure 5.4. It is the graph of the elevator door, which shall be discussed in more detail in Section 8.1.1. In this graph there is no vertex with a degree smaller than 2. Therefore, we start directly with $k = 2$. With this $k$ we can remove the vertices 3, 5, 6, 7, 9, and 10, then the vertices 2, 4, and 8, and finally vertex 1. Now the graph is empty. Hence, its width is 2. A width-optimal ordering is given by the removal sequence from back to front. Therefore, $< 1, 6, 4, 2, 10, 9, 7, 6, 5, 3 >$ achieves a width of 2.

One of the central results in constraint satisfaction theory is the connection between the width of the graph and the level of consistency.
Theorem 5.1 (Freuder [Fre82]) There is a backtrack-free search order for a CSP if the level of strong consistency is greater than the width of the graph.

Proof Let \( k \) be the width of the graph and the graph be strongly \( k+1 \)-consistent. Take an ordering with a width of \( k \). In such an ordering each variable \( i \) is connected to at most \( k \) variables preceding it. Due to the fact that the graph is at least strongly \( k+1 \) consistent, each allowed assignment for those \( k \) adjacent variables must at least have a support in \( i \). Therefore, an allowed assignment for those \( k-1 \) variables can always be extended to \( i \). This holds for each variable. Consequently, we can assign all variables by following the ordering without backtracks.

The condition for backtrack-free search can be slightly improved as shown by Dechter and Pearl in [DP88]. They showed that it is not necessary to support \((k-1)\) -tuples in all variables if a variable ordering is fixed in advance. In order to assign the variables along the given ordering support is needed only in variables succeeding the \((k-1)\)-tuple. Hence, directional consistency with respect to the used ordering is sufficient to achieve backtrack-freeness. In a second step the authors observe that consistency needs be enforced only in the parents of each variable. This observation leads to the concept of adaptive consistency which is discussed in [DP88] also.

If there is a backtrack-free search order, a single solution can be found in linear time. Simply compute a width-optimal ordering in
5.5 Providing the Results

The ROMA-solver enforces strong 3-consistency. Hence, graphs with a width of 2 are solved. Unfortunately, this considers the width after the enforcement step. During consistency enforcement the width is often increased by adding new constraints, which are necessary to ensure the consistency. Typically we end up with an almost complete graph. Therefore, we do not expect to find a solution backtrack-free. In the next chapters, we provide methods to increase the number of graphs solved after enforcement of strong 3-consistency.

In the following, we present our strategy to deliver as much information about unsolved graphs as possible. We also argue that the constraints that are not satisfied need not necessarily be a disadvantage but can even help the engineer.

5.5.1 Instantiation

Assigning a solution is done as shown in Algorithm 5.7. First, the width is checked. If it is at most 2, the theorem of Freuder tells us that we can assign a solution backtrack-free. Otherwise we relax the problem. The relaxed problem is a minimum spanning forest. A forest has a width of 1. Hence, it can be assigned without backtrack. The priorities to define the minimum are determined by GetPriority, see Section 4.4.4, which assigns high priority to easy and restricted constraints.

In either case the resulting CSP has a width of at most 2. Hence, we can assign it along a width-optimal ordering by satisfying the constraints. Now we want to find a location for \( v_i \) which is consistent with the location of the already placed bodies. In order to achieve that, we need to satisfy all constraints to variables which precede \( v_i \) in the ordering. By \( \text{adj}_{v_i} \), we denote the set of vertices adjacent to \( v_i \) which precede \( v_i \) in the chosen ordering. There are three cases to be distinguished.
(1) **procedure** RomaAssign
(2) // precondition: CSP is strongly 3-consistent
(3) // assign a (partially) valid solution
(4) **begin**
(5) if width > 2 then
(6)     relax the problem (minimum spanning forest)
(7) **end if**
(8) compute a width-optimal ordering < v_1, ..., v_n >
(9) **for** i ← 1 to n **do**
(10) adj_{v_i} ← adj(v_i) \cap \{v_1, ..., v_{i-1}\}
(11) if adj_{v_i} = \{w\} then
(12)     Satisfy(C_{wv_i})
(13) else if adj_{v_i} = \{w_1, w_2\} then
(14)     c_{wv_i} ← Add(C_{w_1v_i}, C_{w_2v_i})
(15)     Satisfy(c_{wv_i})
(16) **end if**
(17) **end for**
(18) **end**
5.5. Providing the Results

1. \( \text{adj}_v_i \) is empty. Then an arbitrary location for \( v_i \) can be chosen. In the algorithm, \( v_i \) remains at its current location.

2. \( \text{adj}_v_i \) contains a single element: the vertex \( w \). Then the constraint \( C_{wv_i} \) is satisfied by moving \( v_i \).

3. \( \text{adj}_v_i \) contains two vertices: \( w_1 \) and \( w_2 \). In that case we need to find a location for \( v_i \) that satisfies both constraints \( C_{w_1v_i} \) and \( C_{w_2v_i} \) simultaneously. Due to the fact that \( w_1 \) and \( w_2 \) are already fixed and will never be changed again, these parts can be considered rigid. Thus, we can simply add the two constraints \( C_{w_1v_i} \) and \( C_{w_2v_i} \) to find the intersection of the possible locations for \( v_i \). Afterwards the resulting constraint is satisfied by moving \( v_i \).

Note that \( \text{adj}_v_i \) cannot contain more than two vertices. Otherwise the ordering would not have a width of 2.

In principle, instead of relaxing to a spanning forest, also a subgraph with a width of 2 could be assigned. However, there is no algorithm known to us which computes a minimal subgraph of width 2 efficiently.

After the call of \texttt{RomaAssign} some constraints may still not be satisfied. It might happen that the imperfection of \texttt{Cat} causes some constraints to be not explicit enough. That is, the representation is to difficult for \texttt{Satisfy}, or that the problem was relaxed to achieved a width of at most 2. As demonstrated by the examples in Section 8.1 this is more the exception that the rule. Most assemblies—even those including difficult kinematic chains—can be reduced to representations which can be assigned perfectly. Nevertheless, we need to provide a scheme to deal with those cases that cannot be solved entirely.

Assume that the solution contains unsatisfied constraints. When we assign it, in most cases the engineer is surprised, because he/she has a clear expectation of the result. Note, however, that all cases involve constraints that are unrestricted or in certain ways \textit{difficult}. Due to the fact that the solving is done in the same manner as the engineer thinks, namely by qualitative analysis of the situation, the infeasible cases for the algorithm correspond to situations that are also difficult for humans. Hence, this might well be the place where constraints are not clearly designed but are given too informally.

Furthermore, note that the final situation is often rigid when driving input constraints are used, see Section 3.1.4. Then unrestricted constraints might be the place where the engineer is currently working.
Procedure RomaExplore

1. procedure RomaExplore(a, b, T)
2. // precondition: CSP is instantiated by ROMA-Explore
3. // move b with transformation T
4. // reestablish a solution such that a remains fixed
5. begin
6. move b with T
7. if $C_{ab} \in C$ then Satisfy($C_{ab}$) end if
8. RomaAssign
9. end

Algorithm 5.8

Also in this case the wrong instantiation is a hint, where the next design step should be performed.

Finally, the assignment to the relaxed problem can be used for numeric approaches like Newton/Raphson or Numerica. These approaches need good initial values to approximate the intended solution efficiently. In the current version of the ROMA-solver we do not make use of numeric approaches since unsatisfied constraints occur seldom and if they do, they may still be useful as argued earlier.

5.5.2 Exploring the Solution Space

In situation with remaining mobility the engineer is interested in exploring the solution space by moving bodies according to the remaining mobility.

The procedure Satisfy is used to move a body to a single legal location. If there are several, an action is chosen which is as easy as possible. In general, this is difficult to define and to compute. In most cases, however, the choice is naturally. For instance, to satisfy a pt-coi-pt, the difference vector between the two points is computed and the second body is translated along that vector. This transformation does not include a rotation of the second body.

With that strategy of Satisfy we can explore the remaining mobility between two bodies $a$ and $b$, see Algorithm 5.8. This routines allows the engineer to move $b$ by transformation $T$. Then an instantiation is established which is legal and which leaves $a$ at its original place. By
5.5. Providing the Results

this the engineer has the feeling that he/she moves body $b$ relative to $a$. The net transformation of $b$ is not $T$ as this cannot be ensured under the condition that $a$ is fixed. However, part $b$ snaps to a location near to the intended one. If the routine is called with small transformations, the user gets the imagination that he/she can move $b$ according to the remaining mobility between $a$ and $b$.

Note that for this assignment the same limitations hold as for the initial assignment. Namely that we need to relax the problem and that satisfy may not be able to legally instantiate.

The routine Roma-Explore works as follows. First, the body $b$ is moved by $T$. This motion is only done in the data structure and is not directly performed on the screen. Then we satisfy the constraint $C_{ab}$. This brings $b$ into a legal location with respect to $a$. After that we reestablish a consistent assignment with Roma-Assign. This routine is deterministic. Compared to the last assignment it now works on a slightly different input graph, where the relative location of $a$ and $b$ was changed. The property of Satisfy not to move if not necessary, leads to a solution which reflects the cascading changes that $T$ has on the graph.

In this manner the engineer is able to move a body on the screen relative to another body and to find out what the relative mobility between the parts is. The handling of the parts is not necessarily optimal caused by the potential imperfection of Satisfy and Cat. But in most cases the handling is sufficiently accurate to provide a feeling of the remaining mobility.

Another possibility to explore the mobility of the mechanism in a more specified way is to use driving input constraints as described in Section 3.1.4. Each DIC has a free parameter which can be entered manually or be driven automatically. However, after having changed the parameter, the graph needs to be resolved entirely since the topology of the solution space may change because of degeneracy in singularities. The resolving, however, is fast enough to allow animation of most mechanisms. Note that in case of several solutions, it may happen that the mechanism jumps from one path to the other. This should be avoided by specified appropriate constraints manually.
5.5.3 Additional Information

In addition to a single assignment the ROMA-solver also computes information about remaining mobility between two bodies, redundancy, degeneracy, and numerical instability. All this information is accessed in the same manner, namely by examining the constraint between two parts and by subsequently calling GetMobility or IsRigid on this constraint.

For instance, if we want to have the remaining mobility between parts $i$ and $j$, we examine $C_{ij}$. If the constraint does not exist, then there is no restriction detected by the ROMA-solver and everything is allowed. Otherwise we can, for instance, look up redundancy by examining the red field accumulated by Add and Cat.

In addition, we can take ilist into account, the list of constraint having influenced $C_{ij}$, to locate the source of the effect. However, as mentioned before, consistency enforcement has the annoying property that it produces complete graphs fast. In these graphs often each constraint is influenced by each other constraint. Therefore, information can often be reported for the entire graph only. With the two strategies discussed in the following chapters the source of effects can be located in the biconnected components of the graph.

5.6 Summary

In this chapter, we have formalized the behavior of the global solver with constraint satisfaction theory. The solving is based on enforcing strong 3-consistency in the graph. For this approach we need a ternary Revise that can deal with uncountable domains. The routine Cat presented in the previous chapter serves for this purpose.

After enforcement of strong 3-consistency Freuder’s theorem tells us that all problems with a width of at most 2 are solved and an assignment can be found in linear time. If the width of the graph exceeds 2, we compute a minimum spanning forest, which represents a relaxed problem. A solution to this problem can be used either as starting point for numeric iterative methods or can directly be drawn on the screen. In the latter case the unsatisfied constraints, either complicated or unrestricted ones, are most likely the points where the engineer needs to work on the design.
5.6. Summary

Additional information like mobility, redundancy, degeneracy, and numerical instability can be determined by looking them up in the respective constraints. However, the source of this information cannot be located properly since constraints are often influenced by many other constraints.

In the following chapters, we discuss two strategies to improve on consistency enforcement. Both strategies lead to an increase in efficiency, to a clearer structure of the constraint graph obtained, and often to reduction of the width of the graph. With the use of the strategies almost all practical mechanisms can be solved satisfactorily.
Seite Leer / Blank leaf
In the previous chapters, we have seen that the ROMA-solver is based on enforcing strong 3-consistency. It is required that each pair of variables is supported by each other variable. To increase efficiency we are interested in considering only pairs that are really needed. Furthermore, the constraint graph is often complete after the enforcement such that we cannot report the source of redundancy and inconsistency anymore.

As mentioned in Section 4.2, assemblies often own a modular structure. The overall complexity is broken down into subtasks (modules), which are combined in the actual mechanism. Unfortunately, we cannot exploit these modules directly because they occur only implicitly in the constraint graph.

Typically a module contains at least one loop. As argued in the previous chapters the ROMA-solver tackles those loops by contracting two constraints in series to one. Doing that the lengths of the shortest paths between vertices are decreased.

Eventually this may yield paradoxical situations. Consider for instance the assembly of an airplane. At some time there might be a constraint between the pilot’s seat in the cockpit and the bearing of the rightmost wheel and another constraint between the bearing and the door of the refrigerator in the kitchen. Then these two constraints in series are concatenated to a new constraint between the seat and
the door. Of course there is a certain spatial restriction between these bodies. Maybe the ROMA-solver can even compute and represent this restriction. But is it really necessary to take it into account to solve the airplane assembly? Probably not. However, for enforcement of strong 3-consistency there has not yet been a concept to avoid the previously mentioned behavior.

Starting from the intuition that constraints belonging to different modules need not to be considered several questions arise. How do we detect the modular structure which is only implicitly present in the constraint graph? Is the user required to enter structural information explicitly? Instead could the modules be detected on the constraint graph level? And if yes what is the underlying structure?

Reconsider why we enforce strong 3-consistency. As explained in Chapter 4 the consistency enforcement aims at dealing with loop situations. But if there are no loops, only arc-consistency is needed. Hence, the set of bodies that are connected over loops seem to be the desired structure. These sets correspond to the *biconnected components* of the constraint graph. Thus, the idea is to enforce 3-consistency only in the biconnected components.

Now new questions arise. How can the results of the single biconnected components be combined to an overall solution? Do we compute everything if we are restricted to the biconnected components? Obviously we do not compute as much as for ordinary consistency because we deliberately refrain from considering certain constraint combinations. Can it happen that we miss something due to this restriction?

In this chapter, we aim at answering these question on a general level. Consequently, the results are applicable to all CSPs—finite or infinite. However, solving does only benefit from the new approach if the constraint graph is structured and contains more than one biconnected component. For the assembly this is often the case. The effect even increases in combination with the strategy to deal with rigid constraint, presented in the next chapter.

This chapter is organized as follows. Section 6.1 contains the basic definitions. In Section 6.2, a new consistency—block-consistency—is defined. Section 6.3 deals with extending block-consistency to consistency. We show that by enforcing strong 3-block-consistency, already sufficient information is generated. Finally, in Section 6.4, we adapt the ROMA-solver to enforce only block-consistency.
6.1 Definitions and Notation

In this section, we state the fundamental definitions needed for our approach. On the one hand, these definitions are related to graph theory. On the other hand, they come from constraint satisfaction theory.

Two vertices are adjacent if they are connected by an edge. The adjacency list of a vertex $v$, $\text{adj}(v)$, is the set of all adjacent vertices of $v$. The adjacency list of a set of vertices $V'$ is the set of all vertices in $V - V'$ which are adjacent to at least one vertex in $V'$. The degree of a vertex is the size of its adjacency list. A vertex with degree 0 is called isolated.

A path $P$ from $v$ to $w$, denoted $v \Rightarrow w$, is a sequence of vertices $P = (v = v_0, v_1, \ldots, v_{p-1}, v_p = w)$ such that each two consecutive vertices $v_i, v_{i+1}$ are adjacent. The length of a path $P = (v_0, v_1, \ldots, v_{p-1}, v_p)$, denoted $|P|$, is $p$. A path is simple if its vertices are pairwise distinct. A path $v \Rightarrow v$ is called a closed path. A closed path $(v_0, v_1, \ldots, v_p = v_0)$ is a cycle if the path $(v_0, v_1, \ldots, v_{p-1})$ is simple.

Two vertices are connected if there is at least one path between them. The maximal connected subgraphs of a graph are called its connected components. A graph is connected if it consists of a single connected component only.

Two vertices $i$ and $j$ are biconnected if at least 2 vertices need to be removed to disconnect $i$ and $j$. In the following, we define biconnectivity in terms of the existence of vertex disjoint paths. Two simple paths between the vertices $v$ and $w$ are vertex disjoint if they do not share any vertices except $v$ and $w$.

Two vertices are biconnected if there are at least two vertex disjoint paths between them. A subset $S$ of vertices of a graph $G$ is biconnected if $S$ is empty, contains a single vertex, or each pair of vertices in $S$ is biconnected in $G$. Note that the vertex disjoint paths are allowed to pass vertices outside $S$.

A biconnected component (block) $B$ is a maximal set of biconnected vertices. If there is a triple of distinct vertices $a, b, v$, such that each path from $a$ to $b$ goes through $v$ and there exists at least one such path, then $v$ is called an articulation point. A bridge is an edge connecting vertices which are not biconnected.

