Spatial Modelling with Geometric Constraints

B. Seybold F. Metzger G. Ogan J. Bathelt F. Collenberg
J. Taiber K. Simon M. Engeli *

Abstract

In recent years constraint-based modelling has become an important method for describing dependencies within a CAD-model. In this paper, we present a constraint-based approach for the *assembly*, the spatial joining of separate rigid bodies. It is mainly influenced by KRAMER's rule-based state automaton, which is an appropriate concept for solving the *geometric constraint solving problem* (GCSP) efficiently. We adapt and enhance his method which enables us to treat a larger class of problems and to compute all solutions for a particular problem. By eliminating unnecessary calculations from the solving process, performance is improved. The implemented prototype is object-oriented, reflecting the underlying problem structure, and open to further extensions. It is designed to be an important subsystem of a professional CAD/CAM-system.

1 Introduction

For an engineer, the design of a mechanism appears as a set of constraints which have to be satisfied in order to fulfil the requirements of manufacturability, assembly and product specifications (cf [3]). Until recently, the design engineer was supported by the CAD-system in the explicit definition of geometry and its visualisation. Modern systems can also maintain a set of constraints and provide routines for the calculation of implied solutions. The focal point of this project is the spatial joining of separate rigid bodies, the *assembly*. Due to the properties of the design process, a constraint-based assembly subsystem should meet the points stated below:

1. Since the creation of a design is interactive and iterative, the set of constraints should be editable in an easy way, which requires frequent and therefore efficient solving.

2. Neither the solving process nor the solution set should be influenced by the constraint insertion and deletion sequence.

3. Usually a design process involves trial and error and may include failures. In order to detect errors and to react to failures, the feedback must be specific and understandable. Of particular interest are under- and overconstrained situations as well as the degree of redundancy.

4. In order to cope with the high complexity of a mechanism, it should be possible to handle subsets of constraints separately in accordance with its hierarchical structure.

*{seybold, simon}@inf.ethz.ch, {metzger, ogan, engeli}@iwf.bepr.ethz.ch
Research supported in part by Swiss Government, KTI 2726.1.

The requirements above are difficult to achieve with algebraic or iterative general purpose solvers such as PrologIII [2] or CLP(R) [7]. Because both methods are based on special input formats, part of the original information is lost for the solving process. Therefore they cannot provide error reports which are of sufficient help. Furthermore, algebraic solvers are inefficient and are useful only for small sets of constraints. Even successful iterative solvers, such as MOBILE [5], have specific requirements for the consistency of the set of constraints.

In [8] KRAMER suggested a constraint-based approach for the geometric constraint solving problem (GCSP). His method is called the *degrees of freedom analysis* and has several major advantages over previously existing techniques (eg numerical or symbolical methods), as the author convincingly pointed out in [8] and [9].

Our concept is a functional extension of KRAMER's method applied to the assembly problem and meets the requirements above. Geometric shape design is beyond the scope of this paper, even though it is an equally suitable area for a similar approach (cf [1], [6] only for the planar problem).

In Section 2 we discuss the terms used and the basic data structures. Section 3 focusses on the design of constraint types and its impact on the solving routines. In Section 4 and 5 the extensions to KRAMER's algorithm are described. Section 6 is dedicated to the implementation of the prototype and the reference example. Conclusions and an outlook are presented in Section 7.

2 Terminology

A *state* describes the mobility of one rigid body relative to another or relative to its environment respectively. This mobility can be classified by means of the *degree of freedom* (DOF). In simpler cases the DOF can be split into a number of translational (TDOF) and rotational (RDOF) degrees. But there are states which are too complex to be classified by simply counting the DOF (eg a rotation with coupled translation). A state with a TDOF and b RDOF is labeled $aTbR$. A spatially totally unconstrained body has 3T3R.

