Planning from First Principles for Geometric Constraint Satisfaction

Sanjay Bhansali
School of EECS
Washington State University
Pullman, WA 99163
bhansali@eecs.wsu.edu

Glenn A. Kramer
Enterprise Integration Technologies
459 Hamilton Avenue
Palo Alto, CA 94301
gak@eit.com

Abstract.

An important problem in geometric reasoning is to find the configuration of a collection of geometric bodies so as to satisfy a set of given constraints. Recently, it has been suggested that this problem can be solved efficiently by symbolically reasoning about geometry using a *degrees of freedom* analysis. The approach employs a set of specialized routines called *plan fragments* that specify how to change the configuration of a set of bodies to satisfy a new constraint while preserving existing constraints. In this paper we show how these plan fragments can be automatically synthesized using first principles about geometric bodies, actions, and topology.

Introduction

An important problem in geometric reasoning is the following: given a collection of geometric bodies, or *geoms*, and a set of constraints between them, find a *configuration*, i.e. position, orientation, and dimension, of the geoms that satisfies all the constraints. Solving this problem is an integral task for constraint-based sketching and design, geometric modeling for computer-aided design, kinematic analysis of robots and other mechanisms, and describing mechanical assemblies.

In [3] Kramer suggests that general-purpose constraint satisfaction techniques are not well suited to problems involving complicated geometry. He describes a system called GCE that uses an alternative approach called *degrees of freedom analysis*. This approach is based on symbolic reasoning about geometry and is more efficient than systems based on algebraic equation solvers. GCE employs a set of specialized routines called *plan fragments*, that specify how to change the configuration of a geom using a fixed set of operators and the available degrees of freedom, so that a new constraint is satisfied while preserving the geom's prior constraints. This approach is canonical: at any point one may choose any constraint to be satisfied without affecting the final result. The resulting algorithm has polynomial time complexity and is more efficient than general-purpose constraint satisfaction algorithms.

Since the most interesting part of problem-solving is performed by plan fragments, the success of this approach depends on one's ability to construct a complete set of plan fragments meeting the canonical specification. In this paper we describe how to automatically generate plan fragments using first principles about geoms, actions, and topology.

Our approach is based on planning. Plan fragment generation becomes a planning problem by considering the various geoms and invariants on them as describing a *state*. Operators are actions, e.g.*rotate*, that change the configuration of geoms, thereby violating or achieving some constraint. An *initial* state is specified by the set of existing invariants on a geom and a *final* state by the additional constraints to be satisfied. A plan is a sequence of actions that when applied to the initial state achieves the final state.

With this formulation, one could presumably use a classical planner, e.g. STRIPS [1], to automatically generate a plan-fragment. However, plan fragment actions are parametric operators with a real-valued domain. Thus, the search space consists of an infinite number of states. Our approach uses *loci information* (representing a set of points that satisfy some constraints) to reason about the effects of operators and thus reduces the search problem to a problem in topology, involving reasoning about the intersection of various loci.

Another issue to be addressed is the *frame problem*: how to determine what properties or relationships do not change as a result of an action. A typical solution is to use the assumption that an action does not modify any property or relationship unless explicitly stated as an effect of the action. Such an approach works well if one knows *a priori* all possible constraints or invariants that might be of interest and relatively few constraints get affected by each action - which is not true in our case. We use a novel scheme for representing effects of actions. It is based on reifying actions in addition to geoms and invariant types. We associate, with each pair of geom and invariants, a set of actions that can be used to achieve or preserve that invariant for that geom. Whenever a new geom or invariant type is introduced the corresponding rules for actions that can achieve/preserve the invariants are added. Since there are many more invariant types than actions in this domain, this scheme results in simpler rules. A unique feature of our work is the use of geometry-specific matching rules to determine when two or more general actions that achieve/preserve different constraints can be reformulated to a less general action.