Note that blocks contain either a single vertex or more than two vertices. Blocks with exactly two vertices cannot exist.
The graph in Figure 6.1 is connected because there is a path between each pair of vertices. It consists of 8 blocks: \{1, 2, 3, 4\}, \{4, 6, 7, 8\}, \{5\}, \{9, 10, 11, 12\}, \{13\}, \{14\}, \{15\} and \{16\}. The vertices 4, 8, 9, 12, and 13 are articulation points. The edges \{4, 5\}, \{8, 9\}, \{12, 13\}, \{12, 16\}, \{13, 14\}, and \{13, 15\} are bridges.

Some of the following theorems are based on the notion of prefix-connected orderings. A prefix of an ordering is a set including each vertex from the beginning of the ordering up to a certain vertex. An ordering of a graph is prefix-connected if each prefix of the ordering induces a connected subgraph.

In Figure 6.1, the ordering indicated by the labeling of the vertices is prefix-connected. Each vertex—except vertex 1—has at least one adjacent vertex with a smaller labeling. The ordering

\[(1, 4, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)\]

is not prefix-connected because the subgraph induced by the prefix \{1, 4\} is not connected.

In this and the following chapter, the concept of redundancy plays an important role. Informally a constraint is redundant if it can be removed without changing the solution set of the CSP. The opposite of redundancy is explicitness. A constraint is explicit if each value pair in the relation of the constraint is part of at least one solution.

To define these terms more formally we need the notion of network-induced constraint. The network-induced constraint \(\overline{C}_{ij}\) between two variables \(i\) and \(j\) comprises the restriction imposed by the CSP without the direct constraint \(C_{ij}\). The associated relation is written \(\overline{R}_{ij}\).
6.1. Definitions and Notation

A constraint $C_{ij}$ is redundant if $R_{ij} \supseteq \overline{R}_{ij}$. A constraint $C_{ij}$ is explicit if $R_{ij} \subseteq \overline{R}_{ij}$. A constraint $C_{ij}$ is satisfiable if $R_{ij} \cap \overline{R}_{ij} \neq \emptyset$.

The Venn diagrams for different kinds of constraints are shown in Figure 6.2. The dot indicates a region where at least one pair of values must exist. We see that for redundant constraints, the allowed pairs include all allowed pairs of the network-induced constraint. Analogous, for explicit constraints, the allowed pairs of the network-induced constraint include the pairs allowed by the direct constraint.

Following the notation of [DD87] we slightly extend the concept of consistency introduced in Section 5.2. A pair of values $(a, b)$ for $i$ and $j$ is allowed by a path $i \Rightarrow j$ if $(a, b) \in |x|_P$. A pair of values $(a, b)$ is path-induced if it is allowed by each path from $i$ to $j$. A constraint $C_{ij}$ is path-consistent if each pair $(a, b) \in R_{ij}$ is path-induced.

The requirement for a constraint to be path-consistent is weaker than to be explicit, because explicitness requires that the remaining pairs are part of at least one solution. MONTANARI has shown that a pair of values is path-induced if and only if it is allowed by all paths of length 2, [Mon74].

A constraint is path-redundant with respect to a path $P$ if each pair $(a, b) \notin R_{ij}$ is also not in $|x|_P$. In the following, path-redundancy is mostly used for paths of length 2. Let $C_{ik}$ be path-redundancy with respect to the path $(i, j, k)$, then

$$\forall (a, c) \notin R_{ik} : \forall b \in D_j : (a, b) \notin R_{ij} \lor (b, c) \notin R_{jk}.$$

That is, each pair not in $R_{ik}$ is not supported by $j$.

Consistency enforcement, introduced in Section 5.1, adds redundancy to the CSP to make the constraints more explicit. A solution can be found without search as soon as each constraint is explicit. Then the graph is called the minimal network of a CSP $P$. The minimal network is unique to each problem $P$ [Mon74].
6.2 Block-Consistency

In this section, we define block-consistency (BC). BC is a weaker kind of consistency in the sense that it requires fewer value combinations to be supported. It leads to more efficient algorithms as well as to smaller graphs because fewer constraints are newly inserted. Nevertheless, we shall show in the next section that strong directional \( k \)-BC is sufficient for strong directional \( k \)-consistency if the given ordering is prefix-connected.

First, we define \( k \)-block-consistency.

**Definition 6.1** A CSP is \( k \)-block-consistent (\( k \)-BC) if each allowed assignment for \( k-1 \) biconnected variables has at least one support in every other variable.

Contrary to \( k \)-consistency, where all \((k-1)\)-tuples need to be supported, \( k \)-block-consistency only requires support for biconnected \((k-1)\)-tuples. Hence, \( k \)-block-consistency is a weaker property than \( k \)-consistency. Consequently, block-consistency can be enforced more efficiently.

**Strong \( k \)-block-consistency** is \( k' \)-block-consistency for each \( 1 \leq k' \leq k \). **Directional \( k \)-block-consistency** with respect to an ordering is block-consistency requiring support for a \((k-1)\)-tuple of biconnected variables only in variables succeeding the entire \((k-1)\)-tuple in that ordering.

Note that 2-block-consistency is equivalent to arc-consistency because each 1-tuple is defined to be biconnected. Differences between \( k \)-consistency and \( k \)-block-consistency only occur if \( k \) is at least 3.

The enforcement of \( k \)-consistency, \( k \geq 3 \), has the property that new constraints may be added to the graph. Simultaneously the width of the graph is possibly increased. The block structure may also be destroyed. Of course BC-enforcement may also insert new edges and increase the width. But the block structure is preserved.

**Theorem 6.1** The block structure is preserved during block-consistency enforcement.

**Proof** Block-consistency enforcement requires support only for biconnected variables. Therefore, only new constraints are inserted between vertices that are biconnected already. Thus, no two vertices can become biconnected and the blocks of the graph remain the same. \( \square \)
6.3. From BC to Consistency

Consequently, all properties defined in terms of blocks are invariant during BC-enforcement. For instance, the \((k-1)\)-tuples which must be considered during consistency-enforcement remain the same. Furthermore, the width of the graph is bounded by the size of the largest block, see [Fre85]. This bound is also invariant during BC-enforcement such that a bound on the width after BC-enforcement can be given. The fact that the width is bounded suggests that a finite CSP can be solved in time exponential in the size of the largest block. This was shown in [Fre85, Fre94].

In the ROMA-solver, see Algorithm 5.5, also redundancy, degeneracy, and numerical instability are computed by enforcing strong 3-consistency. To locate the source of this information it is important to keep track which constraints have influenced which other constraints. If ordinary 3-consistency were enforced, each triple of variables is considered. This may lead to a complete graph where each constraint was possibly influenced by each other constraint. In the case of block-consistency, a constraint can only be influenced by constraints of the same block. Thus, the source of redundancy, degeneracy, and numerical instability can be located on the block level.

6.3 From BC to Consistency

Block-consistency can be enforced more efficiently than ordinary consistency. In this section, we show that is no disadvantage. First, we show that strong directional \(k\)-BC is sufficient for strong directional \(k\)-consistency if prefix-connected orderings are used. Then we show that a CSP which is strongly 3-block-consistent can be made strongly 3-consistent efficiently. In addition, we see that domains, rigidity, illegality, and redundancy are already computed during BC-enforcement. Therefore, it is possible to change the ROMA-solver such that it only computes strong 3-block-consistency.

6.3.1 Directional Consistency

First, it is shown that the intersection of a connected set and the adjacency list of a vertex are biconnected. This property is used in Theorem 6.2 to argue that supports are only needed in biconnected variables.
Lemma 6.1 Let $G = (V, E)$ be a graph. Let $S \subseteq V$ be a set of connected vertices and $v \notin S$. Then $S' = \text{adj}(v) \cap S$ is biconnected in $G$.

Proof If $S'$ is empty or contains a single vertex, the assumption holds because, by definition, a set with at most 1 vertex is biconnected. Otherwise there are at least two vertices in $S'$. Let $a, b$ be two vertices in $S'$, see Figure 6.3. Then we can construct a cycle $(v, a, \ldots, b, v)$. The edges $\{v, a\}$ and $\{v, b\}$ exist and between $a$ and $b$ there is a path in $S$ because $S$ is connected. This proves that $\{v, a, b\}$ is biconnected. The construction is possible for each $a$ and $b$ in $S$. Hence, $S'$ is biconnected in $G$. \qed

Note that as soon as $S'$ contains at least two vertices, also the set $S' \cup \{v\}$ is biconnected.

The following theorem and corollary both require the use of prefix-connected orderings. This may seem to be a restriction for the choice of variable orderings. But in most cases it does not make sense to choose a variable which has no restriction to already instantiated ones. Thus, prefix-connected orderings are a natural choice.

Theorem 6.2 A CSP is strongly directional $k$-consistent with respect to a given ordering if the ordering is prefix-connected and the CSP is strongly directional $k$-block-consistent.

Proof To enable directional $k$-consistency we must show that each $l$-tuple, $l < k$, has a support in variables which succeed all variables in that $l$-tuple. Let $t$ be an $l$-tuple, $l < k$. Let $v$ be a variable succeeding $t$ in the given ordering. Because the ordering is prefix connected, the vertices in $t$ are connected. To prove a support for $t$ in $v$ it is sufficient
to show that $v$ supports those variables in $t$ adjacent to $v$. Now we use Lemma 6.1 to show that the intersection of the adjacency list of $v$ and $t$ are biconnected. Therefore, strong $k$-block-consistency guarantees that the adjacency list of $v$ in $t$—and therefore also the entire $t$—is supported by $v$.

Figure 6.4: *Block-consistency.*

Theorem 6.2 holds if the variable ordering is known in advance and fixed during search. However, if we want to allow a dynamical ordering (for instance for back-jumping [Dec90]), more consistency is needed. This is shown in the following corollary.

**Corollary 6.3** A CSP is strongly directional $k$-consistent with respect to each prefix-connected ordering if it is strongly $k$-block-consistent.

**Proof** Strong $k$-block-consistency requires that each $l$-tuple, $l < k$, of biconnected variables has a support in each other variable $v$. However, the intersection of the adjacency list of each $v$ and the already instantiated variables with respect to each prefix-connected ordering are biconnected (Lemma 6.1). This guarantees the needed support for all value combinations in all prefix-connected orderings.

The vertex numbering in Figure 6.4 determines a variable ordering which is prefix-connected. Making this graph strongly directional path-consistent with respect to the ordering can be reduced to the application of strong directional 3-block-consistency. This produce additional restrictions at most on the variable pairs $(1, 4)$, $(2, 3)$, $(4, 8)$, $(6, 7)$, $(10, 11)$. All pairs which are not biconnected (for instance $(1, 10)$, $(2, 8)$, etc.) do not have to be considered. Furthermore, the blocks
built by the vertices 5, 13, 14, 15, 16 need only be made directional arc-consistent. The edges \{4, 5\}, \{8, 9\}, \{12, 13\}, \{12, 16\}, \{13, 14\}, and \{13, 15\} will never be changed because they are bridges.

### 6.3.2 Strong 3-Consistency

In the previous section, it has been shown that directional \( k \)-block-consistency is sufficient for directional \( k \)-consistency under the condition that prefix-connected orderings are used. In the following, we show that strong 3-block-consistency can easily be extended to strong 3-consistency. The additional constraints are simply joins along the shortest paths between the variables.

For 3-consistency each allowed assignment of each pair of variables needs to be supported. This support is constructed with the help of supports in intermediate variables with certain properties. In the following two lemmata, the existence of such variables is shown.

**Lemma 6.2** Let \( G = (V, E) \) be an undirected graph. Let \( i, j, k \in V \) be vertices. Assume that \( i \) and \( j \) are biconnected and that \( k \) is connected but not biconnected to both \( i \) and \( j \). Let \( P_i \) be a simple path \( i \Rightarrow k \) and \( P_j \) be a simple path \( j \Rightarrow k \). Then there exists a vertex biconnected to \( i \) and \( j \) which is on \( P_i \) and \( P_j \).

**Proof** Let \( v \) be a vertex on \( P_i \) and \( P_j \) and there is no vertex with the same property in front of \( v \) in either path, see Figure 6.5. If \( v = i \) or \( v = j \) we are finished because \( i \) and \( j \) are biconnected. Now consider the path \( P' \) constructed as follows: From \( i \) to \( v \) follow the path \( P_i \),
then follow the path $P_j$ backward to $j$. This path is simple, otherwise $v$
would not be the vertex with the smallest index in $P_i$ and $P_j$. Between
$i$ and $j$ there must be another path $P''$ which is vertex disjoint to $P'$.
Otherwise there would be a single separator vertex for $i$ and $j$. The
paths $P'$ and $P''$ together form a cycle including $i$, $j$, and $v$. Hence, $i$,
$j$, and $v$ are biconnected.

Figure 6.6: The situation for Lemma 6.3.

Note that the case $v = k$ is impossible. Otherwise $k$ and $i$ would be
biconnected, which contradicts our assumption. However, it might be
possible, that $v$ is equal to $i$ or $v$ is equal to $j$.

Consider the graph in Figure 6.4. Let $i$, $j$, and $k$ be 10, 12, and 2.
The common vertex on each paths $i \Rightarrow k$ and $j \Rightarrow k$ that is farthest
from $k$ is always the vertex 9. This vertex is biconnected to 10 and 12.

The next lemma is for the case of 3 vertices that are not biconnected.

**Lemma 6.3** Let $G = (V, E)$ be an undirected graph. Let $i, j, k \in V$ be
vertices that are connected but neither pair is biconnected. Let $P_1$ be a
simple path $i \Rightarrow j$, $P_2$ be a simple path $j \Rightarrow k$ and $P_3$ be a simple path
$k \Rightarrow i$. Then there are three vertices $u, v, w \in V$ which are biconnected
and $u$ lies on $P_1$ and $P_3$, $v$ lies on $P_1$ and $P_2$, and $w$ lies on $P_2$ and $P_3$.

**Proof** Let $u$ be a vertex that is on $P_1$ and $P_2$ and there is no vertex
farther from $i$ with the same property. Let $v$ be a vertex on $P_2$ and $P_3$
and there is no vertex farther from $j$ with the same property. Let $w$ be
a vertex on $P_3$ and $P_1$ and there is no vertex farther from $k$ with the
same property. Now a path can be constructed from $u$ following $P_1$ to $v$, following $P_2$ to $w$ and following $P_3$ back to $u$. This closed path is a cycle. Otherwise $u$, $v$, $w$ or would not be the farthest common vertices. Hence, $u$, $v$, and $w$ are biconnected.

Note that this lemma holds even if some pair of the vertices $u$, $v$, $w$ is equal. Consider the graph in Figure 6.4. Let $i$, $j$, and $k$ be the vertices 1, 6, and 11, respectively. Assume that $P_1$ is $(1, 3, 4, 7, 8, 9, 11)$, $P_2$ is $(11, 9, 8, 6)$ and $P_3$ is $(6, 4, 2, 1)$. The vertex common in $P_1$ and $P_2$ farthest from 1 is 4, the common vertex in $P_2$ and $P_3$ farthest from 11 is 8, and the common vertex on $P_3$ and $P_1$ farthest from 6 is 6. These vertices—4, 8, and 6—are biconnected.

In addition to the structural properties of the graph we need another Lemma on the join over shortest paths between biconnected vertices in a strongly 3-block-consistent CSP. Usually, there is an edge between those vertices because there is a real restriction. But it might be the case that the relation is universal. Then the join over the shortest path between the two vertices is also a universal relation.

**Lemma 6.4** Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP which is strongly 3-block-consistent. Let $i, j \in \mathcal{X}$ be biconnected. Let $P$ be the shortest path between $i$ and $j$. Then $R_{ij} = |x|_P$.

**Proof** Proof by induction over the length of the shortest path.

$|P| = 1$ If the shortest path is $(i, j)$, then $C_{ij} \in \mathcal{C}$ and we are finished.

$|P| > 1$ Let $P = (i, \ldots, k, \ldots, j)$. Let $P_1$ and $P_2$ be the sub-paths of $P$ from $i$ to $k$ and from $k$ to $j$, respectively. Note that $P_1$ is a shortest path between $i$ and $k$ and $P_2$ is a shortest path between $k$ and $j$. $\mathcal{P}$ is 3-block-consistent and $i, j$ are biconnected. Hence, $R_{ij} \subseteq R_{ik} \times R_{kj}$. But, $C_{ij} \notin \mathcal{C}$ and $R_{ij}$ is universal. Therefore, $R_{ij} = R_{ik} \times R_{kj}$. By induction, $R_{ik} |x|_P R_{kj} = |x|_{P_1} |x| |x|_{P_2} = |x|_P$. Thus, the desired property holds for $P$.

Now we come to the main theorem. It gives a recipe how a strongly 3-consistent CSP can be constructed if a strongly 3-block-consistent CSP is given. This construction only involves adding new constraints. The
domains and the existing constraints are left unchanged. The computation is efficient such that it is not necessary to aim for full consistency at the beginning.