If there exists mobility between two parts, they are called *underconstrained*; if the constraints restrict more than 6 DOF, they are *overconstrained*. Underconstrained systems are important for kinematics, where the design intent is to have remaining DOF. They also occur when the design is not yet finished. Sometimes the fact that a part is underconstrained with respect to its environment is even irrelevant (eg a rivet with remaining RDOF). For overconstrained situations, the engineer is interested to know the the degree of redundancy and whether the assembly is still consistent. The problem with consistent overconstrained mechanisms is that the computer may calculate the result to an accuracy which cannot be reproduced in the mounting. This therefore represents a risk which can lead to a jammed mechanism.

The underlying input data structure of all algorithms acting on the assembly is a graph containing information about the parts and their relationship. This *constraint graph* consists of the following elements:

- The *elementary parts* build the nodes of the graph. There are several types of nodes each encapsulating a different geometry, such as a point, a plane or an axis.

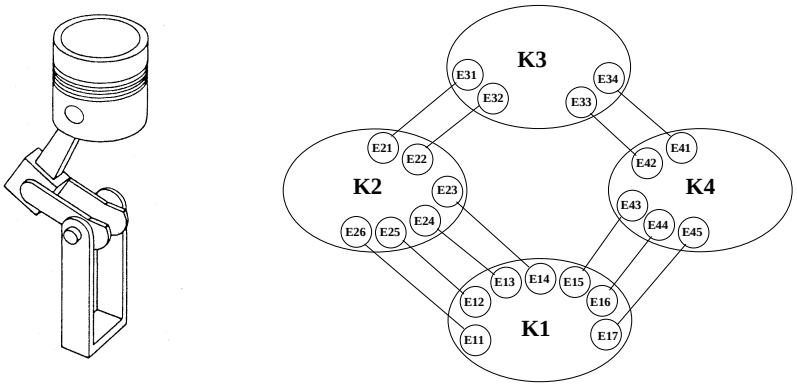

Figure 1: A piston engine.

- A *constraint* is defined between two elementary parts (in principle this can be extended to more than two). Additional parameters are attached. Constraints form the edges of the graph.
- *Components* group elementary parts, thus forming a rigid body. A component can be understood as a subassembly which has already been solved. This can be used for recursively-nested assemblies.

As an example of such a graph we consider a simple slider-crank mechanism, as shown in Figure 1. Modelled as plain 2D-mechanism, it corresponds to a RRRP-Loop[1]. Our system can model and solve it as an RRCC-Loop[2], which corresponds to the 3D-mechanism. The associated constraint graph consists of four components (K1, K2, K3 and K4), whose elementary parts (E11, E12, E21 ...) are connected by various constraints.

The situation still has remaining DOF, since of course the piston is designed to be mobile. The remaining freedom f can be calculated according to the Kutzbach-Grübler criterion [4] for the RRRP-Loop as follows:

$$f = -3n_L + \sum_{i=1}^{n_G} f_{G_i} = -3 + \sum_{i=1}^{4} 1 = 1$$

where n_L is the number of loops, n_G the number of joints and f_{G_i} the number of DOF of the link G_i. The complexity of practical examples is considered in Section 6.

3 Constraint design issues

The power and usability of a constraint solver depends heavily on the selection of the allowed constraint types and their associated solving routines. It is very important to consider the dual task of a constraint:

- Constraints are used to describe functionality or design intent.
- They encapsulate the minimal data which must be observed simultaneously to solve the GCSP analytically.

[1] Revolute, Revolute, Revolute and Prismatic joint
[2] Revolute, Revolute, Cylindrical and Cylindrical joint

