MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1573 March, 1996

SketchIT: a Sketch Interpretation Tool for Conceptual Mechanical Design

Thomas F. Stahovich

Abstract

We describe a program called SketchIT capable of producing multiple families of designs from a single sketch.

The program is given a rough sketch (drawn using line segments for part faces and icons for springs and kinematic joints) and a description of the desired behavior. The sketch is "rough" in the sense that taken literally, it may not work. From this single, perhaps flawed sketch and the behavior description, the program produces an entire family of working designs. The program also produces design variants, each of which is itself a family of designs.

SketchIT represents each family of designs with a "behavior ensuring parametric model" (BEP-Model), a parametric model augmented with a set of constraints that ensure the geometry provides the desired behavior. The construction of the BEP-Model from the sketch and behavior description is the primary task and source of difficulty in this undertaking.

SketchIT begins by abstracting the sketch to produce a qualitative configuration space (qc-space) which it then uses as its primary representation of behavior. SketchIT modifies this initial qc-space until qualitative simulation verifies that it produces the desired behavior.

SketchIT's task is then to find geometries that implement this qc-space. It does this using a library of qc-space fragments. Each fragment is a piece of parametric geometry with a set of constraints that ensure the geometry implements a specific kind of boundary (qcs-curve) in qc-space. SketchIT assembles the fragments to produce the BEP-Model.

SketchIT produces design variants by mapping the qc-space to multiple implementations, and by transforming rotating parts to translating parts and vice versa.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-91-J-4038.

SketchIT: a Sketch Interpretation Tool for Conceptual Mechanical Design

by

Thomas F. Stahovich

B.S., University of California, Berkeley (1988)
S.M., Massachusetts Institute of Technology (1990)

This is an edited version of a document originally submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

SketchIT: a Sketch Interpretation Tool for Conceptual Mechanical Design

by

Thomas F. Stahovich

Originally submitted to the Department of Mechanical Engineering
on May 29, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We describe a program called SketchIT capable of producing multiple families of designs from a single sketch.

The program is given a rough sketch (drawn using line segments for part faces and icons for springs and kinematic joints) and a description of the desired behavior. The sketch is "rough" in the sense that taken literally, it may not work. From this single, perhaps flawed sketch and the behavior description, the program produces an entire family of working designs. The program also produces design variants, each of which is itself a family of designs.

SketchIT represents each family of designs with a "behavior ensuring parametric model" (BEP-Model), a parametric model augmented with a set of constraints that ensure the geometry provides the desired behavior. The construction of the BEP-Model from the sketch and behavior description is the primary task and source of difficulty in this undertaking.

SketchIT begins by abstracting the sketch to produce a qualitative configuration space (qc-space) which it then uses as its primary representation of behavior. SketchIT modifies this initial qc-space until qualitative simulation verifies that it produces the desired behavior.

SketchIT's task is then to find geometries that implement this qc-space. It does this using a library of qc-space fragments. Each fragment is a piece of parametric geometry with a set of constraints that ensure the geometry implements a specific kind of boundary (qcs-curve) in qc-space. SketchIT assembles the fragments to produce the BEP-Model.

SketchIT produces design variants by mapping the qc-space to multiple implementations, and by transforming rotating parts to translating parts and vice versa.

Thesis Supervisor: Professor Randall Davis

Thesis Supervisor: Professor Warren P. Seering

Acknowledgments

It was a tremendous pleasure to work with Randall Davis and Howard Shrobe. They are gifted advisors, insightful teachers, generous with encouragement, and exacting in their standards.

I am extremely grateful to Warren Seering for his sage advice, careful guidance, and constant encouragement in matters both academic and professional.

A special thanks goes to George Celniker, a friend and mentor.

A special thanks also goes to Glenn Kramer who, among other things, introduced me to the AI Lab and the circuit breaker example.

To Kenji Shimada, a great friend and teacher of Japanese culture.

To Leslie Regan for truly caring about students.

To SLCS, especially Bob Young and Jahir Pabon.

To Ron Wiken for building great models of circuit breakers and such, and for being a good friend.

To Mike Wessler for writing the sketching interface.

To Kevin O'Brien for proofreading the thesis.

To my officemates, Brian Anthony, Mark Nahabedian, John Morrell, and Peng Wu: it was fun.

To my many other friends at the AI Lab: Phillip Alvelda, Mike Caine, Robert Irie, Akhil Madhani, Laurel Simmons, and Leon Wong.

To the staff and students of the AI Lab, who make it a great place to be.

Finally, thanks to my parents, Arthur and Marjorie, and my brothers, Jim, Greg, Mark, Dave, and Steve for their unconditional love, support, and encouragement. Without them, I could not have done this.

Support for this project was provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-91-J-4038.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

1.1.1 Context

Drawings have always been an important tool for engineers. The oldest known technical drawing is a ground plan of the ziggurat[1] at Ur which was built around 2100 B.C.; the drawing was made to scale and carved in stone.[2] By the fifteen century, Leonardo da Vinci had perfected many of the drawing techniques that are commonly used today, including cross hatching, sectioning, pictorial sketching, and isometric sketching.

Our work combines this time honored engineering tool, the drawing, with a modern tool, the computer, and addresses the issue of how a computer program might read, understand, and use a sketch of a mechanical device. To do this, we developed a theory about what kind of information is contained in a sketch of a mechanical device and tested this theory with an implemented computer program. The program reads a sketch of a mechanical device and from this produces working geometry as well as multiple alternative implementations for the device. The program's name is "SKETCHIT" for *Sketch Interpretation Tool*.[3]

1.1.2 The Task

A common task in mechanical design is the search for part geometries that provide a desired behavior. Parametric modelers are often useful in this search because

[1] An ancient Babylonian temple tower consisting of a lofty pyramidal structure built in successive stages with outside staircases and a shrine at the top.

[2] Thus began the long tradition of engineering specifications carved in stone.

[3] We considered using the name "etch-a-sketch" for *Engineering Transformation of Clever Hacks* using *A Sketch*, but that was taken by the Ohio Arts Company, an early adopter of graphical user interfaces.

they provide a convenient means for modifying geometry. (Parametric modelers are geometric modelers that allow the geometry to be changed via changes to the dimensions.) However, the designer is still responsible for creating the parametric model in the first place, then for deciding what changes to make. In this situation the designer is thus still responsible for creating the geometry.

In this project, by contrast, we developed a design assistant called SketchIT that assists the designer in finding a geometry that provides a specified behavior. Our program uses a description of desired behavior to transform an informal sketch of a mechanical device into what we call a "BEP-Model," a parametric model with constraints that ensure the geometry produces the desired behavior.[4] The program also generates alternative implementations for the design, each of which is represented as a BEP-Model. Because the constraints of the BEP-Model are sufficient to guarantee the desired behavior, every solution to the constraints is a working design. We have used DesignView, a commercial parametric modeler based on variational geometry, to solve the constraints of the BEP-Models produced by our program.

1.1.3 Examples

We have tested SketchIT on three design problems: a circuit breaker, a firing mechanism for a single action revolver, and a yoke and rotor mechanism (rotation of the rotor causes the yoke to oscillate). Here we use these examples to illustrate SketchIT's operation.

Circuit Breaker

Our first example concerns the design of a circuit breaker similar to those used in a home electrical system. One specific implementation of the device is shown in Figure 1-1. In normal use, current flows from the lever to the hook; current overload causes the bimetallic hook to heat and bend, releasing the lever and interrupting the current flow. After the hook cools, pressing and releasing the pushrod resets the device. (Figure 1-3a shows this behavior.)

SketchIT is a tool for conceptual design, and as such is concerned only with the *functional geometry*, the faces where parts meet and through which force and motion are transmitted. Consideration of the connective geometry (the surfaces that connect the functional geometry to make complete solids) is put off until later in the design process. The designer's task is to indicate which pairs of faces are intended to engage each other; this is done using a simple sketching tool we created. Figure 1-2 shows the stylized sketch of the circuit breaker the designer created with this sketching tool.

The designer also has to describe the desired behavior of the device to SketchIT; this is done using a state transition diagram. Figure 1-3b shows the desired behavior for the circuit breaker. Each node in the diagram is a list of the pairs of faces that

[4] "BEP-Model" stands for Behavior Ensuring Parametric Model.

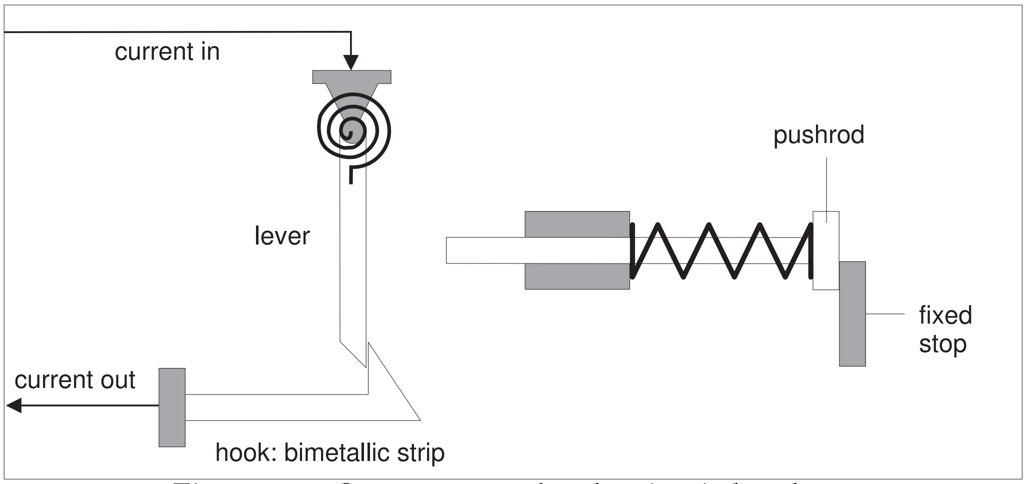


Figure 1-1: One structure for the circuit breaker.

are engaged and the springs that are relaxed.[5] The arcs are the external inputs that drive the device: Figure 1-3b describes how the circuit breaker should behave in the face of heating and cooling the hook and pressing the reset pushrod.