Theorem 6.4 Let $\mathcal{P} = (X, D, C)$ be a CSP which is strongly 3-block-consistent. Construct $\mathcal{P}^* = (X^*, D^*, C^*)$ as follows:

- $X^* = X$
- $D^* = D$
- $R^*_{ij} = \begin{cases} D_i \times D_j : & i, j \text{ isolated in } \mathcal{P} \\ R_{ij} : & i, j \text{ biconnected in } \mathcal{P} \\ |x|_P, P \text{ shortest } i \Rightarrow j \text{ in } \mathcal{P} : & \text{ otherwise} \end{cases}$
- $C^* = \{C_{ij} \mid R^*_{ij} \neq D_i \times D_j\}$

$\mathcal{P}^*$ is equivalent to $\mathcal{P}$ and strongly 3-consistent.

Proof Three things need to be shown: that $\mathcal{P}^*$ is equivalent to $\mathcal{P}$ and that $\mathcal{P}^*$ is both arc-consistent and path-consistent.

Equivalence. By definition two CSPs are equivalent if they have the same solution set. In $\mathcal{P}^*$ only value combinations are forbidden compared to $\mathcal{P}$. Thus, $\text{sol}(\mathcal{P}^*) \subseteq \text{sol}(\mathcal{P})$. It remains to show that $\text{sol}(\mathcal{P}^*) \supseteq \text{sol}(\mathcal{P})$. In the following, we show a stronger property: $\forall i, j \mid R^*_{ij} \supseteq R_{ij}$.

Figure 6.7: Theorem 6.4: the cases for path-consistency.
Chapter 6. Biconnected Components

If \( i \) and \( j \) are isolated or biconnected, then \( R^{*}_{ij} = R_{ij} \). Otherwise \( i \) and \( j \) are connected but not biconnected. By construction \( R^{*}_{ij} = \varnothing \), where \( \mathcal{P} \) is the shortest path between \( i \) and \( j \). By definition of join each pair not in \( \varnothing \) is not supported by some variable on \( \mathcal{P} \). Hence, \( R^{*}_{ij} \) contains at least the allowed pairs for \( i \) and \( j \) in \( \mathcal{P} \). Consequently, \( R^{*}_{ij} \supseteq R_{ij} \) and \( \text{sol}(\mathcal{P}^{*}) \supseteq \text{sol}(\mathcal{P}) \).

**Arc-consistency.** We show that each value \( a \in D_{i}^{*} \) for each variable \( i \in \mathcal{X} \) has a support in each other variable of \( \mathcal{X} \). That is,

\[
\forall i, j \in \mathcal{X} \mid a \in D_{i}^{*} \rightarrow \exists b \in D_{j}^{*} \mid (a, b) \in R_{ij}^{*}.
\]

1. \( i, j \) are isolated in \( \mathcal{P} \)

In this case \( R^{*}_{ij} = D_{i} \times D_{j} \). Each \( b \in D_{j} \) is a support for each \( a \in D_{i} \).

2. \( i, j \) are connected.

By definition \( R^{*}_{ij} = \varnothing \), where \( \mathcal{P} = (i = v_{0}, v_{1}, \ldots, v_{p-1}, v_{p} = j) \) is the shortest path from \( i \) to \( j \) in \( \mathcal{P} \). Because \( \mathcal{P} \) is arc-consistent, there is a support \( a_{1} \in D_{v_{1}} \) for \( a \in D_{i} \). Consequently, there is a support \( a_{2} \in D_{v_{2}} \) for this \( a_{1} \in D_{1} \), and so on. Until there is a \( b \in D_{j} \) which is a support for \( a_{p-1} \in D_{p-1} \). Hence, by definition of join \( (a, b) \in R^{*}_{ij} \). Thus, \( b \) is a support for \( a \).

3. \( i, j \) are biconnected

Then \( R^{*}_{ij} = R_{ij} \). The fact that \( \mathcal{P} \) is 2-block-consistent ensures the supports.

**Path-consistency.** We need to show that for all pairs of variables \( i, j \) and all pairs of values \( (a, b) \in R_{ij}^{*} \), there is a support \( c \in D_{k}^{*} \) in each other variable \( k \).

\[
\forall i, j, k \in \mathcal{X} \mid (a, b) \in R_{ij}^{*} \rightarrow \exists c \in D_{k}^{*} \mid (a, c) \in R_{ik}^{*} \land (b, c) \in R_{jk}^{*}.
\]

The following cases can occur (all properties hold for \( \mathcal{P} \)):

1a. \( i, j \) isolated, \( i, k \) isolated, \( j, k \) isolated

\( R_{ik}^{*} \), and \( R_{jk}^{*} \) are universal. Hence, each pair of values \( (a, b) \in R_{ij}^{*} = D_{i} \times D_{j} \) is supported by each \( c \in D_{k} \).

1b. \( i, j \) isolated, \( i, k \) connected, \( \rightarrow j, k \) isolated

\( R_{jk}^{*} \) is universal. Therefore, the support for \( a \in D_{i} \)—which must exist, otherwise \( \mathcal{P}^{*} \) would not be arc-consistent—is also a support for \( (a, b) \in R_{ij}^{*} = D_{i} \times D_{j} \).
6.3. From BC to Consistency

1c $i, j$ isolated, $i, k$ biconnected, $\rightarrow j, k$ isolated
   analogous to case 1b.

2a $i, j$ biconnected, $i, k$ isolated, $\rightarrow j, k$ isolated
   Analogous to case 1a.

2b $i, j$ biconnected, $i, k$ connected, $j, k$ connected
   Let $P_1$ be the shortest path from $i$ to $k$, and $P_2$ be the shortest
   path from $j$ to $k$. There is a vertex $v$ on both paths $P_1$ and $P_2$
   which is biconnected to $i$ and $k$ (Lemma 6.2). Because $i$ and $j$ are
   biconnected, there must be support $a_v \in D_v$ with $(a, a_v) \in R_{iv}^* =
   R_{iv}^*$ and $(b, a_v) \in R_{jv}^* = R_{jv}^*$. Because $P^*$ is arc-consistent. This
   $a_v$ must have a support $c \in D_k$. The pair $(a, c)$ is in $R_{ij}^* = |x|_P$
   because $(a, a_v)$ is in $R_{iv}^*$ and $(a_v, c)$ is supported by the join
   over the sub-path of $P_1$ between $v$ to $k$. Analogous $(b, c) \in R_{jk}^*$.

2c $i, j$ biconnected, $i, k$ biconnected, $j, k$ connected
   The vertices $j$ and $k$ are not biconnected, but vertex $i$ is biconnected
   to both. Thus, $i$ is an articulation point and lies on each
   path between $j$ to $k$, also on the shortest one. Due to the fact that
   $P$ is 3-block-consistent, each $(a, b) \in R_{ij}$ has a support $c \in D_k$
   such that $(a, c) \in R_{ik}^* = R_{ik}^*$. The join over the shortest path from
   $j$ to $k$ equals $R_{ji}^* \times |x| R_{ik}^*$ (Lemma 6.4). Hence, $(b, c)$ is also in $R_{jk}^*$
   and, subsequently, $c$ supports $(a, b)$.

2d $i, j$ biconnected, $i, k$ biconnected, $j, k$ biconnected
   $R_{ij}^*, R_{ik}^*, R_{jk}^*$, and $R_{jk}^*$ equal $R_{ij}^*$ and $i, j$ are biconnected.
   Hence, strong 3-block-consistency ensures the needed supports.

3a $i, j$ connected, $i, k$ isolated, $\rightarrow j, k$ isolated
   Analogous to case 1a.

3b $i, j$ connected, $i, k$ connected, $j, k$ connected
   Let $u, v, w$ the biconnected vertices constructed as in Lemma 6.3,
   $u$ on the shortest path $i \to j$, $v$ on the shortest path $i \to k$ and $w$
   on the shortest path $j \to k$. Given $(a, b) \in R_{ij}^* = |x|_P$. Because $u$
   and $v$ lie on $P$, there is an $a_u \in D_u^*$ and an $a_v \in D_v^*$. The vertices
   $u$ and $v$ are biconnected. Hence, $(a_u, a_v) \in R_{uv}^*$. Furthermore,
   $(a, a_u) \in |x|_{i \to u}$ and $(a_v, b) \in |x|_{v \to j}$. The vertices $u, v, w$ are
   pairwise biconnected. Hence, there is a support $a_w \in D_w^*$ for
   $(a_u, a_v) \in R_{uv}^*$. This implies that $(a_u, a_w) \in R_{uw}^* = R_{uw}$ and that
   $(a_v, a_w) \in R_{vw}^* = R_{vw}$. The value $a_w \in D_w$ has a support $c$ in
k because $\mathcal{P}^*$ is arc-consistent. Consequently, $(a_w, c) \in |x|_{w \rightarrow k}$. Now we assemble the existence of $(a, a_u) \in |x|_{i \rightarrow u}$, of $(a_u, a_w) \in R_{uw}$, and of $(a_w, c) \in |x|_{w \rightarrow k}$ to show that $(a, c) \in R^*_{ik}$. The analogous construction applies for $(b, c) \in R^*_{jk}$. Hence, $c$ is a support for $(a, b) \in R^*_{ij}$.

3c $i, j$ connected, $i, k$ biconnected, $j, k$ connected

The shortest path between $i$ and $j$ and the one between $k$ and $i$ intersect in a vertex $v$ which is biconnected to both $i$ and $k$ (Lemma 6.2). Let $P$ be the shortest path between $v$ and $j$. By construction of $R^*_{ij}$ there must be at least one $a_v \in D_v$ which is a support for $(a, b) \in R^*_{ij}$. Then $(a, a_v) \in R^*_{iv}$ and $(a_v, b) \in |x|_P$. The vertices $k$ and $v$ are biconnected and $\mathcal{P}^*$ is arc-consistent. Hence, $(a_v, c) \in R_{vk}$. Now from $(a, a_v) \in R_{iv}$ and $(a_v, c) \in R_{vk}$ follows $(a, c) \in R^*_{ik} = R^*_{ik}$ and of $(a_v, b) \in |x|_P$ and $(a_v, c) \in R^*_{vk}$ follows $(b, c) \in R^*_{jk}$.

3d $i, j$ connected, $i, k$ biconnected, $j, k$ biconnected

The vertex $k$ must be a separator for $i$ and $j$. Hence, the shortest paths from $i$ to $j$ runs over $k$ and the part from $i$ to $k$ is also a shortest path between $i$ and $k$. Analogously for $k$ and $j$. By using Lemma 6.4, we have $R^*_{ij} = R^*_{ik} |x|_P R^*_{kj}$. Hence, each $(a, b) \in R^*_{ij}$ has a support in $k$.

This theorem gives us the opportunity to reduce the computation in the assembly problem to strong 3-block-consistency. First of all the two CSPs are equivalent. That is, they have the same solution set. Thus, the newly inserted edges are redundant. Even more, not only is each edge redundant, but they are all simultaneously redundant.

**Corollary 6.5** Let $\mathcal{P}$ and $\mathcal{P}^*$ be as in Theorem 6.4. All constraint in $\mathcal{P}^*$ that are not in $\mathcal{P}$ are simultaneously redundant.

**Proof** The removal of the edges added to $\mathcal{P}^*$ leads to $\mathcal{P}$. $\mathcal{P}$ and $\mathcal{P}^*$ are equivalent. Thus, all edges are redundant. □

$\mathcal{P}$ contains fewer constraints and the initial block structure is preserved. Thus, in the assembly, where we aim at having a clear graph
structure such that the source of redundancy, degeneracy etc. can be located, it is much more convenient to use block-consistency instead of consistency. However, some restrictions are not explicitly available. But they can be computed on the fly by using the construction of Theorem 6.4.

In the following, we make some additional observations important to the ROMA-solver. First, we observe that the domain in $\mathcal{P}$ and $\mathcal{P}^*$ are the same.

**Observation 6.1** Let $\mathcal{P}$ and $\mathcal{P}^*$ be as in Theorem 6.4. $D^* = D$.

Hence, after having enforced strong 3-block-consistency, no value is ever removed from the domains during the enforcement of strong 3-consistency. This is important if the aim of 3-consistency enforcement is to prove that some value for a certain variable can never be part of any solutions. The answer can already be given after enforcement of strong 3-block-consistency.

Furthermore, we see that no relations in $\mathcal{P}^*$ become empty because either the relation are taken from $\mathcal{P}$ or they contain at least one pair of values for each value in the domains because of arc-consistency.

**Corollary 6.6** Let $\mathcal{P}$ and $\mathcal{P}^*$ be as in Theorem 6.4. If there is a $R_{ij}^* = \emptyset$ in $\mathcal{P}^*$, then $D_i = \emptyset \land D_j = \emptyset$.

**Proof** If $R_{ij}^* = \emptyset$ then $i$ has no support in $j$. Hence, $D_i = \emptyset \land D_j = \emptyset$. The domains in $\mathcal{P}$ and $\mathcal{P}^*$ are equal. Thus, $D_i = \emptyset$ in $\mathcal{P}$. $\square$

In general, consistency enforcement aims at searching local inconsistencies. Corollary 6.6 tells us that while we make a strongly 3-block-consistent CSP strongly 3-consistent, there will not be any further inconsistencies between pairs of variables. Hence, 3-block-consistency is enough to prove local inconsistencies occurring during 3-consistency.

One of the main reasons to enforce consistency in the assembly is to find rigidity structures and compute the mobility among them. In Chapter 7, we shall see that in this case: we can reformulate the CSP to find all rigid relations directly. Rigidity corresponds to bijective constraints. Corollary 6.7 tells us that we need not to expect more rigid structures after having applied 3-block-consistency. Of course there might be more relations which get bijective. But the skeleton of them is already present in the 3-block-consistent CSP.
Corollary 6.7 Let $P$ and $P^*$ be as in Theorem 6.4. Let $i$ and $j$ be connected. If $R^*_{ij}$ is bijective, there is at least one $R^*_{uv}$ on the shortest path from $i$ to $j$ that is functional.

**Proof** If $i$ and $j$ are biconnected, then $R^*_{ij} = R_{ij}$ and the assumption holds because of Lemma 6.4.

Now assume that $i$ and $j$ are connected but not biconnected. Let $v$ be a vertex adjacent to $j$ on the shortest path between $i$ and $j$. Let $(a, b) \in R^*_{ij}$. Note that there is no other $b'$ with $(a, b') \in R^*_{ij}$. Now consider the set $S$ of all supports of $a$ in $R^*_{iv}$. The fact that $R^*_{ij}$ is bijective implies that for all $a_v \in S$ there is no other $b' \in D_j$ such that $(a_v, b') \in R^*_{aj}$. In particular, this means that each $a_v \in D_v$ has at most one support in $R^*_{uj}$. Hence, $R_{vuj}$ is functional.

If $i$ and $j$ are isolated, then $R^*_{ij} = D_i \times D_j$. Hence, $R^*_{ij}$ is only bijective if $|D_i| = |D_j| = 1$.

In assembly problems functional constraints are always also bijective due to symmetry. Hence, we follow that there must be at least a bijective constraint on the shortest path between $i$ and $j$. Subsequently, not only the constraint $R_{uj}$ is bijective but also the join over the path from $i$ to $u$. Thus, we can apply Corollary 6.7 again. Hence, the skeleton of rigidity is already computed after the enforcement of strong 3-block-consistency.

### 6.3.3 BC-Enforcement

In the previous section, we have shown that 3-block-consistency is enough to compute certain properties of a CSP, such as value removal of domains, erasure of relations, and the occurrence of bijective constraints in assembly problems. In this section, we discuss how to compute block-consistency efficiently, now that we know that it is sufficient.

A CSP is block-consistent if each biconnected $(k - 1)$-tuple of values has at least one support in every other variable. This suggests that an algorithm for block-consistency enforcement needs to check each $(k - 1)$-tuple against each other variable.

But this is not necessary. The algorithm only needs to check each $(k - 1)$-tuple against each variable which is biconnected to each of the variables of the $(k - 1)$-tuple. The only additional case is to enforce arc-consistency for bridges. This is shown in the following theorem.
Lemma 6.5 Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. If each block-induced sub-problem of $\mathcal{P}$ is strongly $k$-consistent, and if each bridge is arc-consistent, then $\mathcal{P}$ is strongly $k$-block-consistent.

Proof Let $t$ be an $l$-tuple, $l < k$, such that each pair of vertices in $t$ is biconnected. Let $v$ be another vertex not in $t$. If $v$ is biconnected to $t$ then there is a support for $t$ in $v$ because $v$ and $t$ belong to the same induced subgraph. Assume that $v$ is not biconnected to each vertex $t$. But as shown in Lemma 6.1 the intersection of a connected set and a vertex $v$ are biconnected. Therefore, the adjacency list is either empty or consists of a single vertex. If it is empty, we always have support because each relation is universal. If it contains a single vertex, say $w$, then $\{v, w\}$ must be a bridge. Then each allowed assignment of $t$ is supported by the value of $D_v$ that supports the value of $w$ in the assignment.

Hence, $k$-block-consistency is obtained by enforcing $k$-consistency on the block-induced subgraphs and arc-consistency on the bridges. In the ROMA-solver this is ensured by requiring that only triples $(i, j, k)$ are considered where $(i, j)$ are biconnected and $C_{ij}$ and $C_{ik}$ exist. Then it follows that $(i, j, k)$ are biconnected.