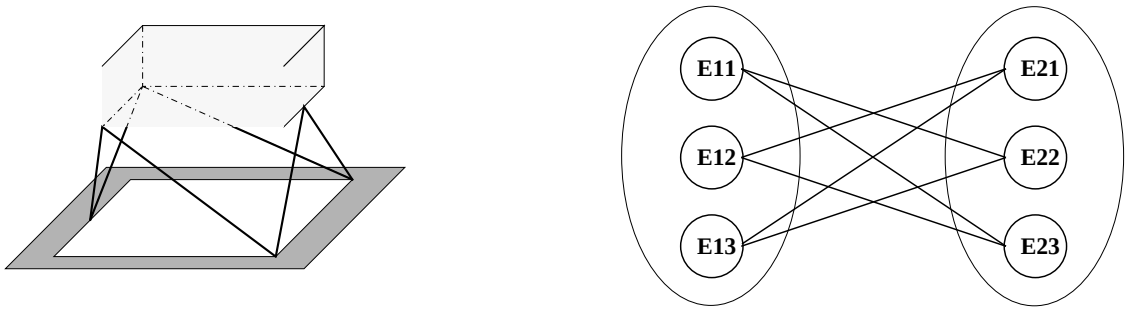

Figure 2: A flight simulator and its associated constraint graph. Even the combination of simple constraints (like the distance between two points) can make the solution set, and especially its calculation, very complicated.

Each of these two tasks implies conflicting arguments for the choice of the constraint types. The main considerations in the difficult determination of a suitable set of constraint types are the following:

- When the engineer is provided with more variety, he can model his ideas more exactly and directly. Often a subassembly can only be understood as a whole, which means that it must be described by a complex constraint (eg the universal link). Unfortunately, there are many such situations, which means that it is not at all feasible to take them all into consideration.
- If the number of constraint types is too big, there is a danger that the clarity of the design is lost.
- In isolation, each constraint is quite simple to solve. The difficulty of the solving process arises from their combination. Therefore, certain situations (eg the flight simulator in Figure 2) require the existence of complex constraints in order that a solution can be found.
- The complexity and the size of the implementation grow rapidly with the number of constraint types.
- The concatenation of constraints (cf Section 5) requires some degree of completeness of the constraint type set.

These arguments led to the conclusion that it is a good idea to keep the number of constraint types as small as possible, but to retain the possibility for extensions if needed. The crucial point here is the integration of the new constraint types and their solving mechanisms without disturbing the solver system. This led directly to a restricted view of the constraints, which is used to formulate the interface between the data structure (Section 2), the global integration (Section 5) and the local solving routines (Section 4). It was also very useful during the implementation with different teams (Section 6).

The choice of states results directly from the choice of constraint types. Basically all the states resulting from a combination of constraints must be considered. As our system provides no limitations in the combination of constraints, these can be quite numerous. However, the amount of work in this part was significantly reduced by using *lazy evaluation*, as introduced in Section 4.

coincident(P_1, P_2)	Point P_1 and P_2 must be coincident.
in-line(P, A)	Point P must be on axis A.
in-plane(P, E, d)	Point P must have distance d to plane E.
parallel-z(V_1, V_2, B)	The vectors V_1 and V_2 must be parallel (B true) or anti-parallel (B false).
offset-x(O_1, O_2, β)	The z-vectors of the two 'orientations' (describing a rotation) should be parallel and the angle of the x-vector of O_2 relative to the x-vector of O_1 should be β viewed from the z-vectors.
offset-z(V_1, V_2, α)	The angle between the vectors V_1 and V_2 should be α.
identical(M_1, M_2)	The markers M_1 and M_2 must be equal with regard to position and orientation.

Table 1: Important base constraints.

The constraint types are conceptually divided into two main groups:

- *base constraints*: They are undecomposable entities to which the solving algorithm is bound. A subset of the chosen base constraints is shown in Table 1. The base constraints represent the level of implementation.

- *compound constraints*: The calculation of a compound constraint can be reduced to that of base constraints, but the semantic is more complex than just combining them. Eg the revolute joint can be reduced to a coincident and parallel-z; but one has to know that these constraints act together in order to understand the concept of the link. All compound constraints are directly presented to the engineer.

The major part of the base constraints consists of the constraints described in [9]. Some base constraints – like semi-algebraic constraints which divide the space for selecting one of a discrete number of solutions – and compound constraints are added according to the requirements of the engineers. The set of constraint types is likely to be increased in the future.

4 The local solver

In this and the next section we describe the solution to the GCSP. First we consider the problem of only two components (*two-body-problem*) which are linked by constraints between several of their elementary parts. This level of the solving mechanism is called a *local solver*. How its ability is integrated into the global solver is discussed in Section 5.