Figure 1-4 shows a portion of one of the BEP-models SketchIT derives from the sketch of the circuit breaker (Figure 1-2) and the desired behavior (Figure 1-3b). The top of the figure shows the parameters that define the sloped faces on the lever and the hook (faces f2 and f5 respectively). The bottom shows the constraints that ensure this pair of faces plays its role in achieving the desired overall behavior: moving the lever clockwise pushes the hook down until the lever moves past the point of the hook, whereupon the hook springs back to its rest position. As one example of a constraint, the last equation, `0 > R14/TAN(PSI17) + H2_12/SIN(PSI17)`, constrains the parameters of the faces so that the contact point on face f2 never moves tangent to face f5. This in turn ensures that when the two faces are engaged, clockwise rotation of the lever always increases the deflection of the hook.[6]

The constraints in a BEP-Model represent the regions in parameter space (i.e., the space spanned by the geometric parameters) that provide the desired behavior, and thus define an entire family of design solutions that the designer can explore. Because the BEP-Model's constraints ensure that all changes to the geometry still produce the desired behavior, the model makes it easy for the designer to explore a whole family of designs.

We have used DesignView to solve the constraints of BEP-Models and to draw

[5]Here we assume that the pairs of faces not listed at a node are by default disengaged, the springs not listed are by default not relaxed. See Chapter 5 for a more complete discussion of our behavior description language.

[6]If the contact point on face f2 did move tangent to face f5, moving the lever would not change the deflection of the hook.

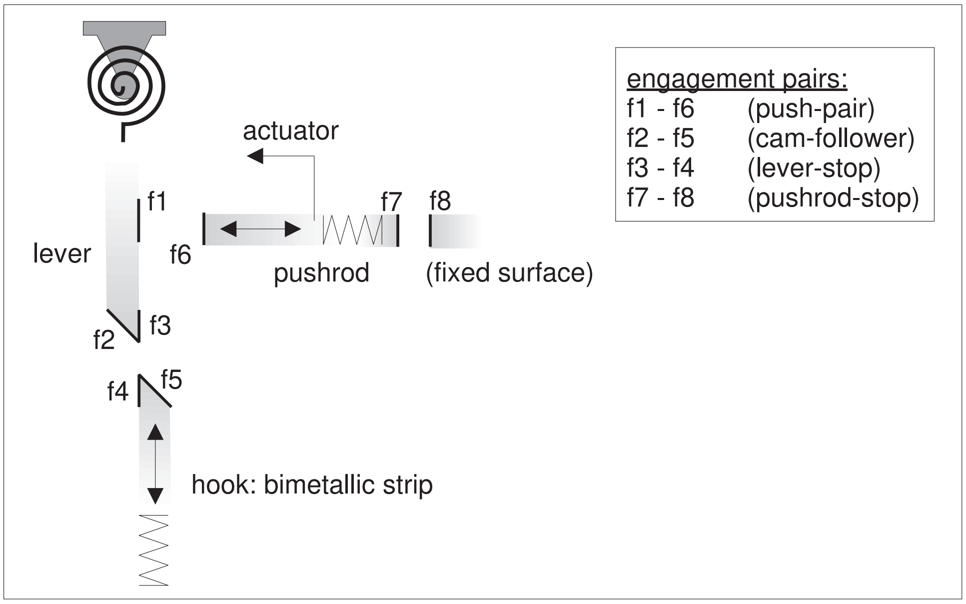


Figure 1-2: The sketch of the circuit breaker as actually input to SketchIT. Engagement faces are shown as bold line segments labeled f1 through f8. The actuator applied to the pushrod represents the reset motion imparted by the operator. The parenthesized names in the list of engagement pairs (e.g., push-pair) are for our convenience in exposition and are not used by the program.

the resulting geometry. DesignView enforces a BEP-Model's constraints while at the same time allowing the user to interactively change the parameter values, either by typing new values with the keyboard or by resizing the geometry with the mouse.

The particular parameter values shown in Figure 1-4 are a solution to the BEP-Model shown there. We used DesignView to compute these values. Figure 1-5 shows another solution we obtained with DesignView. Because these parameter values also satisfy the constraints of the BEP-Model, even this rather unusual geometry provides the desired behavior. As this example illustrates, the family of designs defined by a BEP-Model includes a wide range of design solutions, many of which would not be obtained with conventional approaches.

SketchIT also produces two kinds of design variants. It represents each individual variant with its own BEP-Model, and hence, each design variant is itself a family of designs. The program produces one kind of variant by changing rotating parts to translating ones, and vice versa. Figure 1-6 shows an example in which SketchIT has replaced the rotating lever in Figure 1-2 with a translating part. This particular design variant is appealing because of its simplicity.

SketchIT produces a second kind of variant by using different implementations for the pairs of engagement faces. Figure 1-7 shows an example: In the original implementation of the cam-follower engagement faces, the motion of face f2 is roughly

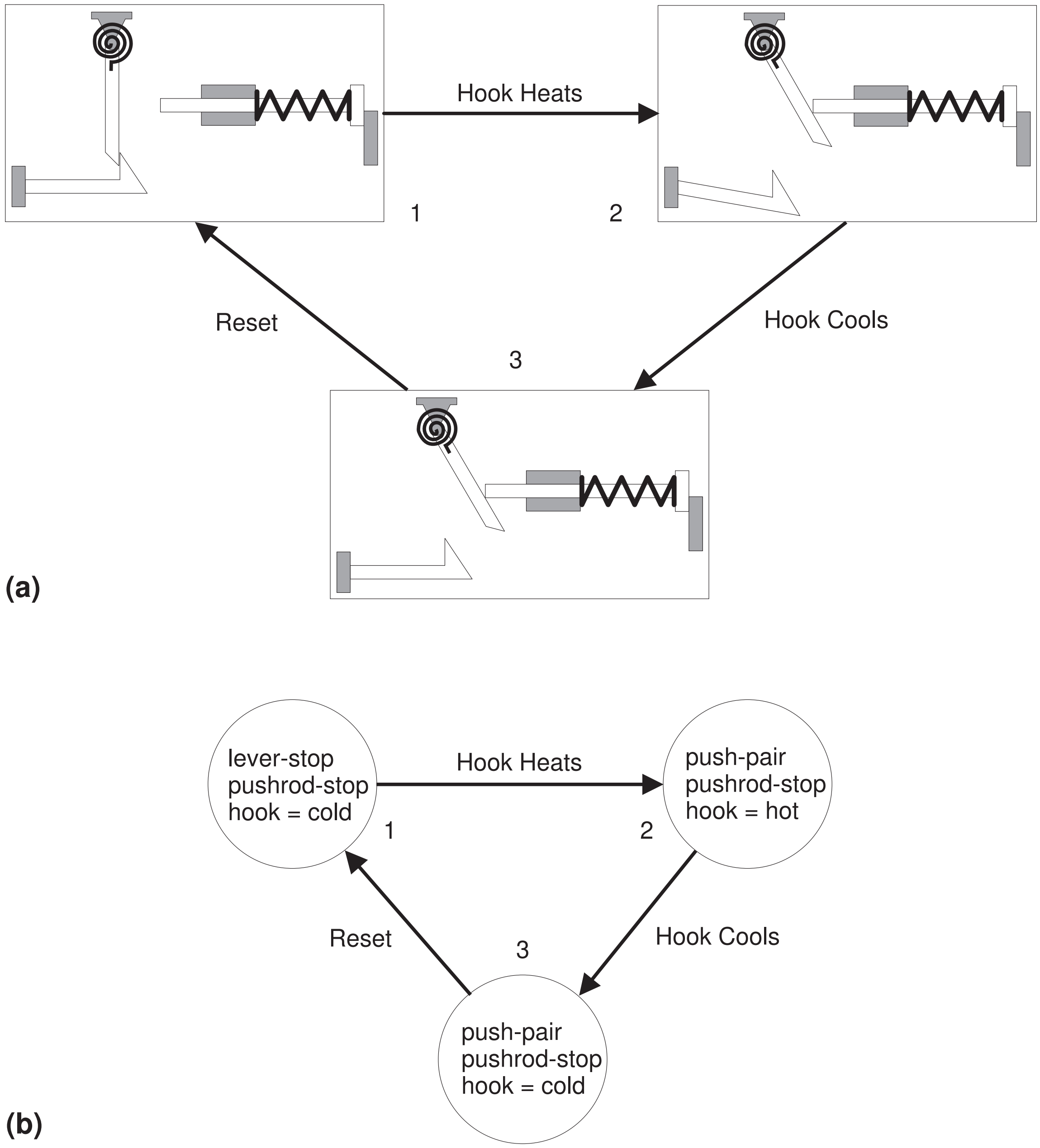


Figure 1-3: The desired behavior of the circuit breaker. (a) Physical interpretation. (b) State transition diagram. In each of the three states, the hook is either at its hot or cold neutral position.

perpendicular to the motion of face f5; in the new design of Figure 1-7 the motions are parallel. Hence, the original design implements the cam-follower interaction with a cam and centered follower while the new design uses a cam and offset follower. Conversely, the push-pair faces, originally implemented with a cam and offset follower, are implemented with a cam and centered follower in the new design. As the figure illustrates, these new implementations for the circuit breaker's engagement faces result in a rather unusual design.