Another shortcoming of using a conventional planner is the difficulty of representing conditional effects of operators. In GCE an operation's effect depends on the type of geom as well as the particular geometry. E.g, the action of translating a body to the intersection of two lines on a plane normally reduces the body's translational degrees of freedom to zero; however, if the two lines happen to coincide then the body still retains one translational degree of freedom and if the two lines are parallel but do not coincide then the action fails. Kramer calls such situations *degeneracies*. One approach to handling degeneracies is to

use a reactive planner that dynamically revises its plan at run-time. However, this could result in unacceptable performance in many real-time applications. Our approach makes it possible to pre-compile all potential degeneracies in the plan. We achieve this by dividing the planning algorithm into two phases. In the first phase a skeletal plan is generated that works in the normal case and in the second phase, this skeletal plan is refined to take care of singularities and degeneracies. This approach is similar to the idea of refining skeletal plans in MOLGEN [2] and the idea of critics in HACKER [4] to fix known bugs in a plan. However, the skeletal plan refinement in MOLGEN essentially consisted of instantiating a partial plan to work for specific conditions, whereas in our method a complete plan which works for a normal case is extended to handle special conditions like degeneracies.

A Plan Fragment Example.

The following is a simple example of a plan fragment specification that is also used to illustrate our approach. The example is illustrated in Figure 1.

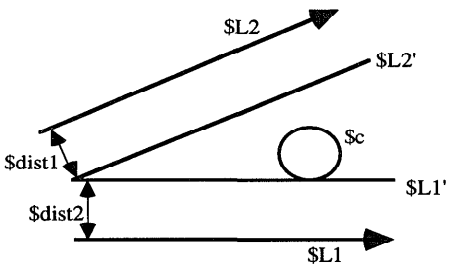

Figure 1. Example problem (initial state)

Geom-type: circle
Name: $c
Invariants: (fixed-distance-line $c $L1 $d1
$BIAS_COUNTER_CLOCKWISE)
To-be-achieved: (fixed-distance-line $c $L2 $dist2
$BIAS_CLOCKWISE)

In this example, a variable radius circle *$c*[1] has a prior constraint specifying that the circle is at a fixed distance *$dist1* to the left of a fixed line *$L1* (or alternatively, the line, *$L1* is tangent in a counterclockwise direction to the circle). A new constraint to be satisfied is that the circle be at a fixed distance *$dist2* to the right of another line *$L2*.

To solve this problem, three different plans can be used: (a) translate the circle from its current position to a position such that it touches the two lines $L2' and $L1' shown in the figure. (b) scale the circle while keeping its point of contact with $L1' fixed, so that it touches $L2' (c) scale and translate the circle so that it touches both $L2' and $L1'.

Each of the above plan fragment would be available to GCE from a plan-fragment library. Note that some of the plan fragments would not be applicable in certain situations. For example, if $L1 and $L2 are parallel, then a single translation can never achieve both the constraints, and plan-fragment (a) would not be applicable. We show how each of the plan-fragments can be automatically synthesized by reasoning from more fundamental principles.

[1]We use the following conventions: symbols preceded by $ represent constants, symbols preceded by ? represent variables, expressions of the form (>> parent subpart) denote the subpart of a compound term, parent.

Overview of System Architecture

Figure 2 depicts the architecture of our system showing the various knowledge components and the plan generation process. The knowledge represented in the system is broadly categorized into a Geom Knowledge-base that contains knowledge specific to geometric entities and a Geometry Knowledge-base that is independent of particular geoms and can be reused for generating plan fragments for any geom. The geom-specific knowledge consists of six knowledge components:

1. Actions. These describe operations that can be performed on geoms. In the GCE domain, three actions suffice to change the configuration of a body to an arbitrary configuration: (**translate** *g v*) which denotes a translation of geom *g* by vector *v*; (**rotate** *g pt ax amt*) which denotes a rotation of geom *g*, around point *pt*, about an axis *ax*, by an angle *amt*; and (**scale** *g pt amt*) where *g* is a geom, *pt* is a point about which *g* is scaled, and *amt* is a scalar.

2. Invariants. These describe constraints to be solved for the geoms. The initial version of our system has been designed to generate plan fragments for a variable-radius circle on a fixed workplane, with constraints that are distances between these circles and points, lines, and other circles on the same workplane. There are seven invariant types to represent these constraints. An example of an invariant is: (**Fixed-distance-point** *g pt dist bias*) which specifies that the geom *g* lies at a fixed distance *dist* from point *pt*; *bias* specifies whether *g* lies inside or outside a circle of radius *dist* around point *pt*.