The aim of solving a geometric constraint problem is to reduce the DOF while satisfying the constraints, which often results in a state of zero mobility (0T0R). Initially, each component of an assembly or mechanism is assumed to be free in its translations and rotations in space with respect to any other component. The final DOF of each component depends on the constraints that are to be satisfied in relation to another component.

The task of the local solver is to fulfil the given constraints between two components in order to find the resulting mobility. At the beginning of the local constraint satisfaction process, the mobility of one component in relation to the other is completely unconstrained. This fact is represented by the so-called *initial state*. All constraints are

applied sequentially to this state in any order. The local solver terminates when either there are no further constraints or the constraint set cannot be satisfied.

The evaluation process can be formalised as follows: From the starting state s_0 the constraints $c_1, \ldots, c_n$ induce the final state s_n (which need not to be 0T0R) by passing through the intermediate states $s_1, \ldots, s_{n-1}$.

$$s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \xrightarrow{c_3} \cdots \xrightarrow{c_n} s_n$$

As proven in [9], the constraints can be applied to the initial state in an arbitrary order. In particular, if two different sequences of the same steps both yield a result, then these results are the same. But it is not guaranteed that every sequence will lead to the computation of a result.

Our approach is based on KRAMER's method for the GCSP. The main extension is the possibility of delaying the satisfaction of constraints to subsequent state transitions which leads to major improvements:

- It is guaranteed that the algorithm does not fail due to the chosen constraint evaluation sequence. The system collects the constraints until it detects a pattern which leads to a (simple) state transition and thus avoids unnecessary dead-ends (cf Figure 3).

- Simplified calculation is possible by applying a constraint transformation (Figure 4) or a constraint combination (eg an in-line and an in-plane constraint which refer to the same point can be combined to form a coincidence constraint between this point and the intersection of the line and the plane).

- The system is able to describe all states defined by any combination of constraints, as they do not have to be mapped to a discrete state hierarchy with given TDOF and RDOF as in KRAMER's method. More complex states are described by simply enumerating the constraints (manipulated by the simplification rules) which must be applied, thus presenting the full information for the calculation of later transitions.

- At the end of the constraint evaluation, the remaining delayed constraints are solved by algebraic means, potentially leading to multiple solutions (Figures 3, 4). These algebraic routines, which are not considered in KRAMER's method, can be time consuming or numerically unstable due to complex computations, therefore they are only applied to the reduced situation at the end.

- Significantly fewer state types are distinguished which leads to less code with respect to KRAMER (cf Section 6) as the number of routines is approximately the number of constraint types times the number of states.

To deal with multiple solutions, KRAMER provides an input variable for the last constraint satisfaction procedure, which allows the last choice to be determined by assigning it TRUE or FALSE. But the choice criteria are rather arbitrary: The question arises which solution corresponds to TRUE. The case where only one of multiple solutions remains (the others disappearing due to later contradiction with other constraints) cannot be dealt with directly, but must be searched for by trying all possible variable values. The new method allows solving on the global level to be continued with a state containing

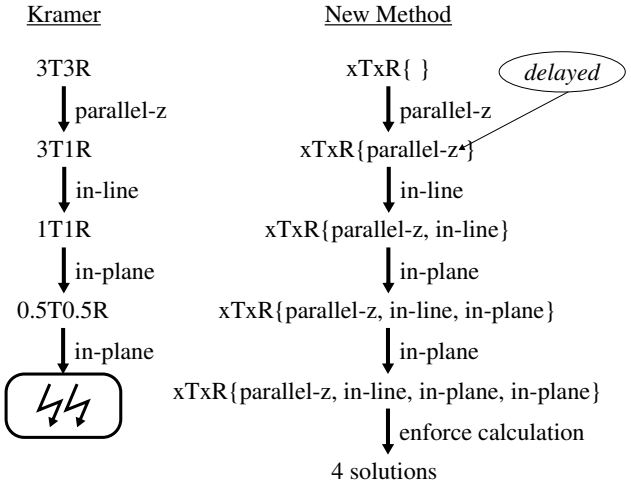

Figure 3: This example consists of two bodies which are constrained by a parallel-z, an in-line and two in-plane constraints. The planes of both the in-plane constraints and the point of the in-line constraint are attached to same body. Even though KRAMER provides pseudo-code for the evaluation of an in-plane constraint against the state called 0.5T0.5R (cf [9] pp. 230), detailed analysis of the pseudo-code shows that inconsistent situations result. Note that in more practical examples, the situation before the enforced calculation is typically more reduced.