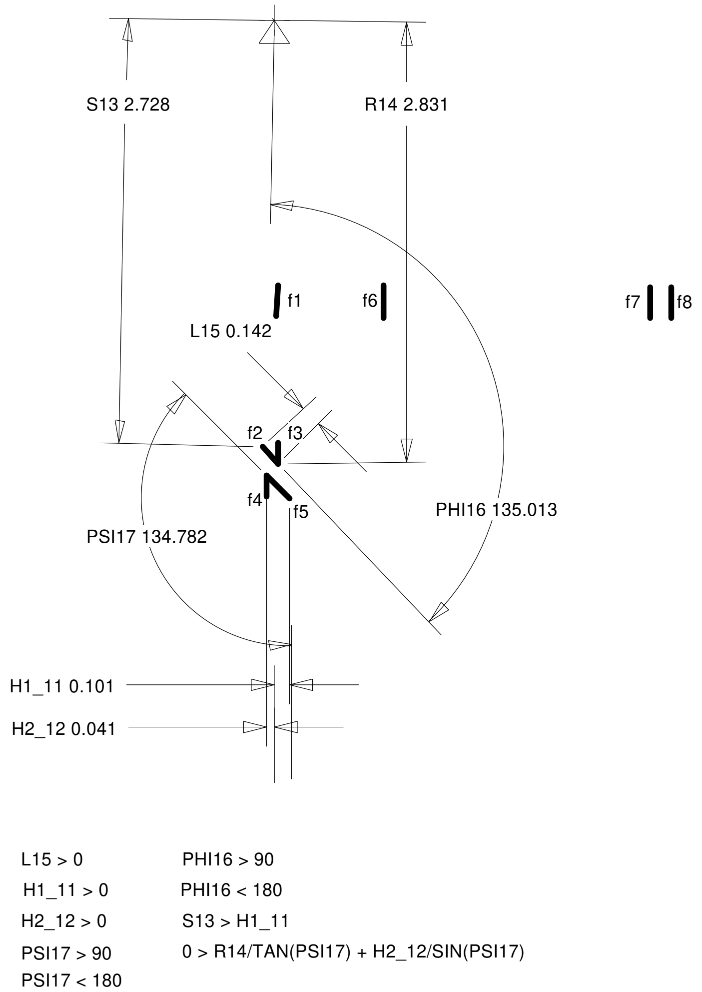


Figure 1-4: A BEP-Model for the circuit breaker. For clarity, only those parameters and constraints that refer to the cam-follower interaction (i.e., faces f2 and f5) are shown. The decimal number next to each parameter (e.g., the "2.831' next to "R14") is the current value of that parameter. This particular set of parameter values is a solution to the BEP-Model, and hence this device achieves the desired behavior.

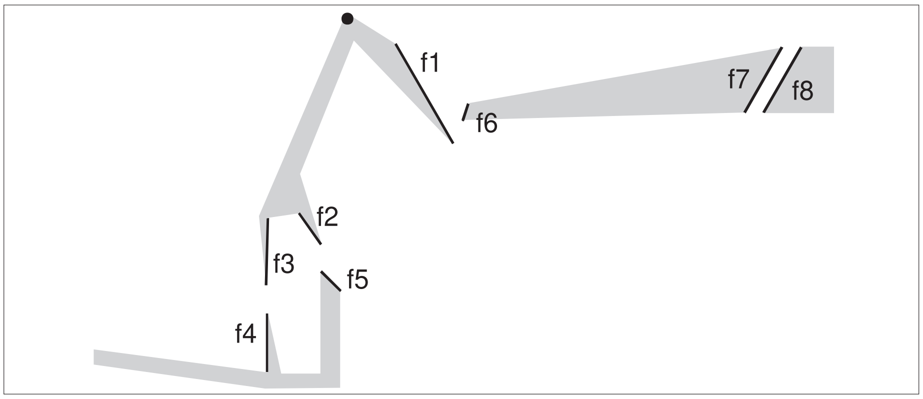


Figure 1-5: Another solution to the BEP-Model of Figure 1-4. Shading indicates how the faces might be connected to flesh out the components. This solution shows that the pair of faces at the end of the lever and the pair of faces at the end of the hook need not be contiguous.

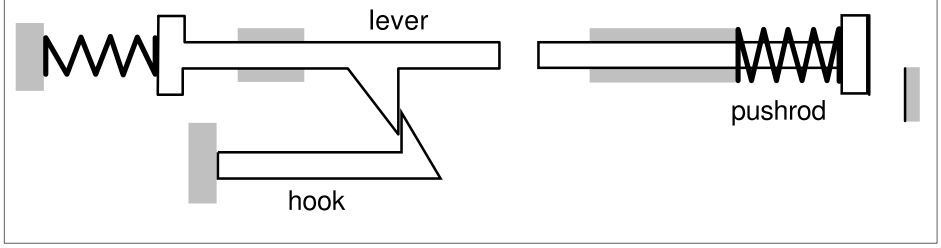


Figure 1-6: A design variant obtained by replacing the rotating lever with a translating part.

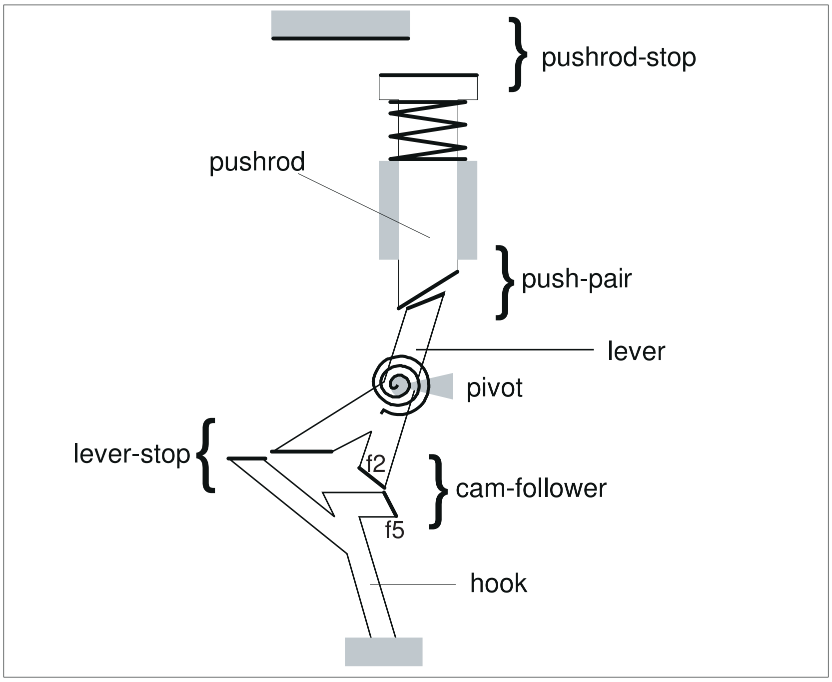


Figure 1-7: A design variant obtained by selecting different implementations for the cam-follower and push-pair engagement pairs. In the state shown, the pushrod is pressed so that the hook is just on the verge of latching the lever. When the pushrod is pressed slightly farther, the cam-follower faces will disengage. Then the hook will relax to the right, placing the dogleg on the hook under the dogleg on the lever, thereby engaging the lever-stop faces.

Firing Mechanism

Our second example concerns the design of a firing mechanism from a single action revolver shown in Figure 1-8. This example provides a second illustration of the stylized sketches and behavior descriptions that SketchIT takes as input.

We describe the device to SketchIT with the stylized sketch in Figure 1-9, created with the sketching tool described in Appendix B. As before, the sketch is constructed from line segments representing interacting faces and icons representing springs, actuators (external motion sources), pivots, and slider joints. These are the primitives from which all of our sketches are constructed. The figure also contains the list of faces that are intended to engage each other. The list of intended engagements is always required input.

We specify the desired behavior, as we always do, with a state transition diagram (Figure 1-10). The device starts with the hammer cocked (state 1); Pulling the trigger causes the hammer to fall (state 2); Pulling the hammer back re-cocks the device, returning it to state 1.

From the sketch and desired behavior, SketchIT computes multiple new designs, each of which is represented with a BEP-Model. These variants include both designs with new motion types and designs with new implementations for the engagement faces.

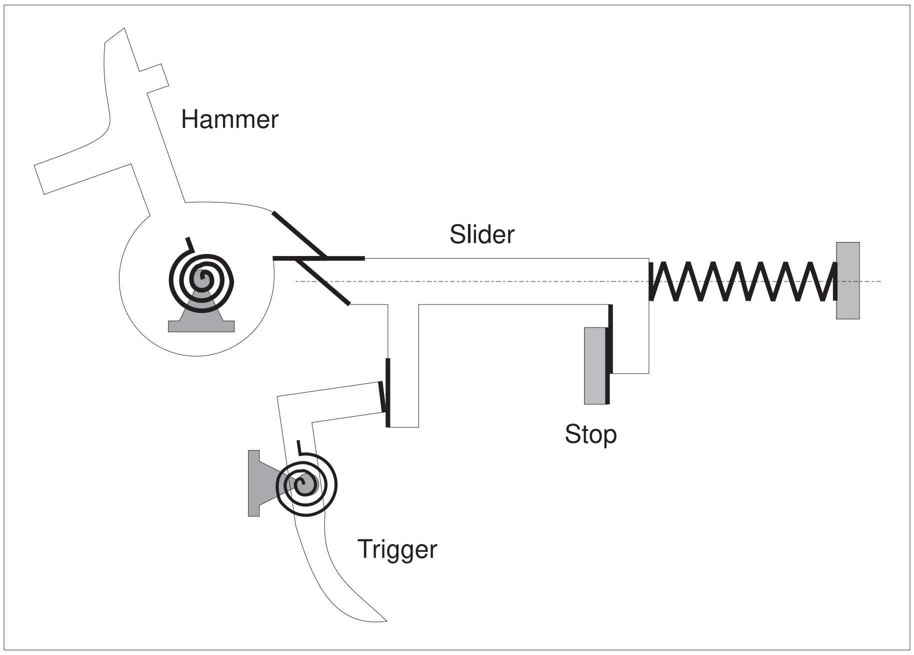


Figure 1-8: A firing mechanism from a single action revolver. Engagement faces are shown as bold line segments.