The block-induced sub-problems can be solved almost independently. Only the articulation points and the bridges of the graph are affected by more than one sub-problem. This suggests a strategy for parallelization of consistency enforcement.

6.4 Extending the ROMA-Solver

In the ROMA-solver we enforce only strong 3-block-consistency. Thus, after solving we can instantiate all graphs of at most width 2 without backtracking as shown in Algorithm 5.7.

Two things need to be done to adapt the ROMA-solver. First, we need to change the routine to compute the triples to be reconsidered when a constraint is updated, see Algorithm 6.1. There only biconnected triples are put into the queue, see Line 5 and 6. In the new enforcement routine, see Algorithm 6.2, RomaBlockRefresh is called instead of the old version, Line 9. Second, the initial queue only contains biconnected triples, see Line 5.
ROMA-BlockRefresh

1. procedure RomaBlockRefresh(i, k, Q)
2. // preconditions: $C_{ik}$ was changed
3. // put into Q all triples that need to be reconsidered
4. begin
5. $Q \leftarrow Q \cup \{(i, k, m) | i, k, m \text{ biconn, } C_{km} \in C, m \neq i\}$
6. $Q \leftarrow Q \cup \{(m, i, k) | m, i, k \text{ biconn, } C_{im} \in C, m \neq k\}$
7. end

Algorithm 6.1

ROMA-Enforce+

1. procedure RomaEnforce+
2. // precondition: Add has been applied on each constraint
3. // enforce strong 3-block-consistency by using Add & Cat
4. begin
5. $Q \leftarrow \{(i, j, k) | i, j, k \text{ biconnected, } C_{ij}, C_{jk} \in C\}$
6. while $Q \neq \emptyset$ do
7. $(i, j, k) \leftarrow \text{pop}(Q)$
8. if RomaRevise($i, j, k$) then
9. RomaBlockRefresh($i, k, Q$)
10. end if
11. end while
12. end

Algorithm 6.2
The restriction to biconnected triples is enough as demonstrated by the following points.

- Strong directional $k$-block-consistency is sufficient for strong directional $k$-consistency if a prefix-connected ordering is chosen, see Theorem 6.2. Hence, there is a backtrack-free search ordering if the width of the graph is at most $k-1$. The ROMA-solver does not impose restrictions on the chosen variable ordering. Thus, a prefix-connected might as well be used in Algorithm 5.7.

- Block-consistency requires that each biconnected $(k-1)$-tuples has a support in each other variables. However, because of Lemma 6.5 only triples $(i, j, k)$ are considered where $j$ is also biconnected to $i$ and $k$. In addition, the bridges must be made arc-consistent. But since values are never removed from domains, the call of Add at insertion time of a constraint suffices to ensure arc-consistency at the end. Hence, bridges are never touched again during solving.

- The aim of consistency enforcement in assemblies is to detect rigidity, illegality, and redundancy. After having enforced strong 3-block-consistency this information is already computed as shown in Observation 6.1, Corollary 6.6 and Corollary 6.7.

- The block structure is not changed during BC-enforcement. Hence, the influences stored in the constraints can only come from constraints of the same block. This restricts the possible source of information to the block level.

- The mobility between a pair of non-biconnected variables is not directly available after enforcement of 3-BC. However, the mobility can easily be computed as the join along the shortest path between these variables, see Theorem 6.4. The join is computed by applying Cat iteratively.

Assembly problems often contain a hierarchical structure which is reflected by the fact that they consist of several blocks. In such problems expensive calls of Cat are saved due to block-consistency. This fact is illustrated in the examples in Section 8.1.
6.5 Summary

In this chapter, we have presented block-consistency. This new kind of consistency was motivated by the fact that assembly constraints that do not share a common loop need not to be considered. The derived techniques, however, hold for general CSPs.

Block-consistency is useful in many ways. Directional block consistency is sufficient for directional consistency if prefix-connected orderings are used. Such orderings are a natural choice in CSPs and not really a restriction. Moreover, BC-enforcement preserves the block structure such that each block-based observation remains invariant during BC-enforcement. In particular, redundancy and degeneracy information can now be located on the block level.

During enforcement of block-consistency fewer computations are made and the resulting CSP is less explicit. Nevertheless, redundancy, inconsistency, and rigidity are already computed by enforcing 3-block-consistency. Hence, the ROMA-solver can be adapted to compute only block-consistency without disadvantage.

In the following chapter, we present a new approach to deal with rigid constraints. This approach helps in breaking the constraint graph into small blocks. This effect is exploited by block-consistency.
Chapter 7

Rigid Constraints

In this chapter, we analyze simplifications of assemblies in the presence of rigidity. Intuitively two bodies without relative mobility can be treated as a single rigid body. This simplifies the problem. Rigid constraints in the assembly correspond to bijective constraint relations. In other words: to functional relations in both ways.

In the following, we analyze functional constraints in the context of general CSPs. Hence, the results obtained are also valid in other applications. However, as we shall see, they are especially useful in geometric CSPs.

### 7.1 Introduction

Solving a problem means to reformulate the problem description until a representation is reached which is considered a solution. Consistency enforcement is a technique to reformulate problems into equivalent ones with more explicit information. A solution is obtained as soon as the level of consistency exceeds the width of the graph, see Theorem 5.1. Then an allowed assignment can be found in linear time.

Consider the graph of the 5-axes machine in Figure 7.1 on the left. It consists of a loop with 7 vertices and 7 constraints. The width of the graph is 2. The graph on the right consists of the same vertices and 6 constraints, its width is 1. If we can reformulate the problem such that the graph on the right is obtained, then arc-consistency is enough to
find a solution without backtrack. During arc-consistency enforcement no new constraints are added such that the width remains 1. Thus, the graph on the right is considered a solution of the graph on the left.

Rigidity between bodies can be used to reformulate the graph. In fact, it is not rigidity but functionality which allows to make edges redundant and remove them. Actually, a CSP with functional constraints can be solved like an arbitrary CSP. The normal consistency enforcement procedure can be applied. However, if we do that we give away advantages which could be exploited.

Consider for instance the graph of the saber saw—discussed in Section 8.1.3—in Figure 7.2. On the left side there is the initial graph, which contains some functional constraints, namely the 3 Identical (I) joints and the 1 driving Revolute (DR). When we enforce con-
7.1. Introduction

Consistency with RomaEnforce, Algorithm 5.5, the graph in the middle is obtained. This graph is complete and has a width of 11. The level of consistency in the graph is 3. Hence, we do not expect to find allowed assignments directly.

However, if we analyze the strongly 3-consistent CSP, then we note that some constraints are redundant. More precisely, all constraints are rigid. A redundant constraint can be removed from the problem without changing the solution. If we do that, then we obtain the graph on the right side in Figure 7.2. This graph has a width of 1. Therefore, arc-consistency is sufficient for global consistency. Because the CSP was strongly 3-consistent and we did not remove any values of domains, the CSP obtained is still arc-consistent. Hence, the problem is solved. Consequently, the problem in the middle is also already solved, but we did not detect it in the first place.

What can be done to get the graph on the right side instead of the graph in the middle? It is not a good approach to enforce arc- and path-consistency and detect redundancies afterwards. The detection of redundancies involves computing the so-called network-induced constraint, that is to compute the solutions of the graph without the constraint. This is as difficult as solving the problem itself.

In the following, we analyze how to obtain the graph on the right side in an efficient way. Our approach is based on detecting redundancies on the fly. In fact, in presence of a functional constraint we can remove redundant constraints directly to reduce the degree of vertices. If we enforce block-consistency instead of consistency, we ensure that an isolated vertex remains isolated because it is not biconnected to other vertices.

This chapter is organized as follows. First, related work is presented in Section 7.2. Then in Section 7.3, basic definitions and notations are given. In Section 7.4 we review the approach of DAVID to deal with functional constraints and modify to delete redundant edges. In Section 7.5, we show how block-consistency and constraint removal elegantly work together. In Section 7.6, we analyze how the reformulation can be implemented efficiently for geometric constraints using a new extension of the Union/Find data structure. The ROMA-solver is extended in Section 7.7.
7.2 Related Work

Usually, in constraint satisfaction theory consistency enforcement is used to get more explicit knowledge on the problem. Value combinations that cannot be extended to solutions are deleted. This makes the solution space smaller. The drawback of this approach is that the width of the graph may be increased because new (redundant) constraints are inserted.

In contrast to this Dechter and Dechter claimed in [DD87] that removing redundancy from the graph can make the problem easier by eventually being able to delete edges to obtain a simpler graph structure. They indicated an approach and some examples in which redundancy removal leads to deletion of constraints. However, they did not provide an exact scheme nor conditions by which the success of this procedure can be estimated.

Functional and bijective constraints have first been treated in CSP theory by van Hentenryck, Deville, and Teng in [VDT92]. They proposed a generic arc-consistency algorithm (AC-5), which more efficiently solves CSPs that entirely consist of particular constraint types. The types include functional, bijective, anti-functional\(^1\), and monotonic\(^2\) constraints. Then only time \(O(dm)\) instead of \(O(d^2m)\) is used to make a finite CSP arc-consistent, see Table 5.3.

Tractable constraint types have been characterized by Cooper, Cohen, and Jeavons in [CCJ94]. They found that problems composed of a special class of constraints, 0/1/all constraints, can be treated more efficiently. A special algorithm, ZOA, solves such CSPs in polynomial time. The authors also show that 0/1/all constraints are in fact the only class of constraints which can be handled in polynomial time.

In [Dav93] David pointed out that not only consistency enforcement is improved by functional constraints, but that also solutions can be generated more efficiently. In addition, his approach is applicable to CSPs where only some constraints are functional. For that he defines a so-called root set, a subset of the variables such that all other variables are directly or indirectly functionally dependent of the root set. If arc- and path-consistency are enforced in the graph, each variable outside the root set is directly determined by some variable in the root set.

\(^1\)A relation is anti-functional if its complement is functional.
\(^2\)A relation is monotonic if there is an ordering for the values such that if \((a, b) \in R\) then \((a', b') \in R\) for all \(a' \geq a\) and \(b' \geq b\).
Afterwards it can be checked in polynomial time, whether or not a solution for the root set can be extended to a global solution.

In [Dav95] David extended his approach by defining pivot consistency, a weaker kind of path-consistency. Instead of applying strong path-consistency only pivot-consistency is needed to exploit functional constraints. Unfortunately, the overall solving strategy is also determined, because the time bounds are only obtained if a search can be performed on the root set. Furthermore, pivot consistency is defined with respect to a given ordering of the variables. Finally, the approach with pivot consistency includes computing the root set a priori. Thus, it does not consider bijectivity which is visible only after some solving steps. This, however, happens often in geometric CSPs.

In this chapter, we modify the approach of David to enforce consistency more efficiently in CSPs with functional constraints. Our approach deals with dynamically occurring functionality and leads to CSPs with a decreased width because constraints are removed.

### 7.3 Definitions and Notation

In general, we repeat the definitions of [Dav95]. A constraint $C_{ij}$ is *functional* if for each $a \in D_i$ there is at most one $b \in D_j$ such that $(a, b) \in R_{ij}$, written $i \rightsquigarrow j$. If a constraint $C_{ij}$ is functional, then

$$\forall a \in D_i, b, b' \in D_j : (a, b) \in R_{ij} \land (a, b') \in R_{ij} \Rightarrow b = b'.$$

Hence, we can find a (partial) function $f_{ij}, D_i \rightarrow D_j$ which corresponds to $R_{ij}$. That is, $f_{ij}(a) = b \iff (a, b) \in R_{ij}$. Note that the inverse of $f_{ij}$ need not necessarily exist since $f_{ij}$ may be non-injective. A constraint $C_{ij}$ is *bijective* if $C_{ij}$ and $C_{ji}$ are functional.

From now on we use arrows in constraint graph picture to indicate functional constraints. The arrow goes from the depending variable and points to dependent. Bijective constraints are indicated with double arrows. In Figure 7.3, the constraints $C_{12}, C_{14}, C_{15}, C_{23}$, and $C_{56}$ are functional.

A vertex $w$ is a *descendant* [Ber70] of a vertex $v$ if there is a path $(v = v_0, v_1, v_2, \ldots, v_p = w)$ where $v_{i-1} \rightsquigarrow v_i$ for all $i$. A *root set* is a set $\mathcal{R} \subseteq V$ such that every vertex not in $\mathcal{R}$ is a descendant of some vertex in $\mathcal{R}$. A root set is *minimal* if no subset is also a root set. A root set is called *minimum root set* if there is no smaller root set. For instance,
for the graph in Figure 7.3 \( \{1, 5\} \) is a root set. However, it is not a minimal root set because \( \{1\} \) is also a root set. This set is also the only minimum root set.

An ordering is \( \mathcal{R} \)-compatible if the root set is a prefix of the ordering and each dependent variable is preceded by at most one variable of which it is dependent. In the example of Figure 7.3 the ordering \((1, 3, 2, 4, 6, 5)\) is not \(\mathcal{R}\)-compatible because 3 is not determined by a variable preceding it in the ordering. The ordering \((1, 2, 3, 4, 5, 6)\) is \(\mathcal{R}\)-compatible.

Two constraints \( C_{ik} \) and \( C_{jk} \) are \( x_k \)-compatible\(^3 \) if each value pair in \( R_{ij} \) has at least one support in \( D_k \) for \( C_{ik} \) and \( C_{jk} \). A functional constraint \( C_{ik} \) is a pivot of \( Y \subseteq V \) if for all \( x_j \in Y \) such that \( C_{jk} \in \mathcal{C} \), \( C_{ik} \) and \( C_{jk} \) are \( x_k \)-compatible.

### 7.4 Functional Constraints

In this section, we show how to detect redundancy in presence of a functional constraint on the fly. This is done by modifying the approach of DAVID. In the following, we give a more detailed description of his scheme.

---

\(^3\)With \( x_k \) DAVID denotes the variable \( k \). We did not change the term \( x_k \)-compatible to be consistent with his notation.
7.4.1 Pivot Consistency

David's approach is based on the use of minimum root set. In a first step each variable is made directly dependent of some variable inside the root set by subsequently shortening the path between the depending variable and the dependent one. This process is called pivot consistency. Then solutions are searched in the root set, each of which can be extended to exactly one solution for the entire CSP. More detailed, the solving scheme is divided into four phases:

1. Compute a minimum root set $\mathcal{R}$ of size $r$ and a $\mathcal{R}$-compatible assignment ordering. The root set can be found in time $O(n + m)$, the assignment ordering is constructed also in time $O(n + m)$.

2. Enforce pivot consistency in the CSP as follows. The variables not in the root set are processed from back to front along the $x_k$-compatible ordering. For each those variable a variable $i$ is chosen of which $j$ is dependent. Then for each $k$ in the adjacency list of $j$, $C_{kj}$, and $C_{ij}$ are made $x_k$-compatible with Compatible, see Algorithm 7.1. The entire procedure needs $O((n^2 - r^2)d^2)$ time.

3. Search for solutions in $\mathcal{R}$. This is done in time $O(m_Rd^r)$ time, where $m_R$ is the number of constraints in the root set.

4. Instantiate $\mathcal{R}$ and extend the solution to a global solution. The instantiation of $\mathcal{R}$ needs $O(r)$ time, the extension is done in $O(n - r)$. Hence, the instantiation can be computed in $O(n)$ time.

For pivot consistency the procedure Compatible is used, see Algorithm 7.1. This procedure makes $C_{ij}$ and $C_{kj}$ $x_k$-compatible. This is identical to making $C_{ij}$ path-consistent with respect to the path $(i, j, k)$ under the condition that $C_{ij}$ is functional.

Observation 7.1 After Compatible$(i, j, k)$, $C_{ik}$ is path-consistent with respect to the path $(i, j, k)$.

Proof The constraint $C_{ij}$ is path-redundant with respect to the path $(i, j, k)$ if and only if

$$\forall (a, c) \in R_{ik} : \exists b \in D_j : (a, b) \in R_{ij} \land (b, c) \in R_{jk}$$

This corresponds exactly to the check in Line 7 which is done for each $(a, b) \in R_{ij}$. \qed
Compatible [Dav95]

(1) procedure Compatible(i, j, k)
(2) \hspace{1cm} // precondition: i \sim j
(3) \hspace{1cm} // make C_{ij} and C_{kj} \iff k-compatible
(4) begin
(5) \hspace{1cm} forall (a,c) \in R_{ik} do
(6) \hspace{2.5cm} b <- f_{ij}(a)
(7) \hspace{2.5cm} if (a, b) \notin R_{ij} or (b, c) \notin R_{jk} then
(8) \hspace{4cm} R_{ik} <- R_{ik} - \{(a, c)\}
(9) \hspace{2.5cm} end if
(10) \hspace{1cm} end forall
(11) end

Algorithm 7.1

After pivot consistency is established, each variable outside the root set is directly dependent of one variable inside the root set. David shows that each solution of the root set can be extended to exactly one global solution. Thus, search is performed only on the root set.

In the world of assemblies search is not possible because the domains are uncountable. Hence, phase 3 of David’s approach cannot be performed. Furthermore, if we look at a typical assembly problem, only few constraints are rigid at the beginning. However, the result often consists of a single rigidity component. Consequently, rigidity is explicit only after some solving steps. Pivot consistency requires to compute the root set at the beginning. Therefore, dynamically occurring functionality cannot be handled.