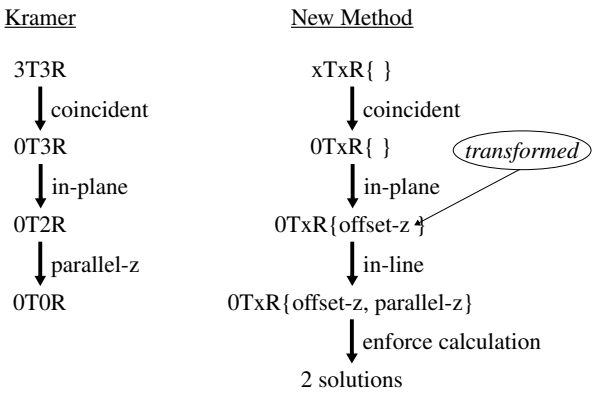

Figure 4: An example sequence with constraint transformation and enforced calculation, resulting in two solutions. The in-plane constraint is transformed to an offset-z as the current state only allows rotation. At the end, the algebraic routines produce two solutions.

all solutions simultaneously. In Figure 4, an example is given where KRAMER has to perform a complex state transition (0T3R to 0T2R solving an in-plane).

The DOF restriction implied by a constraint does not necessarily lead to a full reduction because a particular DOF could be restricted already. A constraint can be partly or totally redundant or even contradict the state. It is very important to keep track of the redundancy information and to present it to the engineer.

The following cases can occur while trying to satisfy one constraint in the two-body-problem:

- The constraint contradicts the current state. Then the next state is of type *illegal*, all temporal states from now on are also illegal, so the computation is terminated.
- The constraint is consistent and (partly) redundant to the current state, meaning that a full DOF restriction is not implied, but all constraints can still be satisfied simultaneously. The next state is calculated (it could even be of the same type) and the redundancy is increased.
- The constraint can be directly satisfied by changing the current state type. The calculation of a new legal position is performed.
- The constraint is not directly evaluated. It is delayed, possibly combined with another (the combination is applied recursively), and considered later.

5 The global solver

In this section we focus on the combination of the local solving routines for the two-body-problem described in the last section. The global solver takes care of choosing the appropriate solving mechanism as well as of grouping components into blocks.

It is essential to state that the addition or deletion of local solving routines does not change the behavior of the global solver but only how often a 0T0R is detected.

A very important property of a GCSP is that the whole assembly still has all DOF with respect to the rest of the world. This allows the position and orientation of one part to be chosen without loss of generality. This part is considered fixed and is called the *base component* BC.

The solving algorithm runs in three passes as follows:

Pass 1 (see algorithm 5.1) Constraints are transformed into states as described in Section 4: Choose a base component (line 1), fix it and transform all adjacent constraints into states (line 5). If a state 0T0R occurs, the corresponding component is united with the base component (line 6 - 8) and the adjacent constraints of the new component are also considered in the next step. If no more groupings are possible, a new base component is chosen (line 9 - 11). Pass 1 is finished when all constraints have been considered.