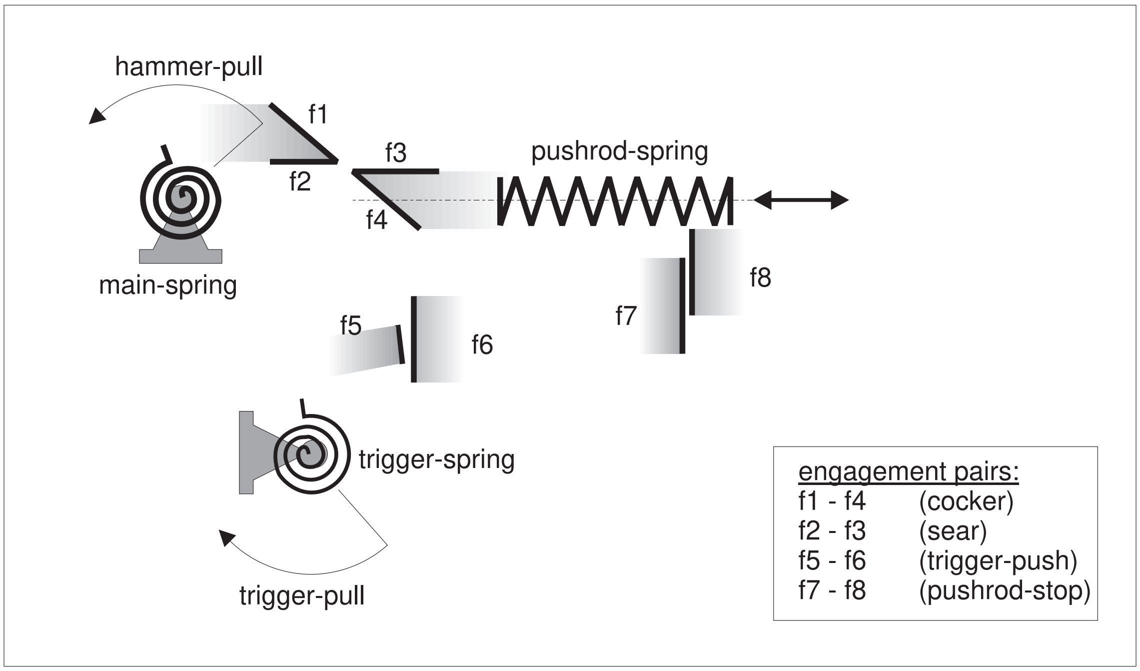


Figure 1-9: The sketch of the firing mechanism as actually input to SKETCHIT. Engagement faces are shown as bold line segments labeled f1 through f8. The actuator applied to the hammer (hammer-pull) represents pulling the hammer back. The actuator applied to the trigger (trigger-pull) represents pulling the trigger. The parenthesized names in the list of engagement pairs (e.g., sear) are for our convenience in exposition and are not used by the program. The curved arrow represents the motor turning the rotor. f3, f4, f6, and f8 are attached to the pushrod. f7 is fixed to ground. f1 and f2 are attached to the hammer.

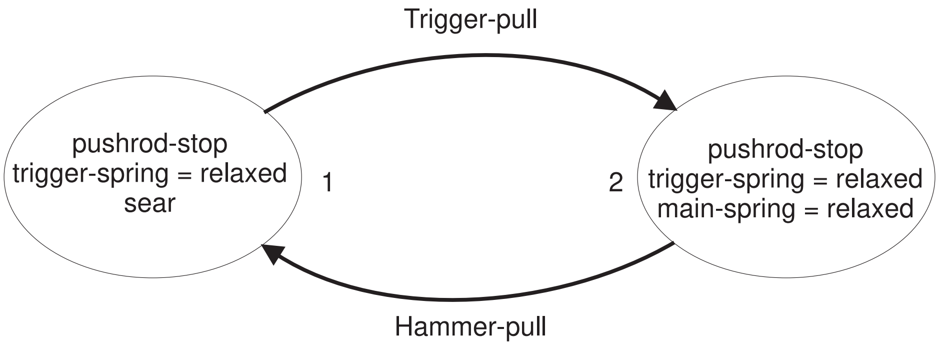


Figure 1-10: The desired behavior of the firing mechanism.

Yoke and Rotor

Our final example concerns the design of the yoke and rotor device shown in Figure 1-11. Continuous counter-clockwise rotation of the rotor causes the yoke to oscillate left and right with a brief dwell between each change in direction. We use this example to illustrate how SKETCHIT assists the designer in refining an initial concept to meet specific design requirements.

There are two important differences between this example and the previous two. First, in previous examples the rotating parts rotated less than a full revolution but here the rotor's motion is unbounded. Because the rotor can turn through full revolutions its motion is periodic: when the rotor turns far enough in a single direction, it will return to its initial position.[7] SKETCHIT employs qualitative reasoning techniques and periodic motion is often problematic for qualitative reasoners.[8] We, however, have developed special techniques to handle this kind of periodic motion (see Section 4.7). The second difference is that, while previously each face interacted with exactly one other face, here each face engages multiple faces. Consequently, in this example SKETCHIT must design geometry for each face such that the face correctly achieves multiple, distinct interactions.

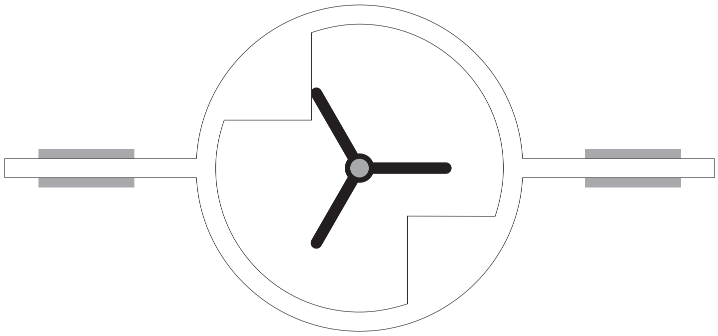


Figure 1-11: The yoke and rotor device.

We describe the yoke and rotor to SKETCHIT with the stylized sketch in Figure 1-12. The desired behavior is to have each of the rotor blades engage each of the yoke faces in turn as shown in Figure 1-13. From this input SKETCHIT generates the BEP-Model in Figure 1-14.

The designer can now use the family of designs specified by this BEP-model as a tool for refining the initial concept of Figure 1-12 to meet specific design requirements.

[7]One consequence of the rotor having periodic motion (i.e., turning through full revolutions) is that it cannot be replaced with a translating part; Translating parts cannot exhibit periodic motion. See Chapter 6.

[8]See for example [33].

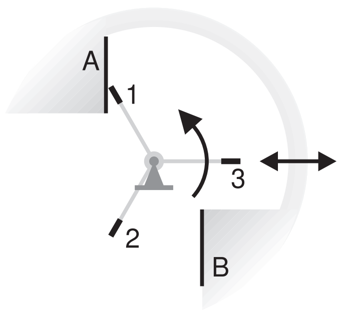


Figure 1-12: The stylized sketch of the yoke and rotor as actually input to SKETCHIT. The faces on the yoke are labeled A and B; those on the rotor are labeled 1–3. Each of the rotor faces is intended to engage each of the yoke faces.

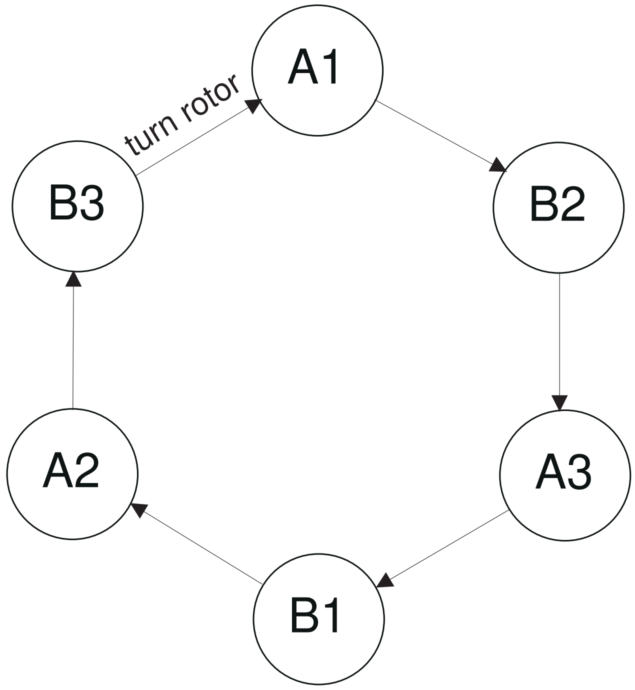


Figure 1-13: The desired behavior of the yoke and rotor. The letter and number in each node indicate which yoke face and which rotor face are engaged in that state. Turning the rotor is the external input that causes each of the transitions.