Nevertheless, we want to benefit from the existence of rigidity. This can be done by observing that certain constraints are redundant.

7.4.2 Detecting Redundancy

The basic idea is to reformulate the constraints of a variable \( j \) in terms of an adjacent variable \( i \). This is possible if \( C_{ij} \) is functional. At the end we have a situation where \( j \) is only connected to \( i \) and \( i \) undertakes the role of \( j \) with respect to the adjacency list of \( j \).

First, the connection between path-consistency and path-redundancy is shown in presence of functional constraints.
Theorem 7.1 Let $\mathcal{P}$ be a CSP. Let $i, j, k$ be variables. If $C_{ik}$ is path-consistent with respect to the path $(i, j, k)$ and $C_{ij}$ is functional, then $C_{jk}$ is path-redundant with respect to $(j, i, k)$.

Proof This statement is equivalent to: if $C_{jk}$ is not path-redundant with respect to $(j, i, k)$ and $C_{ij}$ is functional, then $C_{ik}$ is not path-consistent with respect to the path $(i, j, k)$.

If $C_{jk}$ is not path-redundant with respect to $(j, i, k)$, then

$$\exists a, b, c : (b, c) \not\in R_{jk} \land (a, b) \in R_{ij} \land (a, c) \in R_{ik}.$$ 

In addition, $C_{ij}$ is functional. Hence, $(a, b) \in R_{ij}$ implies that $b = f_{ij}(a)$ and for all $b' \neq b$, $(a, b')$ is not in $R_{ij}$. Consequently, $b$ is the only possible support for $(a, c) \in R_{ik}$. Therefore, $(a, c) \in R_{ik}$ is not supported by $j$. Hence, $R_{ij}$ is not path-consistent with respect to $(i, j, k)$.

Note that this theorem holds independently of the existence of either $C_{kj}$ or $C_{kj}$.

As seen in Observation 7.1, the procedure $\text{Compatible}(i, j, k)$ makes the constraint $C_{jk}$ path-consistent with respect to the path $(i, j, k)$. Thus, we conclude that $C_{jk}$ is redundant after $\text{Compatible}(i, j, k)$.

Corollary 7.2 After $\text{Compatible}(i, j, k)$, $C_{jk}$ is redundant.

Proof In Observation 7.1, we have seen that after having called $\text{Compatible}(i, j, k)$, $C_{ik}$ is path-consistent with respect to $(i, j, k)$. Theorem 7.1 tells us that then $C_{jk}$ is path-redundant with respect to $(j, i, k)$. If a constraint is path-redundant then it is also redundant.

Whenever a constraint is redundant, it can be removed without changing the solution set. The condition under which the constraint $C_{kj}$ is redundant and, hence, removable, only depends on the fact that $C_{ij}$ is functional. Thus, we can remove $C_{jk}$ for all $k$ in the adjacency list of $j$. This is done by the procedure $\text{Isolate}$, Algorithm 7.2.

After the variable $j$ was processed by $\text{Isolate}$, it is still dependent of the variable $i$ which is in fact its only adjacent variable. Thus, the vertex $j$ is not biconnected to any other vertex.
Chapter 7. Rigid Constraints

Isolate

(1) procedure Isolate(i, j)
(2)   // precondition: i $\sim$ j
(3)   // make $C_{jk}$ redundant for all $k \in \text{adj}(j) - \{i\}$
(4) begin
(5)   forall $k \in \text{adj}(j) - \{i\}$ do
(6)      Compatible(i, j, k)
(7)      Remove $C_{jk}$
(8)   end forall
(9) end

Algorithm 7.2

Observation 7.2 After Isolate(i, j) no vertex is biconnected to j.

Proof After Isolate(i, j), j is adjacent to only one vertex: i. Therefore, there cannot exist two vertex disjoint path to another vertex. $\square$

After Isolate(i, j) the variable j needs not to be reconsidered in any triple. This is because the newly arising constraint could directly be reformulated again. The variable j does not play any role in further solving, except that arc-consistency needs to be ensured on $C_{ij}$.

The fact that j needs not to be reconsidered can either be stored somewhere in the data structure of j or j can temporarily be removed from the graph. Another method is to use block-consistency. Block-consistency elegantly assures that j is unimportant for the rest of the solving because it will never be part of a biconnected ($k - 1$)-tuple.

7.5 Isolating Block-Consistency

In the previous section, we have seen that redundancy occurs in presence of functional constraints. This can be used to isolate vertices. In this section, we show how this can be exploited in the ROMA-solver, where we enforce strong 3-block-consistency. The idea is to adapt the phases 2 and 3 of David's approach. These two phases are replaced by enforcement of strong 3-block-consistency in conjunction with Isolate as soon as functional constraints are spotted.
7.5. Isolating Block-Consistency

Figure 7.4: Subsequent isolation of vertices.

The isolation of vertices takes place as shown in Figure 7.4. Variable 2 is functionally dependent of variable 1, thus we call Isolate(1, 2) which reformulates the constraints $C_{23}$ into $C_{13}$ and $C_{25}$ into $C_{15}$. Then variable 4 is isolated by reformulating $C_{43}$ into a new constraint $C_{13}$ and $C_{45}$ into the existing $C_{15}$. Likewise, we can take any of the circular functional dependencies between the vertices 3, 5, and 6 to obtain the desired result.

As seen in the example, phase 1 of pivot-consistency can be entirely removed. We do not need a root set explicitly, nor do we need to have a $\mathcal{R}$-compatible assignment ordering. In Phase 2, the enforcement of pivot-consistency is automatically done during enforcement of 3-block-consistency. Isolate is applied as soon as a dependent variable is spotted. If we do this, we obtain the same situation as after enforcement of pivot consistency. The difference is that the dependent variables now only have one adjacent variable in the constraint graph. We call the entire procedure isolating block-consistency or IBC. The final version of the ROMA-solver algorithm works with IBC, see Algorithm 7.14.

To compare our approach with that of DAVID consider the example in Figure 7.5. The initial graph on the left contains 5 functional constraints. For pivot consistency we need to compute a minimum root set, in this case the set $\{1, 3\}$. Then we construct a $\mathcal{R}$-compatible assignment ordering: $(1, 3, 2, 4, 6, 5)$. After enforcement of pivot consistency the graph in the middle is obtained. If, instead, block-consistency in combination with Isolate is used, then we obtain the graph on the right side, which has a width of 1.

The reformulation can be done as soon as a functional constraint is spotted. However, the question is what happens if a variable is depen-
dent of more than one variable. Assume that a variable $k$ is dependent of two variables $i$ and $j$. If we chose to call $\text{Compatible}(i, k, j)$, the functionality of $C_{jk}$ vanishes during reformulation. The process is irreversible.

In addition, the choice can even influence the resulting width of the graph. Consider the graph in Figure 7.6. The variable 4 is dependent of 1 and 2. If we chose to call $\text{Isolate}(1, 4)$, then the graph in the middle is obtained. This graph has a width of 2. However, if we had chosen $\text{Isolate}(2, 4)$, then the result would have been the graph on the right, which has a width of 1. In this graph, only arc-consistency would be necessary to find solutions without backtrack.

Now do we need to analyze the graph when calling $\text{Isolate}$? Fortunately, no. There may be an immediate influence on the width of the graph after $\text{Isolate}$. However, there is no final influence of the decision.

Consider the following. Let $\mathcal{P}$ be a CSP in which strong 3-block-consistency shall be enforced. Let $i, j, k$, be three variables with $i \sim k$
7.6 Reformulating Geometric Constraints

and \( j \sim k \). Now we can chose to call Isolate\((i, k)\) or Isolate\((j, k)\). After either call \( C_{ij} \) is path-consistent with respect to the path \((i, k, j)\), see Observation 7.1. Now remove the variable \( k \) from the graph and enforce the intended consistency. At the end add the variable \( k \) again. Independently of the initial decision either \( C_{ik} \) or \( C_{jk} \) can be added because both constraints are functional and there is exactly one support in \( k \) for each value of either \( i \) or \( j \). After the insertion of either constraint \( C_{ij} \) is still path-consistent because this was already the case after the call of Isolate at the beginning. This implies that the decision which functionality to exploit was reversible. Hence, it is reasonable not to take care of it in Isolate.

The advantages of isolating block-consistency are the following:

- It reduces the number of constraints in the final graph and simplifies the block structure.
- We neither need to compute the minimum root set, nor do we need to construct a \( x_k \)-compatible assignment ordering.
- Functional constraints that occur after some enforcement steps are also taken care of. This is contrary to DAVID's approach, where the minimum root set is computed at the beginning.

7.6 Reformulating Geometric Constraints

In the previous sections, we have discussed constraint reformulation in the presence of functional/bijeccive constraints and its implications to the structure of the constraint graph. The reformulation was based on the routine Isolate which itself called the routine Compatible, see Algorithm 7.1. We have seen that a call of Compatible\((i, j, k)\) makes \( C_{ik} \) path-consistent with respect to the path \((i, j, k)\), see Observation 7.1. Hence, it can be seen as an efficient version of Revise in case that \( C_{ij} \) is functional.

In the ROMA-solver Revise is implemented by calling Add and Cat, see Algorithm 5.4. However, to the same extent as Revise can be simplified to Compatible, also RomaRevise, Algorithm 5.1, can be made more efficient. We observe that if \( C_{ij} \) is rigid. Then Cat\((i, j, k)\) returns \( C_{jk} \), where the geometric entities relative to \( j \) are rewritten relative to \( i \). Hence, we can save the call of Cat. However, we still need to
reformulate all geometric entities which are described locally to \( j \). This might be an expensive task possibly also leading to numerical problems. Fortunately, it can be avoided as demonstrated in the following.

### 7.6.1 Roma-Isolate

Geometric constraints in the assembly problem are defined by restricting positions and orientations of two geometric entities, attached to either body involved in the constraint. The coordinates of the entities are described locally with respect to the coordinate system of the component.

To rewrite a constraint \( C_{ik} \) to \( C_{jk} \) a transformation needs to be computed which transpose the coordinate system \( i \) to that of \( j \). This transformation must be applied to all geometric entities in \( C_{ik} \). Applying a 3D-transformation to a geometric entity consists of several additions and multiplications of reals and is expensive. Thus, we want to save as much operations as possible.

In the ROMA-solver we make use of the fact that reformulation is performed from \( i \) to \( j \) only if \( i \) and \( j \) are rigid with respect to each other. Therefore, we could simply use the constraint \( C_{jk} \) for \( C_{ik} \) if we assure that first the bodies are in a consistent relative location and that second whenever \( i \) is moved also \( j \) is moved. Then we do not need to apply the transformations but can just use the old values. The geometric entities are simply relative to their original coordinate system. However, this system is rigid towards the system to which the entity actually belongs.

This coupling of coordinate system is realized by an extension of the Union/Find data structure presented in Section 7.6.3. Then coupling the coordinate systems of \( i \) and \( j \) corresponds to calling \( \text{Union}(i, j) \).

In RomaIsolate first the constraint \( C_{ij} \) is satisfied. That is, the two coordinate systems of \( i \) and \( j \) are brought into consistent position. Then both systems are coupled by calling \( \text{Union}(i, j) \). The implementation of this procedure is explained in Section 7.6.3. After the call \( i \) and \( j \) act like a single body. The constraint \( C_{jk} \) can then be used as constraint between the variables \( i \) and \( k \). In RomaCompatible, \( C_{jk} \) is added to \( C_{ik} \) to make \( C_{jk} \) redundant, see Algorithm 7.4. Hence, \( C_{jk} \) can be removed of the constraint graph.

The refresh strategy must be adapted also, see Algorithm 7.5. Each time a constraint is changed, it must be checked for rigidity. If this
ROMA-Isolate

(1) procedure RomaIsolate\((i, j, RQ, Q)\)
(2) // precondition: \(i \sim j\)
(3) // make \(C_{jk}\) redundant for all \(k \in \text{adj}(j) - \{i\}\)
(4) begin
(5) Satisfy\((C_{ij})\)
(6) Union\((i, j)\)
(7) forall \(k \in \text{adj}(j) - \{i\}\) do
(8) RomaCompatible\((i, j, k)\)
(9) Remove \(C_{jk}\)
(10) RomaBlockRigidRefresh\((i, k, RQ, Q)\)
(11) end forall
(12) end

Algorithm 7.3

ROMA-Compatible

(1) procedure RomaCompatible\((i, j, k)\)
(2) // precondition: \(i \sim j\)
(3) // make \(C_{jk}\) redundant
(4) begin
(5) \(C_{ik} \leftarrow \text{Add}(C_{ik}, C_{jk})\)
(6) end

Algorithm 7.4
RomaRigidBlockRefresh

1. procedure RomaRigidBlockRefresh($i$, $k$, $RQ$, $Q$
2. // precondition: $C_{ik}$ was changed
3. // put $C_{ik}$ into $RQ$ if it is rigid
4. // put into $Q$ all triples that need to be reconsidered
5. begin
6. if IsRigid?($C_{ik}$) then
7. $RQ \leftarrow \{(i, k)\}$
8. else
9. $Q \leftarrow Q \cup \{(i, k, m) \mid i, k, m \text{ biconn}, C_{km} \in C, m \neq i\}$
10. $Q \leftarrow Q \cup \{(m, i, k) \mid m, i, k \text{ biconn}, C_{im} \in C, m \neq k\}$
11. end if
12. end

Algorithm 7.5

is the case, the rigid constraint is added into a special queue, $RQ$, handled in the ROMA-solver, see Algorithm 7.14. Hence, if a constraint becomes rigid, it is put into $RQ$, Line 7. Otherwise all paths of length 2 containing it are put into $Q$, Line 9 and 10.

Note that in Add and Cat transformations of coordinate systems are performed often. Thus, it is important that the algorithms to transform and to union coordinate systems are efficient. In the following, we show how this can be done. For that purpose we use an extension of the Union/Find data structure presented by Tarjan in [Tar75]. This structure only needs almost linear cost per operation.

7.6.2 Algorithms

The classical Union/Find problem is to maintain a collection of disjoint sets under the operations union (merge two sets) and find (get the set to which a certain element belongs). This problem has been widely studied and there is a well-known implementation which reaches the almost linear lower bound. There are several naive implementation for the Union/Find problem as well, each of which is optimized with respect to one operation. We refrain from reviewing those approaches and refer to [GI91]. The main idea was found by Tarjan [Tar75, Tar79] and uses
7.6. Reformulating Geometric Constraints

\[ O(n + m \alpha(m + n, n))^4 \] time for a sequence of at most \( n - 1 \) Union and \( m \) Find.

With the classical Union/Find we can already test whether two bodies belong to the same rigidity class by comparing their respective find values. The test is needed in the geometric solving routines Add and Cat. If the bodies lie in the different rigidity classes an additional constraint possibly leads to new restriction. Whereas a constraint must only be checked if the bodies are already rigid.

To check whether two bodies are rigid we need to add support for coordinate systems. Each body owns a coordinate system. In addition, we need to manipulate these coordinate system such that the rigidity is invariant. That is, whenever we move one body, the coordinates systems of all coupled bodies must be moved as well. Of course the current location of the system must also be available.

In the following, we analyze a data structure that meets the requirements mentioned. The structure can be used in a more general context than just to manage coordinate systems. It is only assumed that there are values attached to objects. The values can be manipulated and the objects can be coupled.

More formally, we assume that there is a group \( G = (S, \circ) \) with a set \( S \) and a group operation \( \circ \). In the assembly problem this group is built by the set of all Euclidean transformations in 3D-space \( SE(3) \). The location of a coordinate system is represented by the transformation from the inertial system. Furthermore, we assume that there is a set \( V \) of bodies, each of which owns a single value of \( G \).

The operations required are the following:

- **Union**\((A, B)\): couple the two disjoint sets \( A \) and \( B \) of bodies.
- **Find**\((i)\): find the set to which element \( i \) belongs. The name of the class is some identifier which satisfies the condition that \( \text{Find}(i) \) is equal to \( \text{Find}(j) \) if and only if they belong to the same rigidity class.
- **Transform**\((A, t)\): the value of all elements of set \( A \) is composed with the value \( t \in S \).
- **GetValue**\((i)\): return the current value of element \( i \in V \).

\(^4\alpha \) is the inverse of the Ackermann function and grows extremely slowly.
The value of a given vertex $i$ is given by taking the initial value of $i$ and combining it with all $t$ of $\text{Transform}(A, t)$ where $i$ belong to $A$ to the invocation time of Transform.

During assembly solving we do not need to retract a union because only more rigidity is detected. Therefore, the common extensions Split [IA84], De-union [MU86, WT89], or Backtrack [GIT88] are out of scope.

### 7.6.3 Data Structure and Algorithms

The data structure of each object consists of two fields, a pointer to the parent ($\text{parent}_i$) and the weight of the underlying subtree ($\text{weight}_i$). The parent pointer is used to find the path to the root, which is used to name the set. The weight is considered during uniting in order to ensure the logarithmic height of the tree. At the beginning each object is a tree for itself, initialized as $\text{parent}_i = i$ for each $i$.