3. Loci. These represent sets of possible values for a geom parameter, e.g. the position of a point on a geom. The various kinds of loci can be grouped into either a 1d-locus (described by an equation of 1 variable) or a 2d-locus (described by an equation of 2 variables). For example, *(***make-line-locus** *through-pt direc)* represents an infinite line (a 1d-locus) passing through *through-pt* and having direction *direc*. Other loci in the system include rays, circles, parabolas, hyperbolas, and ellipses.

4. Measurements. These are used to represent the computation of some function, object, or relationship between objects. These terms are mapped into a set of service routines which are called by the plan fragments. An example of a measurement terms is: (**0d-intersection** *1d-locus1 1d-locus2*). This represents the intersection of two 1d-loci. In the normal case, the intersection of two 1-d loci is a point. Singular cases occur when the two loci happen to coincide; in such a case their intersection returns one of the loci instead of a point. Degenerate cases occur when the two loci do not intersect; in such cases, the intersection is undefined. These exceptional conditions are used during the second phase of the plan generation process to elaborate a skeletal plan (see Section 3.3).

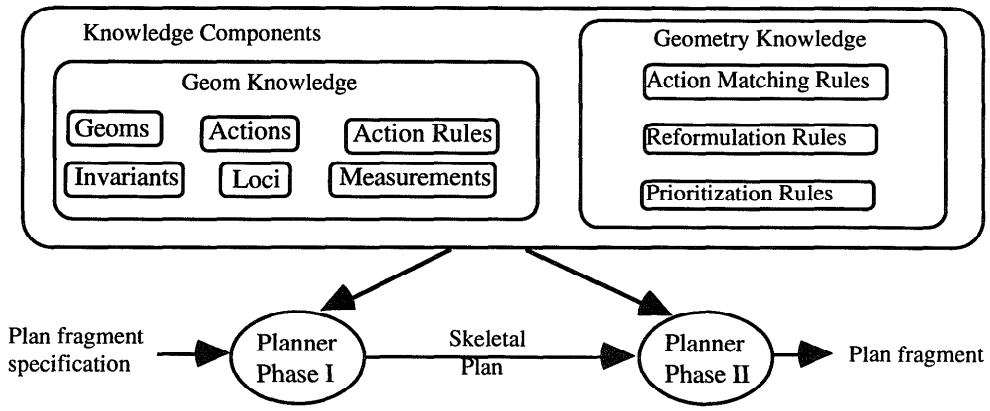

Figure 2. Architectural Overview of the Plan fragment Generator

5. Geoms. These are the objects of interest in solving geometric constraint satisfaction problems. Examples of geoms are lines, line-segments, circles, and rigid bodies. Geoms have degrees of freedoms which allow them to vary in location and size. For example, in 3D-space, a circle with a variable radius has three translational, two rotational, and one dimensional degree of freedom.

The configuration variables of a geom are defined as the minimal number of real-valued parameters required to specify the geometric entity in space unambiguously. Thus, a circle has six configuration variables (three for the center, one for the radius, and two for the plane normal). In addition, the representation of each geom includes the following: *name*: a unique symbol to identify the geom; *invariant-descriptors*: a set of rules that describe how invariants on the geom can be preserved or achieved by actions (see below); *invariants*: the set of current invariants on the geom; *invariants-to-be-achieved*: the set of invariants that need to be achieved for the geom.

6. Action Rules (Invariant Descriptors). An action rule describes the effect of an action on an invariant. The planner must know: (1) how to achieve an invariant using an action and (2) how to choose actions that preserve as many of the existing invariants as possible. In general, there are several ways to achieve an invariant and several actions that will preserve one or more invariant. The intersection of these two sets of actions is the set of feasible solutions. In our system, the effect of actions is represented as part of geom-specific knowledge in the form of action rules, whereas knowledge about how to compute intersections of two or more sets of actions is represented as geometry-specific knowledge (since it does not depend on the particular geom being acted on).

An action rule is a three-tuple (*pattern, to-preserve, to-[re]achieve). Pattern* is the invariant of interest; *to-preserve* is a list of actions that can be taken without violating the pattern invariant; and *to-[re]achieve* is a list of actions that can be taken to achieve the invariant or re-achieve an existing invariant "clobbered" by an earlier action. These actions are stated in the most general form. The matching rules in the Geometry Knowledge base are then used to obtain the most general unifier of two or more actions.