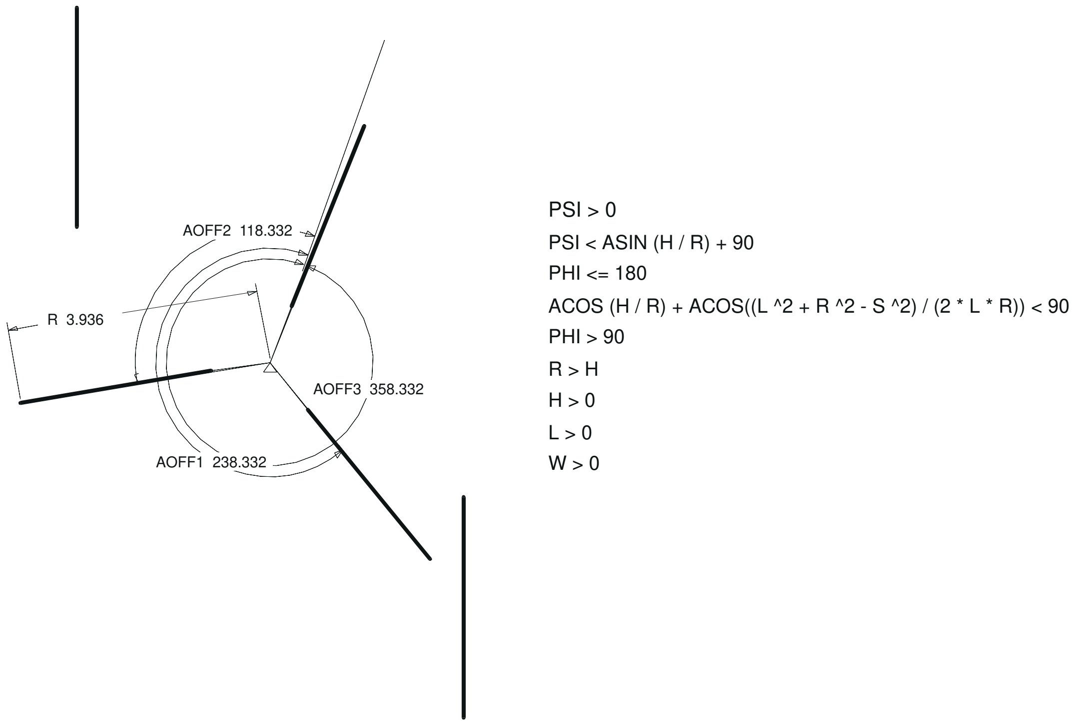


Figure 1-14: A BEP-Model for the yoke and rotor; a representative sample of the parameters and constraints are shown. For simplicity, new variable names have been substituted for sets of variables constrained to be equal. For example, because all three rotor blades are constrained to have equal length, R replaces $R1$, $R2$, and $R3$.

Imagine that one requirement is that all strokes have the same length. A simple way to achieve this is to constrain the yoke and rotor to be symmetric. We do this by adding additional constraints to the BEP-Model, such as the following which constrain the rotor blades to be of equal length and have equal spacing:

$$R1 = R2 = R3$$

$$AOFF1 - AOFF2 = 120°$$
$$AOFF3 - AOFF1 = 120°$$

Imagine further that all strokes are required to be 1.0cm long. We achieve this by adding the additional constraint:

$$LM29 - LM27 = 1.0$$

$LM29$ and $LM27$ are variables that SKETCHIT assigns to the extreme positions of the yoke. We obtain the names of these variables by using a graphical browser to inspect SKETCHIT's simulation of the device. (Because we have constrained the yoke and the rotor to be symmetric, all strokes have the same length.)

Finally, imagine that the dwell is required to be 40°, i.e., between each stroke, the rotor turns 40° while the yoke remains stationary. We can achieve this by adding one additional constraint:[9]

$$LMG - LM8 = 40°$$

We can now invoke DesignView to find a solution to this augmented set of constraints; the solution will be guaranteed to produce both the desired behavior and the desired performance characteristics (stroke and dwell). The interesting point is that we were able to refine the design simply by adding additional constraints to the BEP-Model.

[9] LMG and $LM8$ are the variables that SKETCHIT assigns to the position of the rotor at the beginning and ending of one the dwells. Because we have constrained the yoke and the rotor to be symmetric, all six dwells have the same duration.

1.1.4 Motivation

Designers frequently use sketches because they are a convenient and efficient way to capture and communicate design information. Traditional design tools, on the other hand, do not make use of the designer's "natural language"—sketches; in fact, these tools often require the designer to describe the design using complex and time consuming representations. For example, Finite Element Analysis (FEA) tools require that the geometry of the device be "meshed" with large numbers of simple shapes like square plates and thin rods. By developing methods that exploit the designer's natural language, this project has the potential to fundamentally change the way computation is used in design.[10]

This project concerns the development of a computational tool that can transform a sketch into working geometry, an essential step on the path from conceptual design to detailed design. This brings new computational power to the early phases of design, which traditionally have had very little computational support. Conventional computational tools are of little use in the early design phases because they require detailed design information which is simply not known until later in the design process. FEA tools, for example, require a complete description of the geometry, as well as the material properties and external loading, before they can compute the stresses on the mechanical components in the design. By introducing new computational power into the early phases of design, this project has the potential to greatly accelerate the design process.

During conceptual design, a high premium is put on examining a large number of designs in the hope of finding at least one good solution. The more designs that are explored, the better are the odds of discovering a good solution. By developing methods for automatically generating design variants, this project may lead to a more successful conceptual design process, perhaps producing solutions that would not be obtained with traditional design methods.

1.2 The Approach

1.2.1 Introduction

Figure 1-15 illustrates the basic paradigm that SKETCHIT uses to transform a sketch into working geometry and to generate alternative designs. There are two processes. In the first, SKETCHIT reverse engineers and generalizes the original design. The

[10] An historical note: we originally began by examining the spoken language that designers use to describe mechanical artifacts [47]. We quickly discovered that it is very difficult to describe a mechanical device to another person with spoken language alone. This task becomes much easier if sketches are used. Spoken language is still useful: It augments the sketch by providing additional information about the intended behavior of the device. In much the same spirit, SKETCHIT uses a state transition diagram to annotate the sketch with the intended behavior.

result of this process is what we call a "qualitative configuration space" (qc-space) representation of the design. In the second process, SKETCHIT uses this qualitative representation as a design specification, from which it generates multiple implementations for the design.

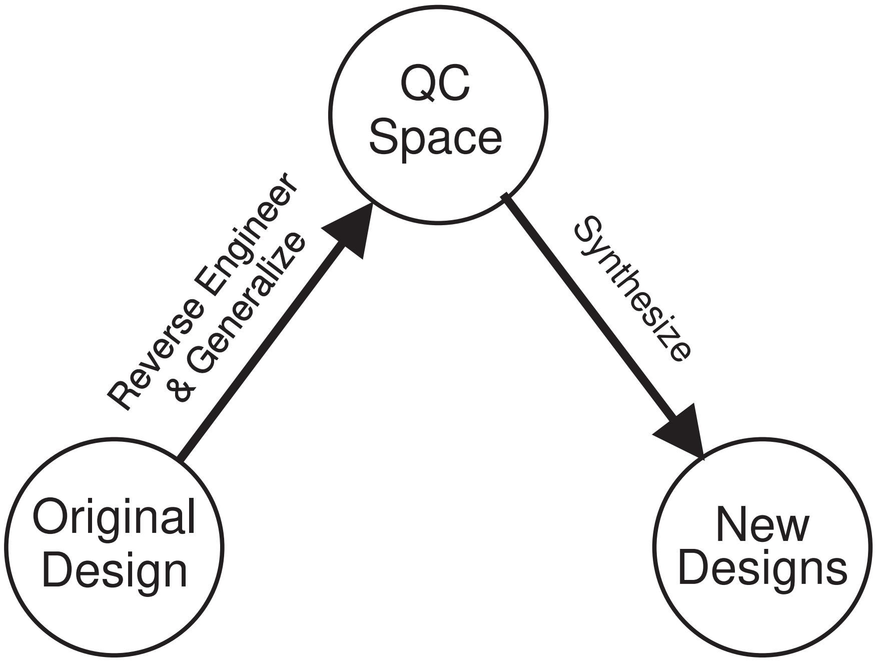


Figure 1-15: The problem solving paradigm.

SKETCHIT's problem solving paradigm is fundamentally one of abstraction followed by re-synthesis. SKETCHIT begins with abstraction because an abstract representation simplifies the reasoning process: A sketch contains an enormous amount of geometric detail, much of which is irrelevant to the behavior. The abstraction process strips away the irrelevant detail to expose what is essential.

SKETCHIT's qualitative configurations space representation is the key technology allowing the program to perform its tasks. In the next section we describe this representation in more detail; then we briefly revisit our problem solving paradigm and finally we describe our implementation.

1.2.2 The Representation

SKETCHIT's task is to transform a given sketch into families of working geometries. To do this the program must first determine what role each part of the device plays in achieving the desired overall behavior. This task is made difficult by the fact that sketches of conceptual designs are often incomplete and incorrect.

Sketches are incomplete in the sense that details are missing: a sketch may depict something about the kind of implementation the designer wants to use, rather than a specific, working implementation (e.g., the designer may want a cam and follower, but not necessarily with the exact dimensions and shapes shown in the sketch). A sketch can be incorrect in the sense that it contains a collection of fragments that implement pieces of the design, but those individual pieces are not integrated to produce the desired overall behavior.

As SKETCHIT overcomes the flaws in the original sketch and identifies the role of each part of the device, it is reverse engineering the design. The usual reverse engineering problem is to identify what role each part of the device plays in achieving the *actual* overall behavior. Here, however, the problem is to determine what role each part *should* play in order to achieve the *desired* overall behavior.

In the domain of sketches of the sort in Figure 1-2, the overall behavior is achieved through a sequence of interactions between pairs of engaging faces.[11] We call a pair of engaging faces an "engagement pair". (All of the engaging faces in Figure 1-2 are shown as bold lines.) If each engagement pair provides the correct behavior, the overall device will behave as desired. Hence, the goal of reverse engineering is to determine what behavior each engagement pair should provide.

After reverse engineering the sketch, SKETCHIT generates new implementations for the device by generating new implementations for its engagement pairs. For example, Figures 1-4 and 1-7 are two alternative implementations SKETCHIT produced for the circuit breaker by selecting different implementations for the engagement pairs. In both of these designs, each individual engagement pair provides the required behavior identified by reverse engineering, hence each design as a whole provides the desired overall behavior.