In addition, we need a field $\text{val}_i$ used to compute the value of a body fulfilling the following invariant:

$$\text{val}(i) = \bigoplus_{j=\text{root}(i)}^{i} \text{val}_j$$

(7.1)

The extended implementation is similar to the classical Union/Find with the additional requirement that Invariant 7.1 is ensured.

First, some remarks on the classical part of the Union/Find. To obtain an efficient time complexity the tree heights must be small. To achieve that, two things are important:

- Use a union rule: either union by size making the root of the smaller tree the child of the root of the larger, or union by rank ([TV84]) which makes the root of the shallower tree point to the root of the deeper one.

- Use a compaction rule: either path compression (Algorithm 7.6, [HU73]), which makes all vertices on the way to the root a direct child of the root, path splitting (Algorithm 7.7, [VV77, Van80]), which makes each vertex on the path to the root the child of its grandparent, or path halving (Algorithm 7.8, [Van80]), which makes every second vertex the direct child of its grandparent.
Figure 7.7: Path compaction strategies: initial situation, path compression, path splitting, path halving.

---

PathCompression

1. procedure PathCompression(i)
2. // make i and its parents the direct child of their root
3. begin
4. if (i ≠ parent_{parent_i}) then
5. PathCompression(parent_i)
6. Up(i)
7. end if
8. end

Algorithm 7.6
Chapter 7. Rigid Constraints

---

**PathSplitting**

1. procedure PathSplitting(i)
2. // make i the child of its grandparent
3. // proceed recursively with the old parent
4. begin
5. if (i ≠ parent\_parent\_i) then
6. \( p \leftarrow parent\_i \)
7. \( \text{Up}(i) \)
8. PathSplitting(p)
9. end if
10. end

---

**Algorithm 7.7**

---

**PathHalving**

1. procedure PathHalving(i)
2. // make i the child of its grandparent
3. // proceed recursively with the new parent
4. begin
5. if (i ≠ parent\_parent\_i) then
6. \( \text{Up}(i) \)
7. PathHalving(parent\_i)
8. end if
9. end

---

**Algorithm 7.8**
7.6. Reformulating Geometric Constraints

---

Up

(1) procedure Up(i)
(2) // make i the child of its grandparent
(3) begin
(4) if (i ≠ parent_i) and (parent_i ≠ parent_parent_i) then
(5)  val_i ← val_parent_i ∙ val_i
(6)  weight_parent_i ← weight_parent_i − weight_i
(7)  parent_i ← parent_parent_i
(8)  end if
(9) end

---

Algorithm 7.9

If either union rule or either compaction rule is used, the optimal amortized costs are obtained [GI91].

Whenever we change the parent of a node by moving it upward in the tree, we must assure that Invariant 7.1 still holds. In the classical Union/Find there are some strategies how to shorten the path up to the root, each of which is optimal.

In order to simplify the implementation of the new features, we introduce a new routine called Up, see Algorithm 7.9. Up makes a vertex the direct child of its grandparent. The Invariant 7.1 is maintained by changing the value of the vertex i accordingly, see Line 5. With the routine Up all path compaction strategies can be implemented without further manipulation of the val field. Only in Union, where we make one vertex the child of another, we must take care that val_i of the new child is transformed by the inverse of the val of the root.

A GetValue(x) is reduced to a Find (after which x is the root or a direct child of its root) plus a constant number of operations. Path compaction is not necessary. But as we have to follow all pointer to the root and combine the groups elements anyway, it is efficient to compact the path during this procedure. We use the path compression technique in order to benefit as much as possible from already performed group operations, which are assumed to be more expensive than pointer operations.

Transform(A, t) is implemented by applying the transformation t to the value of the root of the set A, see Algorithm 7.13. Consequently, all elements belonging to set A are influenced.
Chapter 7. Rigid Constraints

### Union

(1) **procedure** Union(A, B)
(2) /// union the set A and B
(3) begin
(4) if \( \text{weight}_A < \text{weight}_B \) then
(5) \( x \leftarrow A; y \leftarrow B \)
(6) else
(7) \( y \leftarrow A; x \leftarrow B \)
(8) end if
(9) \( \text{val}_y \leftarrow \text{val}_x^{-1} \circ \text{val}_y \)
(10) \( \text{parent}_y \leftarrow x \)
(11) \( \text{weight}_x \leftarrow \text{weight}_x + \text{weight}_y \)
(12) end

---

### Algorithm 7.10

---

### Find

(1) **procedure** Find(i)
(2) /// get the set to which \( i \) belongs
(3) begin
(4) PathCompression(i)
(5) return parent\(_i\)
(6) end

---

### Algorithm 7.11

---

### GetValue

(1) **procedure** GetValue(i)
(2) /// return the value of \( i \)
(3) begin
(4) if \( (i = \text{parent}_i) \) then
(5) return \( (\text{val}_i) \)
(6) else
(7) PathCompression(i)
(8) return \( (\text{val}_{\text{parent}_i} \circ \text{val}_i) \)
(9) end if
(10) end

---

### Algorithm 7.12
Now to the correctness of the algorithms.

**Proposition 7.3** Up and Union preserve Invariant 7.1.

**Proof** Up(i): Assume that parent\(_i\) \(\neq\) parent\(_{\text{parent}_i}\), otherwise noting happens. The value \(\text{val}(i)\) for \(i\) before Up(i) was

\[
\text{val}_{\text{root}_i} \circ \ldots \circ \text{val}_{\text{parent}_i} \circ \text{val}_i.
\]

Afterwards parent\(_i\) is not more on the path to the root. But instead \(\text{val}_i\) is \(\text{val}_{\text{parent}_i} \circ \text{val}_i\) which compensates the missing term.

Union(A, B): assume that the weight of B is at least the weight of A such that B becomes the child of A. Then everything belonging to the old set A does not change. Each vertex pointing to B still points to B but from there to A. If the new values of \(i\) and B are \(\text{val}^*_i\) and \(\text{val}^*_B\), respectively. Then

\[
\text{val}^*_i(i) = \text{val}_A \circ \text{val}^*_B \circ \ldots \circ \text{val}_{\text{parent}_i} \circ \text{val}_i \\
= \text{val}_A \circ \text{val}^{-1}_A \circ \text{val}_B \circ \ldots \circ \text{val}_{\text{parent}_i} \circ \text{val}_i \\
= \text{val}_B \circ \ldots \circ \text{val}_{\text{parent}_i} \circ \text{val}_i \\
= \text{val}(i).
\]

Hence, Invariant 7.1 is preserved. \(\square\)

Note that no other operation except Up and Union manipulates the values. The amortized time needed for a series of operations can be computed of the time of the standard Union/Find.

**Theorem 7.4** Given a set of elements \(V\), \(|V| = n\), to each of which an element of the group \(G = (s, \circ)\) is attached. A sequence of \(a < n - 1\)
Unions, b \geq a \text{ Finds, } c \text{ Transforms, and } d \text{ GetValues runs in } O(a + c + (b + d)\alpha(a + b + d, a)) \text{ time and performs } O(a + c + (b + d)\alpha(a + b + d, a)) \text{ group operations.}

**Proof** Up runs in constant time and performs one group operation.
Union runs in constant time and performs two group operations.
Find uses Up procedures in the order of the height of node \( i \).
Transform runs in constant time and uses one group operation.
GetValue invokes Find once and uses one additional group operation.

Now if we reduce the extended version of Union/Find to the classical one, we observe that Transform uses constant time plus a single group operation. GetValue is simply a Find plus another group operation. Thus, we get the indicated time complexity by simply summing up the operations.
Whenever we use constant time, we also use at most a constant number of group operations and vice versa. Therefore, we get the identical complexity for group operations.

An efficient implementation of the Union/Find extension is crucial for the running time of the ROMA-solver. For instance, to solve the 5-axes machine, see Section 8.1.2, we need approximately

- 6 calls of Union. Two coordinate systems are coupled whenever two parts are rigid towards each other. We have 7 parts at the beginning and 1 rigidity component at the end.
- 1190 calls of Find. Those are used to determine whether two parts are already rigid with respect to each other. This is done by comparing the result of Find of the two parts. When applying the rules of the local solver, this test must be performed often.
- 23 calls of Transform. At the end coordinate systems are transformed to instantiate a single solution. During solving they are used to generate situations which can be analyzed more easily.
- 1060 calls of GetValue. The current location of the coordinate systems is needed as soon as the location of geometric entities are used. These entities are used to express the constraints. During the decision which rule to apply, the location of the geometric entities are inquired often.
7.7 The Ultimate ROMA-Solver

The previous version of the ROMA-solver, see Algorithm 6.2, enforces strong 3-block-consistency. Now it is enhanced to isolate dependent variables as shown in Algorithm 7.14.

In order to deal with cascading isolations we introduce a second queue $RQ$. Into that queue pairs of bodies are put whose coordinate systems need to be coupled. At the beginning $RQ$ is empty because each constraint has been checked for rigidity already at insertion time.

The content of $RQ$ is always handled before that of $Q$. This avoids unnecessary calls of $\text{Cat}$ and subsequent dependencies of constraints. Whenever there is a rigidity between $i$ and $j$, we have two choices: either $\text{Isolate}(i, j)$ or $\text{Isolate}(j, i)$ can be called. If either one is chosen, we need to reformulate all constraint adjacent to the vertex to be isolated. Thus, it is a good strategy to isolate the vertex with fewer constraints, Lines 11–15. In this way, we obtain a star-like structure, because vertices with large degree tend to be centered.

During isolation of vertices blocks can collapse (see Figure 7.6 on the right). Hence, each time, we take a triple of the queue, a check is needed whether or not the triple is still biconnected, Line 19.

It is a good strategy to test triples first which lead to rigid constraints. Then we avoid unnecessary combination of other constraints. For this purpose the queue $Q$ is now organized as a priority queue. The priority of a triple $(i, j, k)$ is the sum over the estimated restriction of $C_{ij}$, $C_{ik}$, and $C_{jk}$. The estimated restriction is computed by the $\text{GetPrority}$ routine described in Section 4.4.4.

The priority corresponds to an estimate of the Kutzbach-Grübler criterion, see Section 3.3 for the subgraph consisting of the bodies $i$, $j$, and $k$. Although not always exact this is a successful heuristic.

7.8 Summary

In this chapter, we have analyzed how functional and bijective constraints can be handled during consistency enforcement. For this purpose we introduce a new procedure $\text{Isolate}$, which—in connection with block-consistency—$\text{isolates}$ dependent variables by removing redundancy in presence of functional constraints. This scheme is called IBC, isolating block-consistency.
ROMA-Enforce*

(1) procedure RomaEnforce*
(2) // precondition: Add has been applied on each constraint
(3) // enforce strong 3-block-consistency by using Add & Cat
(4) // isolate rigid vertices
(5) begin
(6)   \( RQ \leftarrow \emptyset \)
(7)   \( Q \leftarrow \{ (i, j, k) \mid i, j, k \text{ biconnected, } C_{ik}, C_{jk} \in \mathcal{C} \} \)
(8)   while \( RQ \neq \emptyset \) or \( Q \neq \emptyset \) do
(9)     if \( RQ \neq \emptyset \) then
(10)    \((i, j) \leftarrow \text{pop}(RQ)\)
(11)    if \( \text{adj}(i) \geq \text{adj}(j) \) then
(12)     RomaIsolate\((i, j, RQ, Q)\)
(13)    else
(14)     RomaIsolate\((j, i, RQ, Q)\)
(15)     end if
(16) else
(17)     // pop triple with highest estimated restriction
(18)     \((i, j, k) \leftarrow \text{pop}(Q)\)
(19)     if \((i, j, k)\) biconnected then
(20)       if RomaRevise\((i, j, k)\) then
(21)         RomaBlockRigidRefresh\((i, k, RQ, Q)\)
(22)       end if
(23)     end if
(24) end if
(25) end while
(26) end

Algorithm 7.14
In geometric CSPs rigidity can elegantly be handled by IBC. *Isolate* is applied greedily. Thus, we are able to treat dynamically occurring functional constraints as well. Such constraints occur in geometric CSPs often. It is important to note that the resulting CSPs are not affected by the sequence of isolating.

During isolation of vertices constraints need to be reformulated. For geometric constraints, we refrain from an explicit reformulation of constraints. Instead, the reformulation is implemented by an extension of the Union/Find data structure. In this way, we save the expensive call of *Cat* in *RomaCompatible*. In addition, the operations that manage coordinate systems in the local solver run efficiently.
Chapter 8

Discussion

In the previous chapters, we have introduced the ROMA-solver. In this chapter, we first present some example to show the merits of the ROMA-solver, see Section 8.1. Then we give some conclusions in Section 8.2, state open problems in Section 8.3 and provide the reader with some lessons learnt, see Section 8.4.

8.1 Examples

The following examples are typical constructions occurring in CAD/CAM systems. They are mostly modeled using kinematic joints. Note that these joints are mapped onto base constraints of the ROMA-solver as described in Section 3.1.3.

The elevator door is one of the examples in [Kra92b]. Although the problem is modeled with 3D parts, it is in fact only a 2D problem. In this example enforcing block-consistency helps in preserving the structure of the problem. In addition, running-time is decreased because fewer triples are examined.

Contrary to the elevator door the STARRAG milling machine [STA] is a real 3D problem. It consists of a large rigid loop of 6 parts. The ROMA-solver deals with this loop by propagating combined restriction in the constraint graph. Eventually the restriction is combined in a single constraint which becomes rigid. Then reformulation yields other rigidity until eventually the problem is solved.
The BOSCH saber saw [BOS] illustrates how constructions are built of modules. The saw is built of four smaller entities, each containing loops. In the actual constraint graph these loops are superimposed and not directly visible. This typical for assembly problems as argued in Section 4.2. The solving strategy of the ROMA-solver is able to detect the modules implicitly and solves them efficiently by using isolating block-consistency.

8.1.1 Elevator Door

The elevator door is one of the examples, see [Kra92b]. There 3 examples of kinematic linkages are introduced: a sofa bed mechanism, an automobile suspension, and the elevator door. We chose the elevator door as example because it demonstrates the merits of block-consistency nicely.

In [Kra92b], Kramer writes:

[Figure 8.1] shows the mechanism used for opening and closing the door of an elevator. Its design has changed little in the past ninety years, and is used by a large number of elevator manufacturers. It is a ten-bar-linkage, consisting of two interconnected six-bar-mechanisms which share link 1 and the ground.

The drawing in Figure 8.1 is a so-called kinematic diagram. In such a diagram the links are represented by long sticks. In turn, the joints are the nodes of the graph. Contrary to that, in a constraint graph, see Figure 8.2 the links are the nodes and the joints are the edges.

The ground link in a kinematic diagram is fixed in space. It is also referred to as reference frame. In the constraint graph the ground link is introduced as separate body, the part 1.

Again Kramer:

The ground link is the body of the elevator cab. The passenger compartment is shown as a gray rectangle. Link 2 is a flywheel which in the real mechanism is driven by a motor; since only two points on the flywheel are of interest, it is represented here as a line. The rotation of the flywheel is the driving input. As link 2 rotates, it moves the push-rod (link 3) causing link 4 to rotate about its ground joint. Link
5 is connected to link 6, the elevator door, which is drawn as block rectangle to aid visualization of the mechanism. The door slides along the Prismatic joint. Link 5 is necessary because the end of link 4 moves in a circular path and link 6 moves in a straight line.

The 13 constraints comprise 1 driving Revolute used to drive the mechanism, 2 Prismatic joints to let the door slide horizontally, and 10 Revolute joints.

Whenever the angle of the driving Revolute is chosen, the left and right door can be computed separately. However, if we look at the constraint graph in Figure 8.2, we see a single block only. However, a driving Revolute can be treated as a rigid constraint. Isolating block-consistency detects this automatically.

Vertex 2 has a smaller degree than vertex 1. Thus, we isolate 2. This takes place as soon as the driving Revolute is inserted. The graph in Figure 8.2 on the right side is obtained. It consists of 3 biconnected components \(\{1, 3, 4, 5, 6\}, \{1, 7, 8, 9, 10\}, \text{ and } \{2\}\), which are treated separately by block-consistency. The constraints of the right and left door will never influence each other because no `Cat` was called before isolation.
Initially queue \( RQ \) of Algorithm 7.14 is empty. Queue \( Q \) is filled with the following 18 triples:

\[(1, 3, 4), (1, 4, 3), (1, 4, 5), (1, 6, 5), (3, 1, 4), (3, 1, 6), (3, 4, 5), (4, 1, 6), (4, 5, 6),
(1, 7, 8), (1, 8, 7), (1, 8, 9), (1, 10, 9), (7, 1, 8), (7, 1, 10), (7, 8, 9), (8, 1, 10), (8, 9, 10)\]

Note that without block-consistency we would have an additional 15 triples in \( Q \): 6 involving the constraint \( C_{12} \) and 9 containing an edge from the left and the right part of the mechanism each, for instance \((7, 1, 3)\). Hence, using block-consistency approximately halved the number of triples in \( Q \).