An example of an invariant descriptor, associated with variable radius circle geoms is:

pattern: (1d-constrained-point ?c (>> ?c CENTER) ?1dlocus)
to-preserve: (scale ?c (>> ?c CENTER) ?any)
(translate ?c (v- (>> ?1dlocus ARBITRARY-PT)
(>> ?c CENTER))
to-[re]achieve:(translate ?c (v- (>> ?1dlocus ARBITRARY-PT)
(>> ?c CENTER))

This descriptor is used to preserve or achieve the constraint that the center of a circle geom lie on a 1d locus. Two actions that may be performed without violating this constraint: (1) scale the circle about its center, and (2) translate the circle by a vector that goes from its current center to an arbitrary point on the 1d locus. To *achieve* this invariant only one action may be performed: translate the circle so that its center moves from its current position to an arbitrary position on the 1-dimensional locus.

The *Geometry specific knowledge* is organized as three different kinds of rules:

1. Matching Rules. These are used to match terms using geometric properties. The planner employs a unification algorithm to match actions and determine whether two actions have a common unifier. Here standard unification is not sufficient since it is purely syntactic and does not use knowledge about geometry. To illustrate this, consider the two actions: (i) (**rotate** *$g $pt1 ?vec1 ?amt1*), and (ii) (**rotate** *$g $pt2 ?vec2 ?amt2*). Each denotes a rotation of a fixed geom $g, around a fixed point about an arbitrary axis by an arbitrary amount. Standard unification fails when applied to the above terms because no binding of variables makes the two terms syntactically equal. However, knowledge about geometry allows matching the two terms to yield (**rotate** *$g $pt1 (v- $pt2 $pt1) ?amt1*), denoting a rotation of the geom around the axis passing through *$pt1* and *$pt2*. The point around which the body is rotated can be any point on the axis (here arbitrarily chosen as *$pt1*) and the amount of rotation can be anything.

2. Reformulation Rules. These are used to rewrite pairs of invariants on a geom into an equivalent pair of simpler invariants (using a well-founded ordering). Here equivalence means that the two sets of invariants produce the same range of motions in the geom.

Besides reducing the number of plan fragments, reformulation rules also help to simplify action rules. Currently all action rules (for variable radius circles and line-segments) use only a single action to preserve or achieve an invariant. If we do not restrict the allowable signatures on a geom, it is possible to create examples where we need a sequence of (more than one) actions in the rule to achieve the invariant, or we need complex conditions that need to be checked to determine rule applicability. Allowing sequences and conditionals on the rules increases the complexity of both the rules and the pattern matcher. This makes it difficult to verify the correctness of rules and reduces the efficiency of the pattern matcher.

3. Prioritizing Rules. Given a set of invariants to be achieved on a geom, a planner generally creates multiple solutions. A majority of the solutions contain redundant actions which can be easily eliminated (e.g. if there are two consecutive translations, they can be replaced by a single translation). However, after such redundant actions are eliminated, the planner may still have multiple solutions. A set of rules called *prioritizing rules* are then used to choose a preferred solution. We have identified two types of prioritizing rules:

1. Prefer solutions that subsume an alternative solution. This rule permits more flexibility in resolving degeneracies in the plan fragment later.

2. Choose the solution that minimizes a geom's motion as measured by a motion function. This rule reflects that in most applications,e.g. computer-aided sketching it is desirable to produce the least amount of perturbation to a geometric system in order to satisfy a set of constraints.

Plan Fragment Generation

The plan fragment generation process is divided into two phases. In the first phase a specification of the plan fragment is taken as input, and a planner is used to generate a set of skeletal plans. These form the input to the second phase which chooses one of the skeletal plans and elaborates it to take care of singularities and degeneracies. The output of this phase is a complete plan fragment.

Phase I

A skeletal plan is generated using a breadth-first search process. Figure 3 gives the general form of a search tree produced by the planner. The planner first tries the reformulation rules to rewrite the geom invariants into a canonical form. Next, the planner searches for actions that produce a state in which at least 1 invariant in the Preserved list is preserved or at least 1 action in the To-be-achieved (TBA) list is achieved. The preserved and achieved invariants are pushed into the Preserved list, and the clobbered or unachieved invariants are pushed into the TBA list of the child state.

The above strategy will produce intermediate nodes in the search tree which might clobber one or more preserved invariant without achieving any new invariant.