The Global Solver, Pass 1

```
(1)   BC ← some component
(2)   CC ← BC
(3)   progress ← true
(4)   while progress do
(5)         expand CC
(6)         if there is a component with 0T0R with respect to BC then
(7)               unite BC and this component
(8)               CC ← this component
(9)         else if there is another unexpanded component then
(10)              BC ← this component
(11)              CC ← this component
(12)        else
(13)              progress ← false
(14)        end
(15)  end
```

Algorithm 5.1

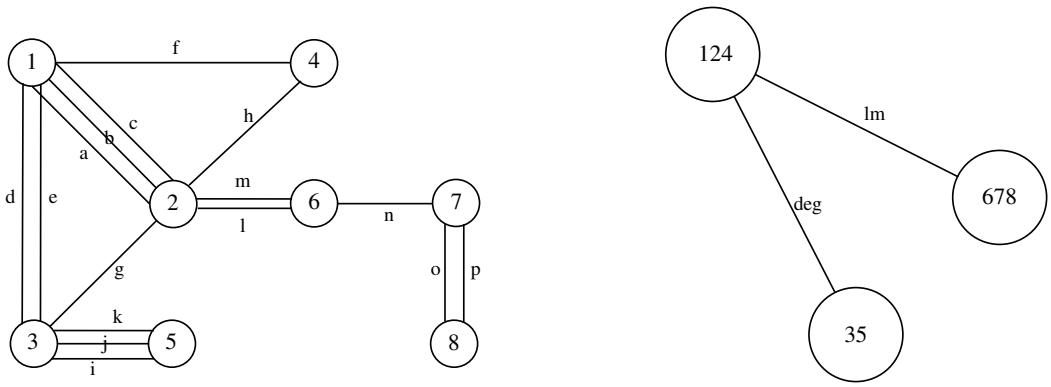

Figure 5: Example for pass 1: We take component 1 as the base component and transform the constraints a, b, c, d, e and f into states. Let the state between 1 and 2 be 0T0R. Then 2 is united with 1 (now called block 12) and the constraints h, g, l and m are transformed into states between 12 and the components 3 and 4 respectively. Let the state between 12 and 4 be 0T0R as well. Then 4 is added to the block 12. The components 3 and 6 have remaining DOF to the block 124. All constraints adjacent to the base block are considered and a new base component is chosen. Let 3 be this new base component. The constraints i, j and k are examined and lead to a grouping. Now the base component is switched again (to 6) and the remaining constraints are transformed. This leads to a grouping. The resulting situation consists of three blocks 124, 35 and 678 and two states, one between 124 and 35 (formed by constraints d, e and g) as well as one between 124 and 678 (a combination of l and m). As a next step pass 2 is performed.

The Global Solver, Pass 2

```
(1)  progress ← true
(2)  while progress do
(3)      if there occurred a state 0T0R then
(4)          unite the two blocks
(5)      else if there is a state A with delayed constraints then
(6)          calculate(A)
(7)      else if there are parallel states A, B then
(8)          combine(A, B)
(9)      else if there is an unconsidered triangle with states A, B, C then
(10)         combine(concatenate(A, B), C)
(11)         progress ← "result is non-trivial"
(12)     else
(13)         progress ← false
(14) end
```

Algorithm 5.2

Pass 2 (see algorithm 5.2) The situation is the following: There are several blocks interconnected by states. Four operations can now be performed to simplify. First it is checked whether a state 0T0R has occurred, which then results in uniting two blocks (line 3 - 4). The second operation is to enforce calculation of delayed constraints stored in the state (line 5 - 6). The third is to combine two states which are defined between the same blocks (line 7 - 8). The last and most difficult strategy is to look for triangles of states (the case of greater polygons is not shown in the algorithm) and to concatenate them (line 9 - 11), meaning that if there exists a state A between the bodies K1 and K2, a state B between K2 and K3 and a state C between K1 and K3, then a new state between K1 and K3 is computed out of A and B and combined with the state C.

Pass 3 After calculating the states as precisely as possible, the blocks must be moved into a position which fits their states. This is achieved by starting at an arbitrary block and moving the neighbours according to the states in breath-first order. However, if the resulting situation after pass 2 is a cyclic graph, all states cannot be satisfied simultaneously because moving is done independently and could destroy earlier considered states. Therefore it has to be decided which states are to be satisfied. For this purpose each state gets a priority, and a minimal spanning tree for each compound component is computed.

Basically the constraints could be considered in any order, but by applying source-oriented grouping, blocks can be built at an early stage which reduces work later, since the situation is considerably centralised. If the base component is chosen as the main part intended by the engineer, the centralisation is almost ideal as this takes into account the structure of the design ([8] 4.1.3, p.342).