In the following, we only report on the solving of the right block. The left block is solved simultaneously.
First, a triple is taken of the queue $Q$, namely one with maximal estimated restriction. The loop $(1,3,4)$ is restricted most because 3 Revolute joints are involved, each of which reduces 5 DOF. There are 3 triples involving these variables. An arbitrary one is chosen, for instance $(1,3,4)$. Consequently, Cat($C_{13}$, $C_{34}$) is called. Because the axes of the two R joints are parallel, the result is a vc-coi-vc, pt-on-pl plus a pt-on-sp. This combination is then added to the original $C_{14}$.

As result we get a rigid constraint $C_{14}$ with 3 redundancies. The redundancy is reported to be influenced by $C_{13}$ and $C_{34}$ only. This helps the engineer to possible change the design. Here the redundancy seems to be in order. But it is important to notice it.

The constraint $C_{14}$ is rigid. It is put into the queue $RQ$ and is treated next. The rigid connection $C_{14}$ is used to isolate vertex 4, see Algorithm 7.3. For this we bring 1 and 4 into a legal relative position and orientation by satisfying $C_{14}$, and call Union(1, 4). Furthermore, the adjacent $C_{43}$ is added to $C_{13}$ and $C_{45}$ is stored as new $C_{15}$. The graph is Figure 8.3 is obtained.

The $C_{13}$ is also rigid. It is put into $RQ$. In the next step vertex 3 is isolated by coupling the coordinate systems of 1 and 3. There are no constraints adjacent to 3 expect 1. Thus, no further reformulation takes place.

Solving proceeds with the next triple in $Q$ that is still biconnected and has a maximal estimated restriction. This is the triple $(1,5,6)$. We process this triple by calling Cat($C_{15}$, $C_{56}$) and add the result to $C_{16}$. The result is rigid. Therefore, it is put into $RQ$. Consequently, vertex 6 is isolated, the adjacent constraint $C_{65}$ is reformulated which leads to a rigid $C_{15}$, and 5 is also isolated. The right block is solved, see Figure 8.4.

The left block with vertices 1, 7, 8, 9, and 10 is solved accordingly. Finally, we obtain the graph in Figure 8.4.

By choosing the triple $(1,3,4)$ first, we get only slight influences among constraints. The constraint $C_{14}$ is influenced by $C_{13}$ and $C_{34}$. The constraint $C_{13}$ is the combination of the initial $C_{13}$ and $C_{34}$. $C_{15}$ is composed of the initial $C_{15}$ and $C_{56}$ and $C_{16}$ is influenced by $C_{15}$ and $C_{56}$. These are the fewest influences possible. If we had chosen the triple $(1,4,3)$ at the beginning, vertex 3 would have been isolated first, and $C_{13}$ would also be influenced by $C_{14}$. Note that in either case, the influences are only of constraints in the same biconnected component.
In this example we have seen that by isolating vertices in case of rigidity, we get cascading effects leading to further rigidity. This is often the case in larger assemblies as well. Typically, as soon as a loop is solved, we reformulate and receive a reduced problem with fewer bodies. Our experience has shown that solving the first loop in a large assembly is often the hardest.

8.1.2 STARRAG Milling Machine

The next example is to solve the inverse kinematics of a 5-axes machine. The machine used in this example in a STARRAG milling-machine, see Figure 8.5. It is used to manipulate a workpiece by a tool. Between the workpiece and the tool there is a kinematic chain consisting of 5 parts, which can be moved relatively to each other in a defined way indicated by the constraints.

Often working machines or robots are designed in a similar way. The differences are in the numerical data of the axes or in the dynamical behavior, which is not treated by the ROMA-solver.

In inverse kinematics the goal is to compute the position and orientation of all intermediate parts such that the tool has a predefined position and orientation with respect to the workpiece. Note that in general there are several solutions. However, by restricting the allowed parameter space for the joints, we may tailor the problem to have a unique solution. The restriction is realized by inserting appropriate inequality constraints, such as a \( \text{vc-minang-vc} \), see Section 3.1.1.
The computation of the relative position and orientation is used in a larger context, for instance to steer the mechanism or to simulate it.

In the STARRAG milling machine a spindle rotates to work on the workpiece. The DOF of the spindle is not used to steer the mechanism. For modeling we have two opportunities:

1. Theoretically the tool can be split into two parts, 1 and 2 between which there is a Revolute joint, see Figure 8.6 on the left. The one part represents the spindle connected to the mechanism. The other part represents a virtual part attached at the workpiece. Between the second part and the workpiece, there is a rigid connection.

2. In practice the modeling is as follows. The desired relative position of the workpiece and the tool is not determined by a full $fr$-$coi$-$fr$, but by giving only a point, vector pair which must coincide. This constraint corresponds to a Revolute joint. In that model the tool is not split into two parts, see Figure 8.6 in the middle.

In either case the loop is solved in the same way. In the first situation the ROMA-solver directly isolates vertex 7 such that an RRRPPP loop is obtained with an additional rigid vertex. In the second case there is also an RRRPPP loop but without the additional vertex. Here isolating-block-consistency proves to be an elegant approach to get the same problem.
of different models, see Figure 8.6 on the right. In the following figures, vertex 7 is skipped.

The chain consists of 6 parts each of which corresponds to a link. The workpiece (vertex 7) is attached to a part (vertex 6), which can pushed in one direction. This corresponds to a Prismatic joint. The part 6 itself can be moved along an axis relative to part 5, which itself is movable to part 4. Both joints corresponds to further Prismatic joints. The axes of the three Prismatic joints are pairwise orthogonal. Part 4 is the frame of the machine. Attached at this frame, there is a part which can be rotated (vertex 3). This is modeled as a Revolute joint. Another Revolute joint, which has a rotation axis orthogonal to the first, is attached to part 2.

The queue $Q$ contains all paths of length 2 where all vertices are biconnected. These are 6 triples. In the following, we restrict ourselves to the most interesting steps.

First, we consider the triple $(1, 6, 5)$ implying two Prismatic joints, see Figure 4.8. A call of $\text{Cat}(C_{16}, C_{65})$ delivers a pt-on-pl plus a th-coi-th, which is inserted as $C_{15}$. Then we consider the triple $(1, 5, 4)$. As the result of $\text{Cat}$ we get a th-coi-th stored as $C_{14}$. This constraint restricts all RDOF. The graph on the left of Figure 8.7 is obtained.

Then the triple $(1, 4, 3)$ is examined. A th-coi-th in series with a Revolute implies a vc-coi-vc. Afterwards $(1, 2, 3)$ is examined. The
result of two Revolute in series is, in this case, a Universal joint, which is added to the already existing $C_{13}$. In this case, adding a Universal joint and a vc-coi-vc delivers a rigid constraint. Hence, we can isolate vertex 3 and obtain the graph in Figure 8.7 on the right.

The adjacent constraints of 3 is reformulated of which follows that $C_v$ becomes rigid. Hence, 2 is also isolated, see Figure 8.8 on the left. In addition, the reformulation of $C_{34}$ eventually leads to the solving of the remaining parts such that entire mechanism is rigid, see Figure 8.8 on the right.

There are some steps in the solving which have not been reported. For instance, we would also have inserted the constraint $C_{64}$ as result of considering the triple (6, 5, 4). This would not be necessary for solving. However, it is difficult to decide this a priori.

Solving is fast. Running on a normal workstation, the ROMA-solver needs less than a second, whereas Numerica needs 243 seconds to solve a similar 5-axes machine as reported in [VMD97]. In addition, Numerica only delivers a single solution, whereas the ROMA-solver reports on rigidity, remaining DOF, redundancy, degeneracy, and numerical instability of the problem.
Figure 8.8: 5-axes machine: final steps.

Figure 8.9: Saber saw: picture without housing.
8.1.3 Saber Saw

The BOSCH saber saw, see Figure 8.9, demonstrates nicely that mechanisms are constructed modularly and that the ROMA-solver handles them similarly.

The saber saw is an electrically driven saw. From the kinematic point of view the task is to convert a rotating motion of the engine into the desired motion of the saw rod, see Figure 8.10.

In order to solve this task, the problem is divided into subtasks. For the saber saw we have 4 subtasks or modules. Each module consist of a basic kinematic loop. At the end these modules are superimposed to build the overall mechanism.

The fact that the saw is modeled in modules helps to overcome the complexity of the mechanism and to structure it. This general scheme is called divide and conquer.

The saber saw basically consists of four module.

1. The frame into which the mechanism is mounted.

2. A mechanism to generate an oscillating motion out of a rotation shaft. This is used to push the saw rod forward and backward.

3. A mechanism to generate another, smaller oscillation to move the saw rod up and down.

4. A mechanism to move the saw rod.

The frame of the saber saw is built of 5 bodies, see Figure 8.12. Part 5 is the lower housing. Into that part two flanges are mounted, bodies 2 and 3. Each flange is fastened at part 5 by two screws. The screws are not modeled as objects but as a Revolute joint. Then two screws already result in a rigid connection. Actually, modeling screws as
Figure 8.11: Saber saw: kinematics.
8.1. Examples

Revolute does not correspond to reality. However, the actual motion of screws is not interesting in this case. The upper housing, body 6, not shown in Figure 8.12, is rigidly connected to the lower housing such that the inner part of the mechanism is protected.

Into the flanges two rods are inserted, bodies 1 and 4. Each rod is mounted to the left flange, body 2, by an Identical joint. In addition, to rods are supported by a Cylindrical joint towards the right flange, body 3. These supports are used to increase stability. The support is implemented as Cylindrical instead of Revolute to enable disassembling the mechanism by shifting the rods along their axes.

The frame is a rigid structure which is used in the following to install the mobile parts.

The function of the second module is to transform the rotating motion of the shaft, body 7, into a tumbling motion of the tumbling piece, body 9. The shaft is driven by an engine which is attached to the housing. The shaft is inserted into a bearing attached at the housing. The bearing allows the shaft to rotate. Actually, there are two bearings: one allowing a Cylindrical mobility and one—at the engine—which
Figure 8.13: Saber saw: second module, tumbling piece.
8.1. Examples

a Revolute joint. The mobility of the shaft is parameterized to allow driving of the mechanism. Hence, it is modeled as a driving Revolute. By this the mobility is parameterized such that for each angle of the rotation a fixed solution can be generated.

In order to generate this motion, the tumbling piece is implemented to rotate with respect to the shaft. The rotation axis is skewed with respect to the axis of the shaft. Hence, the tumbling piece has two DOF with respect to the housing, one to rotate along its axis and one to rotate together with the shaft. In order to remove the latter mobility, the upper part of the tumbling piece is put into a cylindrical hole of the counter weight, body 8. The counter weight is attached to the two rods of the housing, bodies 1 and 4, such that a Cylindrical joint is allowed with respect to each of the rods. These two Cylindrical joints together form a Prismatic joint because their axes are parallel, as seen in Figure 4.8.

The sphere attached at the tumbling piece fits into the hole of the counter weight. This rare joint is modeled as a pt-on-ln where the point is attached at the center of the sphere and the line is axis of the hole in the counter weight. In order to remove the duality—the tumbling piece could also be in a position on the other side of the axis of the shaft—an additional pt-abv-pl is entered, indicated by *. The relative mobility of the tumbling piece with respect to the housing is, hence, the concatenation of a pt-on-ln with a Prismatic. This results in a pt-on-pl, where the plane is spanned by the vector of the line and the vector of the Prismatic joint.

Combining the two restrictions imposed on the tumbling piece results in a tumbling motion when the shaft is driven.

The third module is to implement an oscillating motion of the support, body 10, with respect to the housing. Its purpose is to contain the saw rod. The support is attached to the upper housing. The upper housing is fixed towards the lower housing. The support is able to rotate around an axis attached at the upper housing. This is modeled as a Revolute.

On the other side, the support touches the shaft. At the shaft an eccentric piece is attached. Therefore, the support oscillates up and down while the shaft is rotating. Actually, the constraint between the shaft and the support should be a plane touches sphere. However, this constraint does not exist in our type set. Therefore, it is simplified as a pt-on-pl where the point is in the center of the eccentric piece and the
Figure 8.14: Saber saw: third module, support.

The purpose of the fourth module is to move the saw rod using the tumbling motion of the tumbling piece. Actually, only the revolving motion is used because the rotating motion of the tumbling piece around its own axis is taken away by the Cylindrical joint between the tumbling piece and the nipple. The saw rod is able to glide in the support, modeled as a Cylindrical joint. Hence, it may translated along an axis and rotate. As we have seen in the third module, the support oscillates slightly with respect to the housing when the shaft is driven. However, for each fixed position of the shaft the support is rigid with respect to the housing.

To the saw rod a nipple, body 12, is attached with a Cylindrical joint. This nipple fits into a hole in the upper part of the tumbling piece. When the tumbling piece is oscillating, the nipple is moved more or less into that hole. In addition the saw rod is translated.

When putting all four modules together, the constraint graph in Figure 8.16 is obtained. In this graph the modular structure vanished.
8.1. Examples

Figure 8.15: *Saber saw: fourth module, saw rod.*

Figure 8.16: *Saber saw: the entire constraint graph.*
because it is modeled on the level of parts instead of on the module level. However, as we shall see, the ROMA-solver is nevertheless able to use the modular structure implicitly during solving.

The relative motion of the support and of the tumbling piece are such that the saw rod moves forward, then down, then backward, and then up. The forward/backward motion is dominant. However, it is important that the saw is moved forward above the material to be sawed, then inserted into the material, and finally drawn back.

The kinematic purpose of the counter weight was already explained earlier. However, there is also a dynamic purpose. When the saw rod is moved forward, the counter weight is moved backward and vice versa. This is to support the backward force of the mechanism and to make the mechanism more convenient for the end-user.

The ROMA-solver is able to detect the modules implicitly. First, the rigid structure of the housing is detected and used to isolate several bodies. Either body 2 or 3 is taken as the center of the housing, because the vertices have the largest degree. Then the inner mechanisms is solved.

First, the shaft is also isolated, because a driving Revolute is treated as a rigid constraint. Then we can solve the support, body 10, which is also rigid with respect to the housing. We isolate that vertex. Then the triangle (7, 8, 9) can be solved. Finally, the triangle (3, 12, 11) becomes rigid, which corresponds to the fourth module. The
structure of the final graph is again star-like, see Figure 8.17 and has therefore a width of 2. Every relation is rigid such that a solution can be assigned easily.

It is typical for constructions to comprise several modules which are constructed sequentially. Likewise, it is typical for the ROMA-solver to exploit the modular structure and to solve the mechanism constructive. By the Kutzbach-Grübler heuristic to consider triples first that are most restricted, the ROMA-solver most likely finds the beginning of the module chain. Note that if it does not find the proper start of the module chain, the same result is still obtained. However, solving is slowed down slightly. Isolating block-consistency helps in not considering useless triples and to minimize influences between constraints.

8.2 Conclusions

In this thesis, we presented the ROMA-solver for the assembly in mechanical engineering. The ROMA-solver is rather successful in typical constructions we are aware of. This is demonstrated by the following facts.

- The ROMA-solver is able to handle even difficult mechanisms built of kinematic joints as demonstrated in the examples.
- The ROMA-solver is fast—most constructions up to 100 parts are solved in under 1 second—such that it can be used in commercial systems.
- The ROMA-solver deals with under- and over-constrained 3D-systems easily and supports variational design.
- The language in which the constraints are entered corresponds to language of the engineer, namely to kinematics. The result is also expressed in that terminology and is therefore clearly comprehensible to the user.
- The ROMA-solver provides more information than previous solvers on rigidity structure, mobility, the amount and the source of redundancy, degeneracy, and numerical instability.

The success of the ROMA-solver is based on the following conceptual decisions and theoretical achievements.
- The solving process of the ROMA-solver resembles a human constructor. Thus, constructible cases correspond to solvable cases. Those cases that cannot be handled by the solver are most often badly designed and should be changed. In addition, the result is given on a high abstraction level, such that it has a clear semantic meaning for the engineer.

- The ROMA-solver avoids to search for rigid substructures. Instead, it enforces consistency with the help of the procedures Add and Cat. This has two major implications. First, the number of geometric cases that need to be implemented is reduced. Second, solving is generally faster because search is limited.

- Consistency enforcement requires a careful selection of constraint types, because constraints are not only used to describe the initial situation, but also to propagate combined restrictions. With the powerful constraint type set presented in Chapter 3.1, this approach is useful for the assembly.

- The clear separation between geometry and constraint satisfaction, see Chapter 5, may seem as a limitation. However, it has proven to lead to fruitful abstractions and thin interfaces which compensate for the limitations.

- Block-consistency presented in Chapter 6 saves a considerable amount of time during solving. More specifically, the number of triples that need to be supported is decreased. In addition, during block-consistency enforcement, the block-structure is preserved. Thus, constraints influence each other only if they are in the same block. By this the source of redundancy can be located on the block level. Nevertheless, all necessary information on rigidity, redundancy, mobility, and numerical instability is already computed during BC-enforcement.

- Isolating vertices in presence of functional constraints is a powerful tool to simplify a problem as demonstrated in Chapter 7. With this approach, not only the initial but also the dynamically occurring rigidity can be exploited. Isolation of vertices leads to smaller blocks which can be exploited by block-consistency. Consequently, there are fewer influences of constraints.