To generate these kinds of design alternatives, SKETCHIT must be able to represent the behavior of an engagement pair in a way that abstracts away the particular implementation, thereby affording SKETCHIT the opportunity to select new implementations. For this purpose SKETCHIT uses a novel behavior representation we call qualitative configuration space. Qualitative configuration space (qc-space) is an abstraction of the conventional (numerical) configuration space (c-space) representation commonly used in robotics (e.g., [34]). Because it abstracts away the implementation used in the sketch, qc-space generalizes the design.

Because SKETCHIT represents the behavior of engagement pairs with qc-space, the goal of reverse engineering is to find a qc-space representation of the design that produces the desired overall behavior. Qc-space, explicitly represents the behavior of individual engagement pairs, rather than the behavior of the overall device. SKETCHIT determines the overall behavior by qualitatively simulating the device's

[11] In general, the behavior of a mechanical device is determined by the interactions between the shapes of the parts. There are well known design techniques for devices with constant interactions, such as the gear train and linkage design techniques in Erdman and Sandor [10]. We focus on devices with time varying interactions because this class of devices is not as well understood.

qc-space, using a technique we describe below.

1.2.3 Problem Solving Paradigm

As the previous section described, the end result of the "reverse engineer and generalize" process in Figure 1-15 is a working qc-space—a qc-space that produces the desired overall behavior. This section briefly describes how SKETCHIT computes a working qc-space from the sketch, then how it uses this to generate new designs.

SKETCHIT begins by abstracting the numerical c-space of the sketch to produce a qc-space; this initial qc-space represents the actual behavior of the engagement pairs in the sketch. SKETCHIT uses qualitative simulation to determine if this initial qc-space provides the desired behavior (i.e, if the device as initially drawn would work). If so, SKETCHIT is done with this step. If not, the program modifies this initial qc-space until it does provide the desired behavior.

Once SKETCHIT finds a working qc-space for the sketch, the synthesis process begins. The working qc-space specifies what behavior each engagement pair should provide. The synthesis process selects for each engagement pair an implementation that achieves the behavior specified by the working qc-space.

As we mentioned before, an important property of qc-space is that it abstracts away implementation: it abstracts away both the motion type (rotation or translation) of the components and the geometry of the engagement faces. Hence, during the synthesis process, SKETCHIT is free to select a motion type for each component and geometry for each engagement face.[12] For example, SKETCHIT produces the circuit breaker design in Figure 1-6 by selecting rotation for the motion type of the lever, rather than the translation used in the original sketch (Figure 1-2). Similarly, SKETCHIT produces the circuit breaker design in Figure 1-7 by selecting different geometries for the cam-follower and push-pair faces.

1.2.4 Implementation: System Overview

Figure 1-16 illustrates the way that SKETCHIT implements the problem solving paradigm of Figure 1-15. The reverse engineer and generalize process is a form of generate and test. The qc-space generator produces candidate qc-space representations of the sketch and passes them to the simulator. Each candidate qc-space is a guess at what behavior each engagement pair should provide. The first candidate is an abstraction of the numerical c-space of the sketch; the rest are modifications of the first. The simulator computes the overall behavior of each candidate, which the tester then compares to the desired behavior described in the state transition diagram. This process continues until the tester finds a qc-space that behaves as desired.

The synthesis process is implemented with two modules. The first module selects

[12]See Chapter 6 for the conditions under which changing motion type is possible.

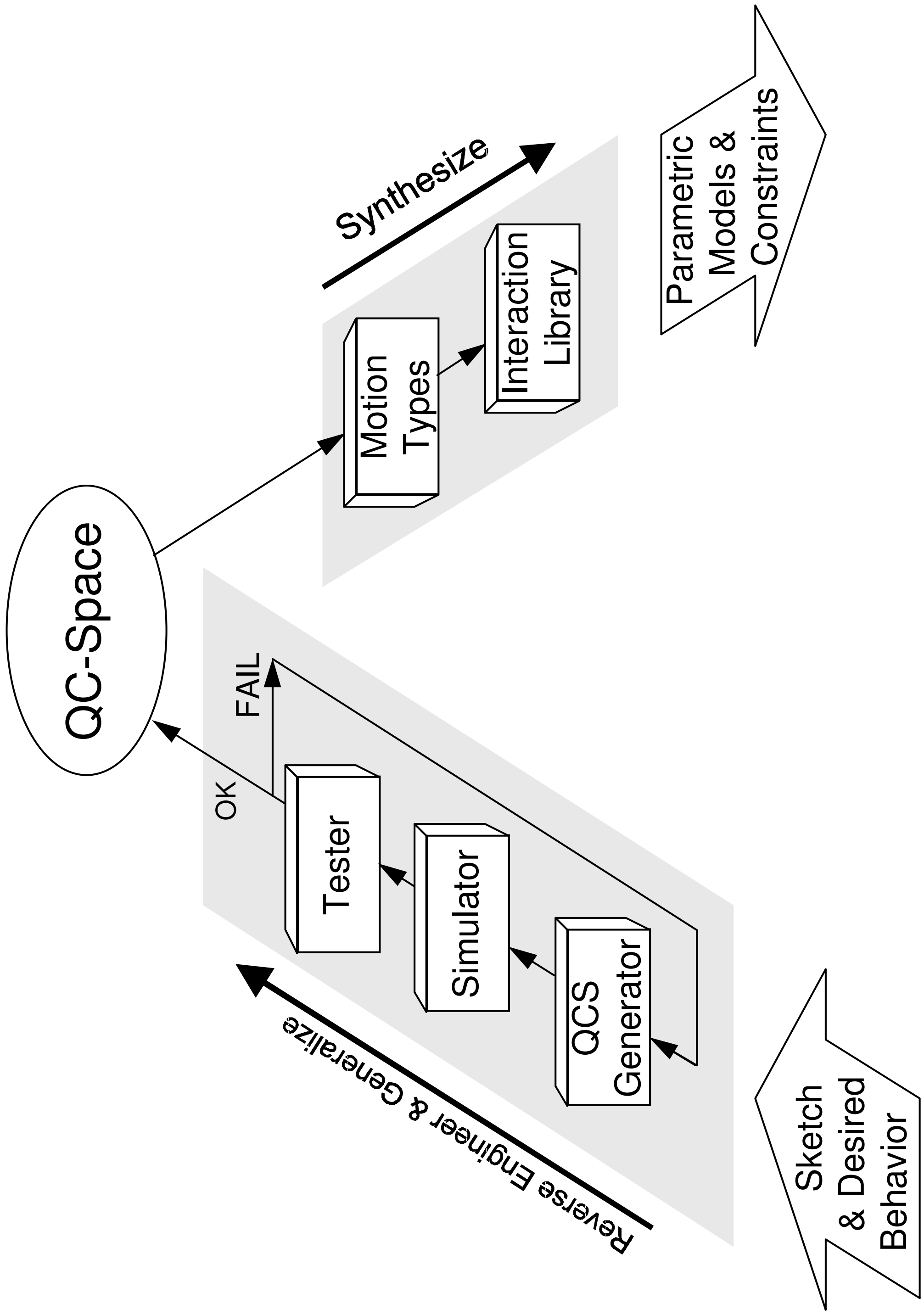


Figure 1-16: Overview of the SKETCHIT system.

a motion type for each component. The second module selects a geometric implementation for each engagement pair from a library of interactions. Each library entry contains a pair of parameterized faces and constraints that ensure that the faces implement a specific kind of behavior. SketchIT assembles the parametric geometry and constraints from the library selections into a BEP-Model.

The bulk of SketchIT's effort is spent reverse engineering and generalizing the design. Because SketchIT synthesizes new designs by using a library of interactions, the synthesis process is straightforward and computationally inexpensive.

1.2.5 Assumptions

SketchIT's domain is the class of mechanical devices composed of rigid bodies, springs, pivots, slider joints, and actuators,[13] subject to the assumptions below, used (primarily) to make simulation of the candidate qc-spaces tractable.

- Fixed-axis devices: Each component in the device has a single degree of freedom, the device as a whole may have multiple degrees of freedom.
- Springs have one end fixed to ground.
- Inertia-free motion: The inertial terms in the equations of motion are negligible compared to the other terms (e.g., due to large applied forces or high damping).
- Inelastic collisions: When two objects collide, they remain in contact rather than bouncing apart.
- Frictionless contacts. When two parts touch, there is no friction to resist one part sliding against the other.
- Monotonic engagements: When one component (the driving component) pushes another component (the driven component), the driven component's motion will not change direction as long as the driving component's motion does not.
- Engagement faces are flat.

Assuming that devices are fixed-axis precludes linkages. This is necessary because SketchIT uses qualitative simulation techniques and it is generally accepted that linkages cannot be effectively analyzed using qualitative techniques.[14]

If both ends of a spring can move, the spring's deflection is the difference between the motion of its two ends. With a qualitative representation, differences in motion are subject to ambiguity. If, on the other hand, one end of a spring is fixed, its

[13] In SketchIT's world an actuator is a motion source, i.e, an actuator applies velocity to a body.
[14] See for example [46].

deflection is determined by the absolute motion of its other end. Hence, by requiring springs to have one end fixed, we reduce ambiguity in the simulation.

The inertia-free assumption is equivalent to computing motion using quasi-statics rather than dynamics. Because our equations of motion are assumed to have no inertia terms, they are first order differential equations. Hence, this assumption is often referred to as "first order dynamics." Although assuming negligible inertia greatly simplifies the qualitative simulation process, it does not pose much of a restriction: Sacks and Joskowicz [43] examined 2500 mechanisms in a catalog of mechanisms and found that 80% could be modeled accurately with first order dynamics.