The first pass of the algorithm considers every constraint just once. Because they are stored in linear lists and the calculation of a transformation takes constant time, work load is of the order of the number of constraints. Every component is examined just once, therefore the time consumed by the first pass is $O(|\mathcal{K}| + |\mathcal{C}|)$, where $\mathcal{K}$ is the set of all components and $\mathcal{C}$ the set of all constraints.

The amount of work for the second pass is not easy to estimate as the size of the remaining graph is not known exactly. But there is strong evidence that the situation does not grow too large since the algorithm follows the intention of the engineer, who tends to model in a hierarchical and structured way. This implies early grouping and small input for pass 2.

A minimal spanning tree can be calculated in $O(n \log n + m)$ time with the PRIM algorithm, where n is the number of remaining blocks and m is the number of remaining states after pass 2.

6 Implementation

The concepts outlined in this paper resulted in the implemention of a prototype, which is designed to form the kernel of a professional assembly module. In this section we describe some aspects concerning this prototype.

To define a typical real world problem, the following facts were taken as limits. An assembly consists of about 100 components. A rigid link between two components can be described by two or three constraints. This leads to an approximate bound for the number of constraints of about 300 per assembly. This value can increase in case of intended redundancy and decrease in case of mobile parts, but an assumption of 300 to 400 constraints per assembly seems reasonable.

As a reference example we consider the 7-cylinder radial type engine shown in Figure 6. It consists of about 40 components, 100 compound constraints resulting in 200 base constraints and 200 elementary parts [3]. The constraint graph has a star-like structure. The centre is surrounded by seven slider-crank mechanisms, which are more complex versions of the one in Figure 1, but conceptually have the same cycles and remaining DOF.

The system architecture is shown in Figure 7. The different modules are directly implied by the concept structure, namely the separation of global and local solver and the splitting between API and solver. The implementation in C++ has been done by several people mainly because of timing reasons, but also to enforce the development of a good interface. The limited state set (cf Section 4) leads to smaller code which made development faster and debugging easier.

The prototype can solve an assembly with 100 components, 300 constraints and 300 elementary parts on a SPARC 5 in under 1 second, so results can be obtained fast enough to allow the user to work interactively.

[3] It is a good idea to model every constraint with different elementary parts even if they encapsulate partly the same geometry. This facilitates later changes.

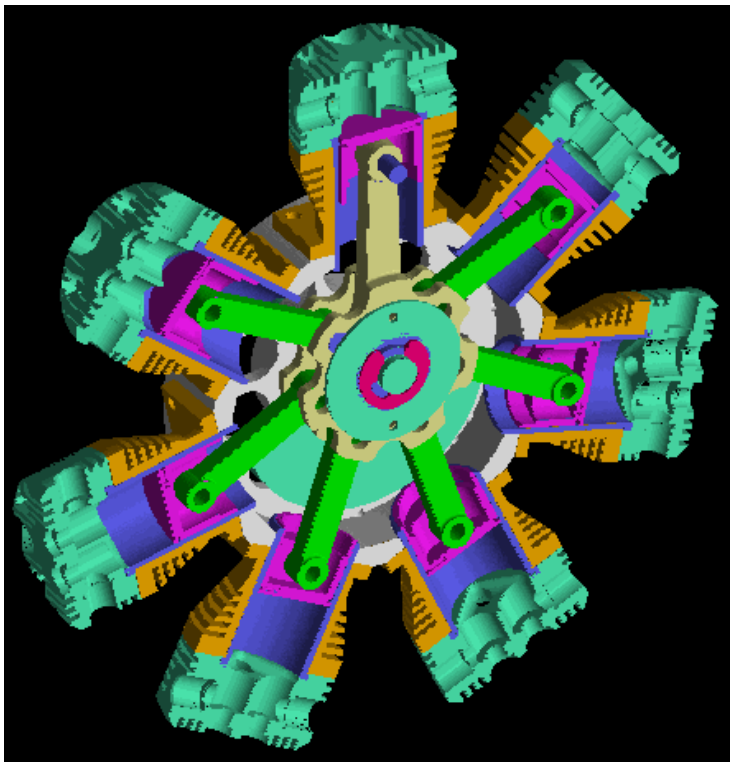

Figure 6: Reference example: 7-cylinder piston engine.