The planner iteratively expands each leaf node in the search tree until one of the following is true: (1) The node represents a solution, i.e. the TBA list is nil. (2) The node represents a cycle, i.e. the Preserved and TBA lists are identical to an ancestor node. The node is then marked as terminal and the search tree is pruned at that point. If all leaf nodes are marked as terminal, then the search terminates. The plan-steps of each of those solution nodes represents a skeletal plan fragment. When multiple skeletal plan fragments are obtained, the planner chooses one of them using the prioritizing rule described earlier and passes it to the second phase of the plan fragment generation.

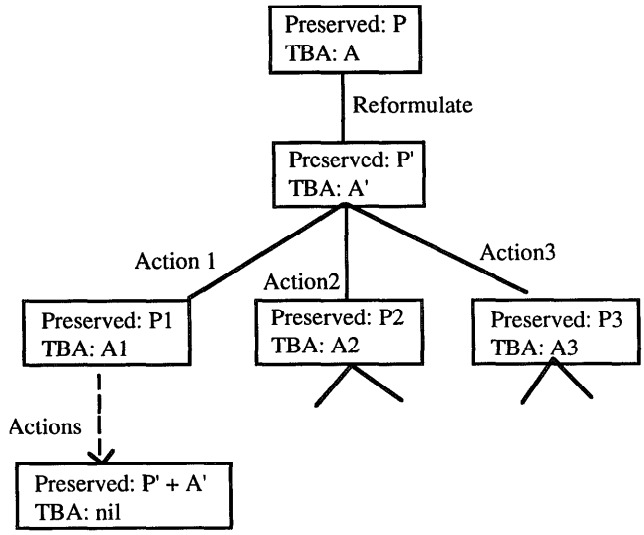

Figure 3. Overview of the search tree produced by the planner

Phase I: Example

We use the example of Section 1 to illustrate Phase I of the planner. The planner begins by trying to reformulate the constraints. It uses a reformulation rule to produce the search tree shown in Fig. 4. Next, the planner searches for actions that can achieve the new invariant or preserve the existing invariant. We only describe the steps involved in finding actions that satisfy the maximal number of constraints (in this case, two). The planner first finds all actions that achieve the *1d-constrained-point* invariant by examining the action rules associated with the variable circle geom. The action rule given in section 1 contains a pattern that matches the *1d-constrained-point* invariant. The relevant action after the appropriate substitutions is:

```
(translate $c
        (v- (>> (angular-bisector
                (make-displaced-line $L1 $BIAS_LEFT $d1)
                (make-displaced-line $L2 $BIAS_RIGHT $d2)
               arbitrary-pt)
    (>> $c center)))
```

Similarly, the planner finds all actions that will preserve the *fixed-distance-line invariant*. Matching and performing the appropriate substitutions yields the single action:

```
(translate $c (v- (>> (make-line-locus
                        (>> $c center) (>> $L1 direction))
                      arbitrary-point)
                  (>> $c center)))
```

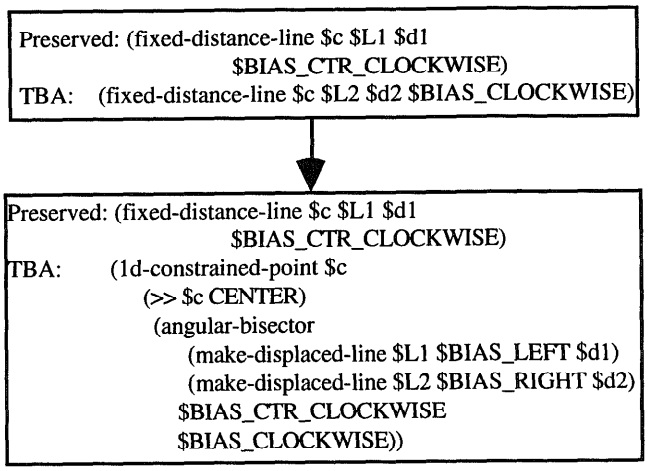

Figure 4. Search tree after reformulating invariants

Now, to find an action that both preserves the preserved invariant and achieves the TBA invariant, the planner attempts to match the preserving action with the achieving action. The two actions do not match using standard unification, but match employing the following geometry-specific matching rule:

(v- (>> $1d-locus1 arbitrary-point) $to)
(v- (>> $1d-locus2 arbitrary-point) $to)

⇓

(v- (0d-intersection $1d-locus1 $1d-locus2) $to)
(To move to an arbitrary point on two different loci, move to the intersection point of the two loci)

to yield the following action:

```
(translate $c (v- (0d-intersection
                    (angular-bisector (make-displaced-line..).)
                    (make-line-locus (>> $c center)
                                     (>> $L1 direction))
                  (>> $c CENTER)))
```

This action moves the circle to the point shown in Figure 5 and achieves both the constraints. This single one-step plan constitutes a skeletal plan fragment. There are two other actions that are generated by the planner in the first iteration, one of which achieves the new constraint but clobbers the prior invariant, while the second one moves the circle to another configuration without achieving the new constraint but preserving the prior constraint.

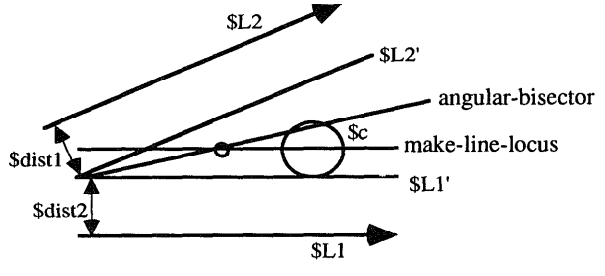

Figure 5. The O denotes the point to which the circle is moved.

After two iterations the following solutions are obtained:

(1) Translate to the intersection of the *angular-bisector* and *make-line-locus*. (2) Translate to an arbitrary point on the *angular-bisector*, followed by a translation to the intersection point. (3) Translate to an arbitrary point of *make-line-locus*, followed by a translation to the intersection point. (4) Translate to an arbitrary point on the *angular-bisector* and then scale.

At this stage the first phase of the plan fragment generation is terminated and the skeletal plan fragments are passed on to the second phase of the planner.

Phase 2: Elaboration of Skeletal Plan Fragment

The first step in Phase 2 is to select one of the skeletal plan fragments. The system begins this process by first eliminating all redundant steps in a plan.

Elimination of Redundant Plan-steps. We assume that there is only one degree of dimensional freedom for each geometric body. Under this assumption it can be proved that 1 translation, 1 rotation, and 1 scale is sufficient to change the configuration of an object to an arbitrary configuration in 3D space. Therefore, any plan fragment that contains more than 1 instance of any action type contains redundancies and can be rewritten to an equivalent plan fragment by eliminating redundant actions, or combining two or more action into a single composite action. As an example, consider the following pair of translations on a geom:

- (translate $g ?vec)
- (translate $g (v- ?to$_2$ (>> $g center)))

where *?vec* represents an arbitrary vector and *?to*$_2$ represents an arbitrary position. If *?to*$_2$ is independent of any positional parameter of the geom, then the first translate action is redundant and can be removed.

After eliminating redundant plan-steps, the system selects one of the plan fragments using the prioritizing rule described earlier, i.e. it selects one of the plan fragments that subsumes the maximal number of other plan fragments.

Least Motion. The least motion principle is meant to reduce the total perturbation in a geometric configuration when satisfying a set of new constraints. This is done by first defining a motion function, $C_{A,G}$ for each action, A, and geom type, G. For example, for a translation of a circle, the motion function, $C_{T,circle}$ could be the square of the displacement of the center of the circle from its initial to its final position. Next, we choose a motion summation function, Σ that sums the motion produced by individual actions on a geom. An example of the summation function is the normal addition operator: plus.

The total motion for a geom is computed using the summation function and the motion functions for action-geom pairs. When a plan fragment is not deterministic, the expression representing the total motion would contain one or more variables representing the ungrounded parameters of the geom. If the motion function and the summation function are chosen so that the resultant expression is analytically evaluable, then we can compute the values of

the variables that would minimize the expression representing the total motion. Substituting these values back in the plan fragment results in a plan fragment producing the least motion.

Exception Handling. Exceptional conditions occur when geometric entities are positioned so that the solution of a set of constraints results in either a solution that has extra degrees of freedom or results in no solution. To detect and characterize degeneracies, we begin by enumerating the elementary geoms and grouping possible values of these elements into equivalence classes. An equivalence class represents a set of solutions that are all associated with some kind of exception condition or a normal situation.