In its current implementation, the program can handle only flat faces. Extending the program to handle other kinds of faces (e.g., circular faces) requires a more complete program module for computing the numerical c-space of the original sketch. (SketchIT uses the numerical c-space only to obtain an initial qc-space.) However, computing numerical c-space is a relatively well understood problem (see for example, [34], [1], [26], and [2]). All other modules of the program, including those that reason about behavior, are completely adequate to handle devices with non-flat faces.[15]

1.3 Guide to This Document

This document is organized around the system flow chart shown in Figure 1-16. There is one chapter for each of the modules in the program, as well as a chapter describing the behavior representation (qc-space).

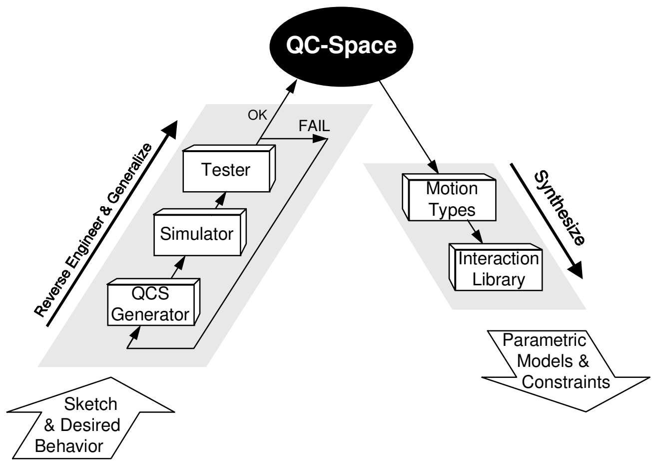
- Chapter 1 introduces the project, describing the motivation and problem solving paradigm.
- Chapter 2 describes qc-space.
- Chapter 3 begins the discussion of the modules in the SketchIT system by describing the qc-space generator.
- Chapter 4 describes the qualitative simulator module.
- Chapter 5 describes the tester module.
- Chapter 6 describes the motion type selector module.
- Chapter 7 describes the interaction library module.

[15]Because (almost) all of the entries in the current interaction library use flat faces, the current program *produces* only designs that use flat faces, regardless of the kinds of faces used in the original sketch. We have, however, begun developing library entries that use circular faces.

- Chapter 8 reviews related work.
- Chapter 9 discusses the contributions of this project and discusses future work.
- Appendix A: Provides a complete listing of the interaction library.
- Appendix B: Describes the sketching tool.
- Appendix C: Describes our techniques for computing numerical c-space.

Chapter 2

The Representation: QC-Space



2.1 Introduction

SKETCHIT's task is to transform a sketch into a set of working geometries, including alternative implementations for the design. For the class of devices that SKETCHIT is concerned with, the overall behavior is achieved through a sequence of interactions between pairs of engagement faces. Hence, to transform a sketch into even a single working geometry, SKETCHIT must be able to represent and reason about the behavior of interacting faces. So that SKETCHIT can generate alternative designs, SKETCHIT's representation for behavior must abstract away any particular implementation, allowing SKETCHIT the opportunity to select new implementations.

Because configuration space (c-space) is commonly used to represent the behavior of interacting faces, our search for a behavioral representation began there. Although c-space is capable of representing the behaviors we are interested in, it does not adequately abstract away the implementation of a behavior. Thus, our next step was to examine ways to extend c-space to obtain a new representation that does adequately abstract away the implementation. We discovered that abstracting c-space into a qualitative form produces the desired effect; hence we call SKETCHIT's behavioral representation "qualitative configuration space" (qc-space).

Because qc-space is an abstraction of c-space, this chapter begins with a brief description of c-space, then describes how to abstract c-space to produce qc-space.

2.2 Configuration Space

2.2.1 Representing Engagement Faces

Configuration space represents the way mechanical components can interact. Consider the rotor and slider in Figure 2-1. If the angle of the rotor, U_R, and the position of the slider, U_S, are as shown, the faces on the two bodies will touch. Hence, these particular values of U_R and U_S define a *configuration* of the bodies in which the faces touch. We can represent this pair of values as a point in the plane shown in Figure 2-2. This plane is called a configuration space plane (cs-plane).

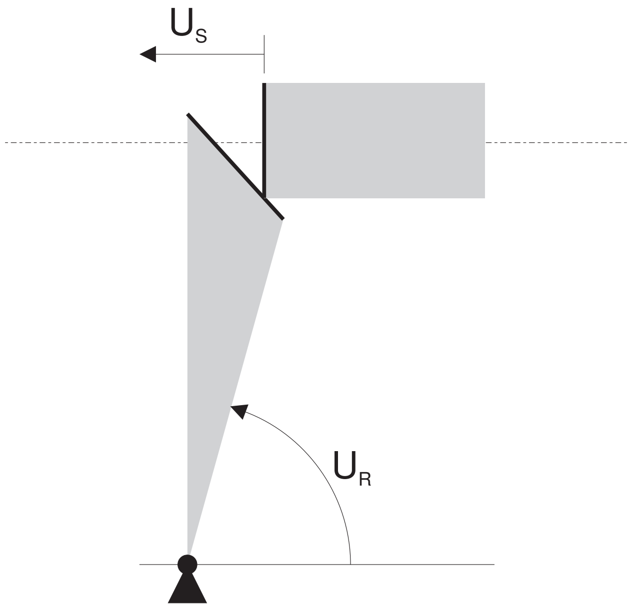


Figure 2-1: A rotor and slider. The slider translates horizontally. The interacting faces are shown with bold lines.

If we determine all of the configurations of the bodies in which the faces touch, and plot the corresponding points in the cs-plane, we get a curve. This curve, called a configuration space curve (cs-curve), is shown in Figure 2-2. The shaded region "behind" the curve indicates blocked space, configurations in which one body would

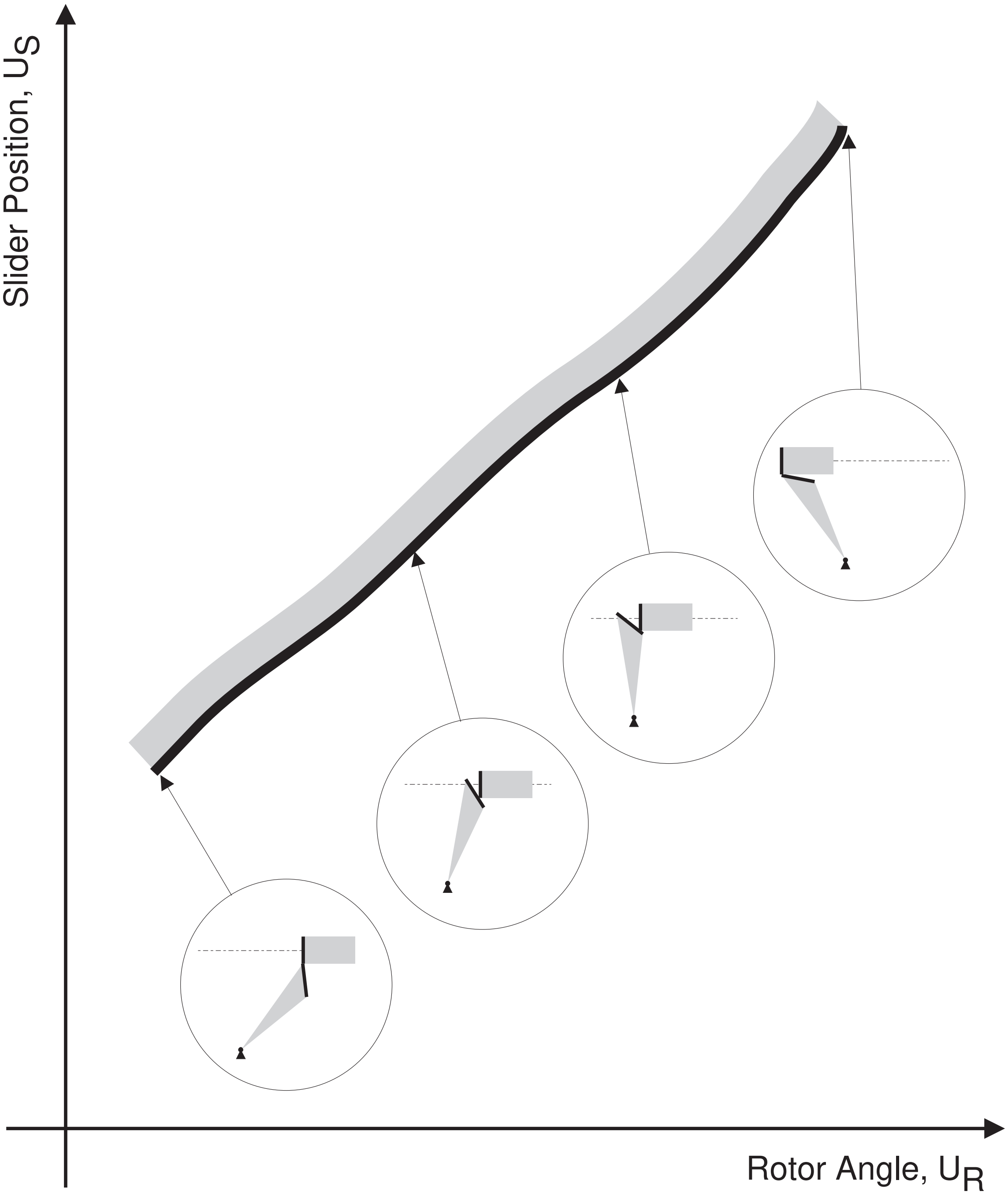


Figure 2-2: The c-space of the rotor and slider. The inset figures show the configuration of the rotor and slider corresponding to selected points on the cs-curve.

penetrate the other. The unshaded region "in front" of the curve represents free space, configurations in which the faces do not touch.