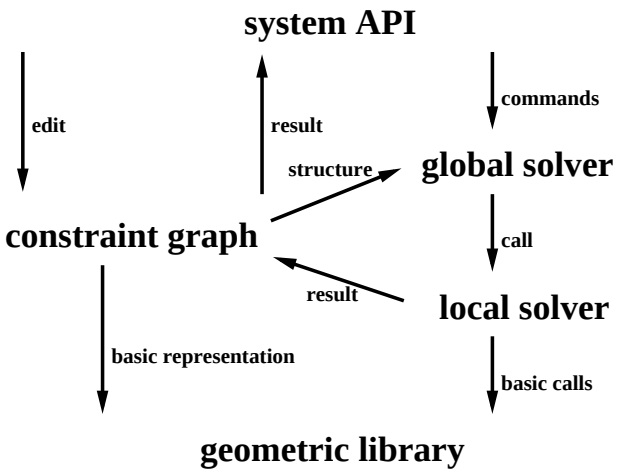

Figure 7: System architecture

7 Conclusions

KRAMER proposed a concept for solving the GCSP efficiently by using a specialised algorithm, the *degrees of freedom analysis*. The approach presented in this paper extends this concept significantly, which allows a larger class of problems to be addressed. We also developed the concept of putting the solving methods in the context of a commercial CAD-system. The resulting system satisfies the requirements of a modern constraint-based subsystem, as described in Section 1, and successfully solves real-world problems.

We intend to continue the project in the direction of new points of interest, some of which are listed below.

- The mechanism resulting from the solution should be animated within the limits of the remaining DOF. Additionally, one could obtain kinematic results like forces and deformation.
- The base concept of a constraint graph could be extended beyond assemblies to the modelling of bodies. This would imply additional constraint types including functional dependencies of values or constraints regarding symmetries.
- The number of routines for solving loops (concatenate) could be increased in order to solve more complex linkings.
- Mechanical tolerances could be added.

8 Acknowledgments

We would like to thank Jakob Magun for the important work he did at the beginning of this project, and all the people and institutions who supported us with their time, interesting ideas, useful criticism, necessary motivation and essential money.

References

[1] W. Bouma, I. Fudos, C.M. Hoffmann; *A Geometric Constraint Solver*; CAD 27, 1995, pp. 487-501.

[2] A. Colmerauer; *An Introduction to Prolog III*; Communications of the ACM, 33(7):69, 1990.

[3] K. Ehrlenspiel; *Integrierte Produktentwicklung*; Carl Hanser Verlag München Wien, 1995.

[4] M. Hiller; *Mechanische Systeme*; Springer Verlag, Berlin, 1983; ISBN 3-540-12521-3.

[5] M. Hiller, A. Kecskeméthy; *A Computer-Oriented Approach for the Automatic Generation and Solution of the Equations of Motion for Complex Mechanisms.*; Proceedings of the 7th IFToMM-Congress, Sevilla, Spain, Pergamon Press, 1987, pp. 425-430.

[6] C.M. Hoffmann, R. Juan; *Erep: An Editable High-Level Representation for Geometric Design and Analysis*; Geometric and Product Modeling, North Holland, 1993.

[7] J. Jaffar, S. Michaylov, P.J. Stuckey, R.H.C. Yap; *The CLP(R) Language and System*; ACM Transactions on Programming Languages and Systems, Vol. 14, no. 3, 1992, pp. 339-395.

[8] G.A. Kramer; *A Geometric Constraint Engine*; Artificial Intelligence 58 (1992), pp. 327 - 360.

[9] G.A. Kramer; *Solving Geometric Constraint Systems*; MIT Press, 1992; ISBN 0-262-11164-0.