Whenever an expression contains a subterm whose value is instantiated at run-time, the computed value is checked for membership in one of the equivalence classes above to see if this situation represents an exception. To implement this, the representation of each elementary geom contains an additional attribute called solution-type which denotes the equivalence class of solution type for that geom. Each service routine that computes one of these geoms would return both the solution type and the solution value(s).

For an aggregate type the exceptional cases are derived by taking the cross product of the exceptional cases of each of the components of the aggregate. In general, the characterization of the value of a subterm as an exception depends on the context in which it occurs. Thus, to determine exception conditions in a plan fragment, we enumerate for each action term, all possible exception conditions of its arguments that might cause the action to fail.

The elaboration of the skeletal plan fragment consists of converting each action in the plan into a case statement. The conditions of the case statement represent one of the exception conditions of interest for the corresponding action. The body of each conditional branch represents the action to be taken to deal with the exception. Depending upon the exception, the action might involve choosing one solution from several alternatives, or generating an error message describing why the action failed.

Phase II: Example

Four skeletal plan fragments were generated in the first phase of the planner. Using the rule for eliminating redundant translations given earlier, the second and third plan fragments can be reduced to single translation plan fragments equivalent to the first plan fragment. This leaves only two distinct plan fragment solutions to consider.

Using the prioritizing rule, the system concludes that the first plan fragment consisting of a single translation is subsumed by the second plan fragment consisting of a translation and scaling. Thus, the second plan fragment is chosen as the preferred solution.

Next, the system discovers that the plan fragment is not deterministic since it contains an action that translates the circle geom to an arbitrary point on the angular-bisector. It grounds the plan by finding a fixed point on the locus based on least motion (Due to space limitations a full discussion on computing least motion is omitted). The grounded plan fragment is:

```
(translate $c (v- (compute-least-motion-points ...)
                  (>> $c center))
(scale $c (>> $c center)
          (line-point-distance $L1 (>> $c center)))
```

Next, each action in the above plan fragment is transformed into a case statement:

```
(let ((vector (v- (compute-least-motion-points ...)
                  (>> $c center))))
  (case vector.solution-type
    (zero-vector (nop) ...)
    (one.of-N (funcall #'select-one-from-N vector))
      (not-ground (print "Error:..."))
      (undefined (print "Error: ..."))
    (t (translate $c vector.value))
(let ((amt (line-point-distance $L1 (>> $c center))))
  (case amt.solution-type
      (zero (print "Zero dimension error ..."))
    (negative (print "Bias inconsistent ..."))
    (undefined (print "Error: ..."))
    (t (scale $g (>> $g center) amt.value))
```

This plan fragment is very concise, containing only the logic for solving a set of constraints; most of the other functionality that used to be part of the plan fragment is pushed down to service routines that deal with topology.

Conclusions and Future Work

We have described an automatic plan fragment generation methodology that can automatically synthesize plan fragments for geometric constraint satisfaction systems by reasoning from first principles about geometric entities, actions, and topology. We implemented the first phase of the planner and used it to synthesize plan fragments for variable-radius circle and line-segment geoms and are currently implementing Phase II of the planner.

Further work includes extending and evaluating the approach to handle more complex (e.g. 3-d) geoms and constraints and pushing the automation one level further so as to automatically acquire some types of knowledge from simpler building blocks. For example, a technique for automatically synthesizing the least motion function from some description of the geometry would be very useful.

Acknowledgment: This work was done while the first author was with the Knowledge Systems Laboratory, Stanford and the second author was at the Schlumberger Laboratory for Computer Science, Austin.

References

[1] Fikes, R. E. and Nilsson, N.J., "STRIPS: a new approach to the application of theorem proving to problem solving, Artificial Intelligence 2, 1971, pp. 198-208.

[2] Friedland, P.E., "Knowledge-based experiment design in molecular genetics", *Technical Report No. 79-771*, Computer Science Department, Stanford University, 1979.

[3] Kramer, G. A. "A Geometric Constraint Engine", *Artifical Intelligence*, 58(1-3), pp. 327-360.

[4] Sussman, G.J., *A computer model of skill acquisition*, American Elsevier: New York, 1975.