- In the ROMA-solver a new extension of the Union/Find data structure is used to reformulate constraints in presence of rigid
constraints, see Section 7.6. For this, the Union/Find data structure is enhanced to deal with additional operations \texttt{Transform} and \texttt{GetValue}. By this approach, coordinate systems can be united, which avoids expensive reformulation of constraints.

- After the enforcement of strong 3-consistency the final graph provides information on the initial situation. However, it might be possible that a solution cannot be found without backtrack, see Theorem 5.1. In these situations, a relaxed problem is computed and instantiated, see Section 5.4. For that, a minimum spanning tree of constraints is taken. The relaxed solution does not satisfy all initial constraints, but it can be used for two purposes. First, it gives a good initial assignment for numeric iterative approaches. Second, in rigid situations, the engineer sees, where there is still work to do. Rigid situations are common, provided that driving input constraints are used to parameterize the actual mobility.

The theoretical contributions—block-consistency, isolating of vertices, and extension of the Union/Find data structure—are examined on a general level. They proved to be useful in assemblies. Nevertheless, they may also be used for further application.

\section*{8.3 Open Problems}

Whereas the overall approach of the ROMA-solver seems to work fine for assembly problems, there still are possible improvements.

Up to now, the ROMA-solver is able to deal with parts which must be positioned and oriented in space. However, it might also be interesting to add a freedom to scale uniformly. Then constraints are not only satisfiable by translating and rotating but also by scaling. In addition, the scale freedom should be supported by special constraints removing the entire scaling, alike \texttt{pt-coi-pt} for translation and \texttt{th-coi-th} for rotation. However, as some preliminary studies have shown, the effort to implement a scale freedom, even a universal one, should not be underestimated.

Instead of increasing the freedom of the parts, the ROMA-solver may also be used for solving problems living in subspaces of $SE(3)$, for instance problems in 2D space, problems without rotation or problems
without translation. These subproblems can be generated by introducing additional constraints. The 2D subproblem for instance is obtained by inserting a \(p_1\)-coi-\(p_1\) between the reference frame and each part.

However, there are still two open problems to be considered. The first problem is that the block structure is destroyed by inserting constraints between the reference frame and each body. Instead, any other spanning tree can be chosen, such that there are no new blocks arising. A spanning tree is enough because the constraints to build the previously mentioned subproblems are transitive. However, the spanning tree may change when constraints are inserted or deleted. The second problem is that the constraint type set may not be appropriate for the new situation. For instance, for 2D-sketchers constraints to handle ellipses might be needed.

In general, it is interesting to extend the constraint type set. For instance, by adding the only lower pair currently not implemented, the Screw joint. Whether it is really necessary is open. Most of the screws are not modeled anyway. However, there are some mechanisms in which a screw-like motion between parts is needed. This is connected with the introduction of a general constraint to cover spindles, cog-wheels, or the constraint “point on general hyperbolic curve”.

Another nice feature for the modeler would be to allow parts with inner symmetries. In the ROMA-solver, each part has 6 initial DOF. However, we can imagine parts that have for instance a rotational symmetry. In the current version, we must add an arbitrary restriction to reduce the rotation for such a part. The problem is: relative to which second part? This might change when the rest of the graph is changed. Then we have to keep track of this by hand. Another solution would be to handle inner symmetries automatically by adding constraints to parts. These constraints are dynamically taken during solving between the appropriate pair of bodies such that the inner symmetry is reduced. We believe that it might be possible to handle inner symmetries without changing the geometric level of the ROMA-solver. However, this still offers a big conceptual challenge.

The ROMA-solver is based on the concept of block-consistency. In the case of the assembly, during block-consistency enforcement, everything useful—redundancy, illegality—is already computed, as shown in Theorem 6.4. Maybe, this is also enough for other applications. Two things are worth further investigations. First, it is interesting to test block-consistency in further applications. Second, it might be fruitful
to use block-consistency as basis for parallelization of constraint solvers, as suggested already in Section 6.3.3.

Another question occurring in the context of block-consistency is whether general $k$-block-consistency can also easily be extended to $k$-consistency as this is the case for $k = 3$. However, for this purpose, $(k - 1)$-ary constraints need to be designed.

During enforcement of strong 3-consistency, solving of sub-parts is separated at the block borders. Fewer triples are examined and, hence, a considerable amount of time is saved. However, within the blocks, the ROMA-solver tends to solve the problem several times at different places in the graph. For instance, if we look at the solving of the 5-axes machine, we see that constraints have been computed that are never used for isolation. Although it seems improbable to be able to detect these constraints in beforehand. There might be heuristics to decrease the number of unused computations.

After having enforced strong 3-consistency to the graph, the result obtained is analyzed. If the width of the remaining problem is not higher than 3, then a solution can be found without backtrack. If this is not the case, one possibility is to consider a relaxed problem, see Section 5.4. The relaxed problem at which we are currently looking is a minimum spanning tree which has proven to be a good hint for the engineers. A minimum spanning tree can be computed efficiently with the algorithm of Prim. However, as we have established strong 3-consistency, any subgraph of width 2 could also be instantiated. However, it is not clear how such a subgraph can be computed efficiently and whether is would help more than the minimum spanning tree. But from the theoretical point of view, it is surely desirable to know about this.

8.4 Lessons Learnt

The project to develop the ROMA-solver started about 4 years ago. It was a project in which several institutions were involved. The CAD/CAM company Precisionsoft, the Institute of Manufacturing Technology and Machine Tools (IWF) and the Institute for Theoretical Computer Science (IfTI). At the beginning of such projects it is always a crucial point to synchronize the impressions which the different parties have. For this, it has proven to be a big advantage to start building a prototype. This has an impact to the project in several ways.
People tend to rely on their experiences. The feeling for the new problem is always influenced by analogies. However, often these experiences are too general and do not apply entirely. A prototype helps in deciding whether the general opinion holds or not.

In a prototype, work is distributed among different project members. To be able to communicate, one builds interfaces. Thin interfaces are better than large ones, because they need to be changed less often. These interfaces can also be used to get an abstraction of the overall problem. On the one hand, this is a restriction because the information flow is limited. On the other hand, it might be fruitful, because energy can be focused on partial problems. In the ROMA-solver, the abstraction between the geometry and the constraint solving is one point, where interfaces have proven to be fruitful.

The time needed to develop a new piece of software is not easy to estimate. Currently, IT projects tend to exceed the estimated resources by far. A prototype is a good thing to have as a backup, when the real thing could not be finished in time.

In this thesis, constraint satisfaction is used heavily. Constraint satisfaction theory is a general framework covering several NP-hard problems. Thus, we never expect to find any efficient algorithm for the general problem.

Nevertheless, for special subclasses there might be efficient solving approaches. For instance, CSPs with functional constraints can be solved efficiently as shown by van Hentenryck et al., David, and also in this thesis. Most of the efficiency is discovered by looking at special applications, in which intuition says that there must be something faster.

Constraint satisfaction theory is a framework to cover many different areas. Thus, it is a melting pot for the ideas of these areas which has proven to be fruitful for the development of solving strategies.
Bibliography


[Nad88] B. A. Nadel. Tree Search and Arc Consistency in Constraint Satisfac-
tion Algorithms. In L. Kanal and V. Kumar, editors, Search in Artifi-

[OMSE98a] G. Ogan, F. Metzger, B. Seybold, and M. Engeli. Kinematic Chain Sub-


[Owe96] J. C. Owen. Constraints on Simple Geometry in Two and Three Di-

[Pou90] J. A. La Poutré. Lower Bounds for the Union-Find and the Split-
Find Problem on Pointer Machines. In Proceedings of the 22nd Annual

[Reu76] M. M. Reuleaux. The Kinematics of Machinery. Macmillian & Co.,
New York, 1876.


[SMO+97] B. Seybold, F. Metzger, G. Ogan, J. Bathelt, F. Collenberg, J. Taiber, K. Simon, and M. Engeli. Spatial Modelling with Geometric Con-


[SOM98] B. Seybold, G. Ogan, and F. Metzger. Functional Constraint and Re-
162 Bibliography


Seite Leer / Blank leaf
List of Figures

1.1 An assembly and its constraint graph. ................. 2
2.1 A plug. ........................................... 9
2.2 The sketch of an Oldham-coupling, [Ehr95]. ............ 11
2.3 Constructing the middle between two points. ........... 11
2.4 The Steward platform. ................................ 17
3.1 Lower pairs, [Cra89]. ................................ 20
3.2 A marker, a trihedral, a frame. ........................ 21
3.3 Geometric elements in 3D space. ........................ 22
3.4 Constraints and their restriction. ........................ 26
3.5 The plug and its constraint graph. ........................ 28
3.6 Rigid and mobile structure with 6 orthogonal sticks. ... 30
4.1 The constraint graph of the saber saw. ................. 40
4.2 A 3-loop and 4-loop. ................................ 42
4.3 Data fields and routines of constraint. .................... 48
4.4 Procedure Add. ..................................... 50
4.5 Two pt-coi-pt constraints in parallel. ................. 50
4.6 Procedure Cat. ...................................... 53
4.7 Two pt-coi-pt constraints in series. .................... 53
4.8 Two Prismatic in series. ................................ 54
4.9 The architecture of the ROMA-solver. .................... 58
5.1 A 2-coloring problem. ........................................... 64
5.2 A CSP which is 3-consistent but not 2-consistent [Fre82]. 65
5.3 Complexity of consistency algorithms for finite domains. 66
5.4 Computing the width. ............................................ 74
6.1 A graph with several blocks. ................................. 86
6.2 Redundant, explicit, and satisfiable constraint. .......... 87
6.3 The adjacency list of a vertex in a connected set. ....... 90
6.4 Block-consistency. .............................................. 91
6.5 The situation for Lemma 6.2. ................................. 92
6.6 The situation for Lemma 6.3. ................................. 93
6.7 Theorem 6.4: the cases for path-consistency. ............. 95
7.1 A problem and a solution graph. ............................. 106
7.2 The saber saw. From left to right: the initial graph, after
enforcement of strong path-consistency, after removal of
redundant constraints. ............................................. 106
7.3 Functional constraint in a graph. ............................. 110
7.4 Subsequent isolation of vertices. ............................. 115
7.5 Pivot consistency compared to block-consistency. ....... 116
7.6 Different width after different choice of Isolate. ....... 116
7.7 Path compaction strategies: initial situation, path com-
pression, path splitting, path halving. ......................... 123
8.1 Elevator door: kinematic diagram, [Kra92b]. ............ 135
8.2 Elevator door: initial CSP, after Isolate(1, 2). ........ 136
8.3 Elevator door: vertex 4 is isolated. ......................... 136
8.4 Elevator door: one block solved, global solution. ....... 138
8.5 The STARRAG milling machine. ............................. 139
8.6 5-axes machine: Modeling. .................................. 140
8.7 5-axes machine: finding a rigid constraint. .............. 141
8.8 5-axes machine: final steps. ................................ 142
8.9 Saber saw: picture without housing. ....................... 142
8.10 Saber saw: scheme. .......................................... 143
8.11 Saber saw: kinematics. ......................... 144
8.12 Saber saw: first module, frame. ................. 145
8.13 Saber saw: second module, tumbling piece. ........ 146
8.14 Saber saw: third module, support. ............... 148
8.15 Saber saw: fourth module, saw rod. ............... 149
8.16 Saber saw: the entire constraint graph. .......... 149
8.17 Saber saw: solved. ................................ 150
List of Algorithms

5.1 AC-3 for finite domains ......................... 67
5.2 Revise for finite domains ....................... 67
5.3 Finding supports in finite domains ............ 68
5.4 ROMA-Revise .................................. 70
5.5 ROMA-Enforce .................................. 71
5.6 ROMA-Refresh .................................. 71
5.7 ROMA-Absign .................................. 76
5.8 ROMA-Explore .................................. 78

6.1 ROMA-BlockRefresh .............................. 102
6.2 ROMA-Enforce+ .................................. 102

7.1 Compatible [Dav95] .............................. 112
7.2 Isolate .......................................... 114
7.3 ROMA-Isolate .................................... 119
7.4 ROMA-Compatible ............................... 119
7.5 ROMA-RigidBlockRefresh ....................... 120
7.6 PathCompression ............................... 123
7.7 PathSplitting .................................... 124
7.8 PathHalving ..................................... 124
7.9 Up ............................................... 125
7.10 Union .......................................... 126
7.11 Find ........................................... 126
7.12 GetValue ....................................... 126

169
7.13 Transform ............................................. 127
7.14 ROMA-Enforce* .................................... 130
Index

2-body-problem, 28, 49
AC-3, 66
AC-5, 108
Ackermann function, 121
Add, 49-52
adjacency list, 85
adjacent, 85
allowed, 63, 87
AreEqual?, 56
articulation point, 85
assembly, 8
assignment, 63
BC, 88
biconnected, 85
set, 85
bijective, 100, 109
block, 85
bridge, 85
C, 20, 25
Cat, 53-55
component
  biconnected, 85
  connected, 85
connected, 85
graph, 85
consistency, 64, 73
  adaptive, 74
  arc, 64
  block, 88, 95
directional, 88, 90, 91
  isolating, 115
  strong, 88
closure, 64
directional, 64, 74, 90, 91
  enforcement, 65
  path, 64, 87, 95
  strong, 64
constraint, 58, 62
  base, 22-23
  binary, 22
  compound, 24
  driving input, 25-27
  explicit, 87
  kinematic, 24-25
  network-induced, 86
  redundant, 87, 98
  relation, 62
  satisfiable, 87
  scope, 62
constraint graph, 27, 57
constraint satisfaction problem, 62
  binary, 62
  geometric, 8
CSP, 62
  equivalent, 63
cycle, 85
Cylindrical, 20, 25
deg, 49
degree, 85
degree of freedom analysis, 33–35
degrees of freedom, 28
descendant, 109
DOF, 28
   rotational, 28
   translational, 28
domain, 62
domain-wipeout, 99
DP, 27
DR, 27

Euclidean group, 28

F, 21, 25
Find, 121, 127
fr-coi-fr, 23
functional, 109, 113

GCSP, 8
generic solvers, 31
geometric constructions, 11
GetMobility, 57
GetPriority, 57
GetValue, 121, 127
ground link, 134
grounded, 29

H, 20
higher pair, 21

I, 25
IBC, 115
Identical, 25
ilist, 49
illegality, 99
instance solvers, 31
instantiation, 63
IsIllegal?, 56
isolated, 85
IsRigid?, 56
join, 63
joint, 20

kinematic diagram, 134
kinematics, 10
   forward, 10
   inverse, 10
Kutzbach-Grübner criterion, 29, 57, 129

layout problem, 12
link, 20
linkage, 73
ln-coi-ln, 24
loop, 29
lower pair, 20

map labeling, 13
marker, 21
minimal network, 87

num, 49

over-constrained, 30
   consistently, 30
   inconsistently, 30

P, 20, 25
path, 85
   closed, 85
   simple, 85
   vertex disjoint, 85
path compression, 122
path halving, 122
path splitting, 122
path-consistent, 111, 113
path-induced, 87
path-redundant, 87, 113
pivot, 110
pivot consistency, 111
pl-coi-pl, 24
Planar, 21, 25
prefix, 86
prefix-connected, 86, 90, 91
Prismatic, 20, 25, 54
driving, 27
pt-abv-pl, 23
pt-coi-pt, 23, 50, 53
pt-on-ln, 23
pt-on-pl, 23
pt-on-sp, 23, 47
R, 20, 25
\mathcal{R}-compatible, 110
RDOF, 28
red, 49
reference frame, 134
relation, 63
Revolute, 20, 24, 25
driving, 27
rigid, 25, 31
topologically, 30
root set, 109
minimal, 109
minimum, 109
ruler-compass constructions, 20
ry-coi-ry, 24
S, 21, 25
Satisfy, 55
Screw, 20, 24
\textit{S}E(3), 28, 121
sketching, 10
Sketchpad, 35
solution, 63
spatial reasoning, 12
Spherical, 21, 25
Steward platform, 17, 51
support, 64
TDOF, 28
th-coi-th, 23
Transform, 121, 127
U, 21, 25
under-constrained, 30
Union, 121, 127
by rank, 122
by size, 122
Universal, 21, 25
universal, 63
variational design, 16, 58
vc-ang-vc, 23
vc-coi-v, 23
vc-maxang-vc, 23
vc-minang-vc, 23
Waltz-Filtering, 65
well-constrained, 30
width, 73
graph, 73
ordering, 73
vertex, 73
\xi_k-compatible, 110
Curriculum Vitae

name: Bernhard Jakob Seybold

birth: December 20, 1969, Kiel (GER)
nationality: German

aug 77 - dec 79 Primary school Kronshagen (GER)
jan 80 - mar 81 Primary school in Binningen (CH)
apr 81 - mar 85 Pre-high-school in Binningen
apr 85 - sep 88 High school in Oberwil
Matura type B (Latin)
nov 88 - mar 94 Studies in computer science at ETH Zurich
Dipl. Informatik-Ing. ETH (Master of Computer Science)
nov 94 - jun 96 Research assistant Institute of Transportation,
Traffic, Highway- and Railway-Engineering (IVT)
at ETH Zurich
may 95 - jul 99 PhD-student Computer Science Department
at ETH Zurich