To be more precise, the shading indicates which configurations in the neighborhood of the curve are blocked space. The complete blocked space is obtained by computing the cs-curves for all pairs of faces in which one face is taken from each body. Here, however, we are concerned with the blocked space of only those pairs of faces that are intended to interact. Our task is to design geometry for these faces such that they provide the desired interaction. In a later stage of design, other surfaces will be added to connect these faces to produce complete solids. At that point, we must compute the complete blocked space to ensure that the blocked space contributed by the connective surfaces does not preclude any of the desired interactions.

In general, c-space represents the configurations of a device in which the bodies do not touch (free space), the configurations in which the bodies penetrate (blocked space), and the configurations in which the bodies just touch (the boundary between free and blocked space). The dimension of the c-space for a set of bodies is equal to the number of degrees of freedom of the set. The axes of the c-space are the position parameters (generalized coordinates) of the bodies.

To simplify geometric reasoning, we assume that devices are fixed-axis. That is, we assume that each body either translates along a fixed axis or rotates about a fixed axis. Hence, in our world, the c-space for a pair of bodies will always be a plane (a cs-plane) and the boundary between blocked and free space will always be a curve (a cs-curve). However, even in our world, a device may be composed of many fixed-axis bodies. Thus, the c-space for the device as a whole can be of dimension greater than two (the dimension is equal to the total number of degrees of freedom of all the bodies). The individual cs-planes are orthogonal projections of the multi-dimensional c-space of the overall device.

2.2.2 Periodic Boundary Conditions

So far we have assumed that the c-space for a pair of fixed-axis bodies is always a plane (a cs-plane), but now we will be more precise. The c-space for a pair of fixed-axis bodies will always be two-dimensional, but the topology will be planar only if the bodies rotate less than a full revolution. If a body does rotates more than a full revolution, its cs-plane must have periodic boundary conditions. For example, if the rotor and slider in Figure 2-1 start from some initial configuration, and the rotor angle increases through a full revolution, the device will be back in its initial configuration. For this to happen, the right edge of the cs-plane must wrap around and connect to the left edge to form a cylinder, as shown in Figure 2-3. (Similarly, the c-space for a pair of bodies that both rotate through full revolutions would be a torus.)

For ease of presentation, in the remainder of this chapter as well as most of the chapters that follow, we will restrict our attention to pairwise configuration spaces that are planar.[1] Then in Sections 4.7 and 7.7 we will complete our discussion by providing the details specific to non-planar pairwise c-spaces.

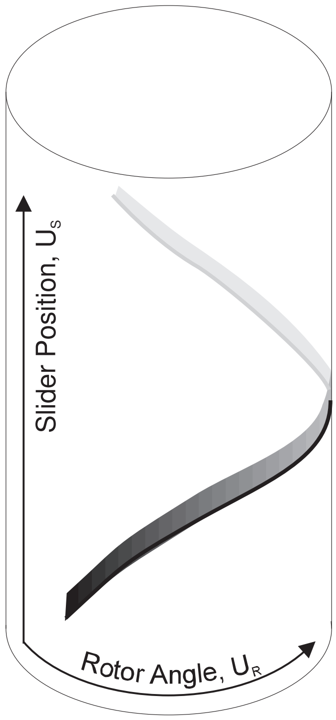


Figure 2-3: If the rotor turns through a full revolution, the c-space must have periodic boundary conditions. Thus, the c-space becomes a cylinder.

[1] By pairwise c-space we mean the c-space of a pair of fixed-axis bodies.

2.2.3 Representing Springs, Actuators, and Fixed Surfaces

In the previous sections we described how c-space represents the interaction between a pair of (moving) engagement faces. The devices that SKETCHIT is intended to handle may also contain springs, actuators, and interactions with fixed surfaces. In this section we describe how c-space represents the behavior of these other components.

We use as our example the device in Figure 2-4. The device consists of two blocks, A and B, which interact through a pair of sloped engagement faces. A spring is attached to block A, an actuator is attached to block B, and there is a fixed surface with which B may collide.

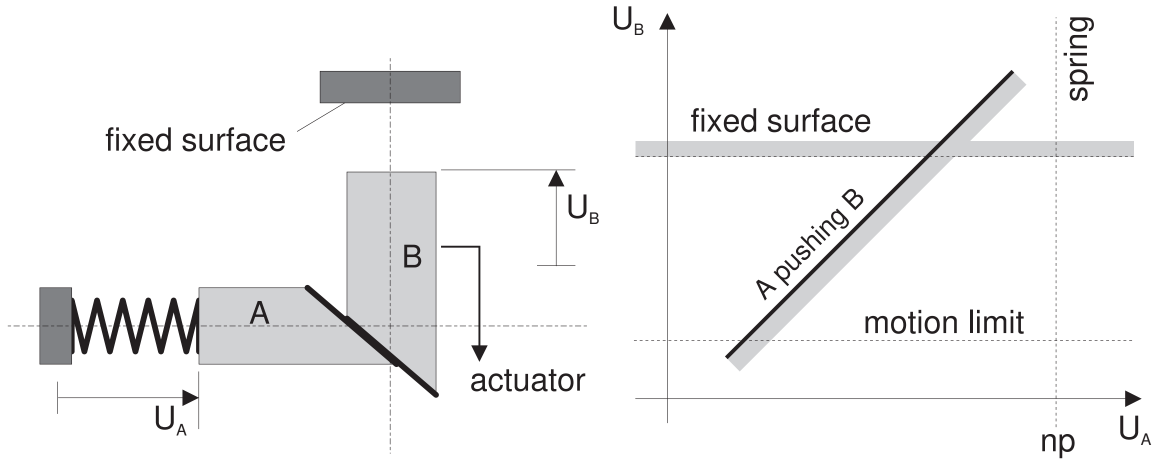


Figure 2-4: A simple mechanical system and its configuration space. Block A translates horizontally, block B translates vertically. A spring pushes A to the right, an actuator pushes B downward, B can bump into a fixed surface (if the actuator is turned off).

We begin with what we already know, the c-space representation for the interaction between the faces of A and B (computed using the techniques described in Section 2.2.1). In the configuration shown, the faces are touching, and thus this configuration is a point on the cs-curve for the engagement pair. If A moves a small distance in the positive direction, it will push B a small distance in the positive direction (assume the actuator is turned off). The faces will still be engaged in this new configuration, and hence we have another point on the cs-curve. Continuing in this fashion we obtain a diagonal cs-curve with positive slope, as the cs-plane in Figure 2-4 shows. If the faces are touching and A moves in the positive direction while B is held fixed, the faces will penetrate each other. Hence, the space to the right of the cs-curve is blocked space. (Alternatively, if B moves in the negative direction while A is held fixed, the faces will penetrate. Hence, the space below the cs-curve is blocked space. Either way, we compute the same blocked space.)

Next we consider the spring attached to block A. The neutral position of this spring corresponds to a particular position of A we call np. In the cs-plane, the locus of configurations in which A's position is np is an infinite vertical line. Hence the neutral position of the spring appears as a infinite vertical line in the cs-plane.

The neutral position of a spring always appears as a vertical or horizontal line in the cs-plane, because in SKETCHIT's world the relaxed state of a spring depends on the position of only one body (recall that a spring always has one end fixed).

In SKETCHIT's world, actuators apply a motion to a body until the body reaches a particular position called the motion limit of the actuator. Once the body reaches this position the actuator turns off (i.e., the actuator no longer applies any force or motion to the body). Because the motion limit of an actuator depends on the position of only one body (just as the neutral position of a spring does), a motion limit always appears as a vertical or horizontal line in the cs-plane. Hence, the motion limit of the actuator attached to block B is a horizontal line in the cs-plane of Figure 2-4.

Finally, we consider the interaction between the fixed surface and block B. (Because the actuator pushes B away from the fixed surface, this interaction will not happen until after the actuator turns off.) Because this interaction is independent of A's position, the corresponding cs-curve is an infinite horizontal line in the cs-plane. Because the fixed surface limits B's motion in the positive direction, there is blocked space above the cs-curve.

As Figure 2-4 illustrates, a cs-plane may contain both finite cs-curves representing the interaction between a pair of faces (the diagonal cs-curve), and infinite boundaries representing spring neutral positions, motion limits of actuators, and interactions with fixed surfaces. As a means of differentiation, we use solid lines for finite cs-curves and dashed lines for infinite boundaries. For shorthand convenience we often refer to both finite cs-curves and infinite boundaries as cs-curves.

2.2.4 Computing Motion Using C-Space

The previous sections described how c-space represents the behavior of all the different kinds of parts that compose the devices in our world. Our eventual goal is to determine what behavior for each part is sufficient for the device as a whole to behave as desired. Hence, at some point we will need to compute the overall behavior of a device from a description of the behavior of each of its parts. For mechanical devices composed of rigid bodies and springs, the overall behavior is characterized by the time history of the motion of each rigid body in the device.

Physics has already given us a well known set of equations for computing the time history of the motion of each body in a device, namely Newton's laws. Thus, in this section we describe how we use Newton's laws to compute motion. We use as our example the device consisting of a rotor and slider in Figure 2-5. To start with, we compute the motion of the rotor and slider by reasoning directly from their structure. Then, we repeat the process, this time computing the motion directly from the c-space description of the device.

Imagine that the rotor and slider are in the initial positions shown in Figure 2-5 and that a force (F) pushes the slider to the left. As it moves to the left, the slider will strike the rotor and begin to push it out of the way. Assuming the collision is inelastic, the rotor and slider will remain in contact, and the slider will continue to push the rotor out of the way. Eventually the slider will push the rotor far enough that the two parts disengage. Assuming the motion of the rotor is inertia-free, the rotor will then stop; the slider will continue to move as long as the force is applied.

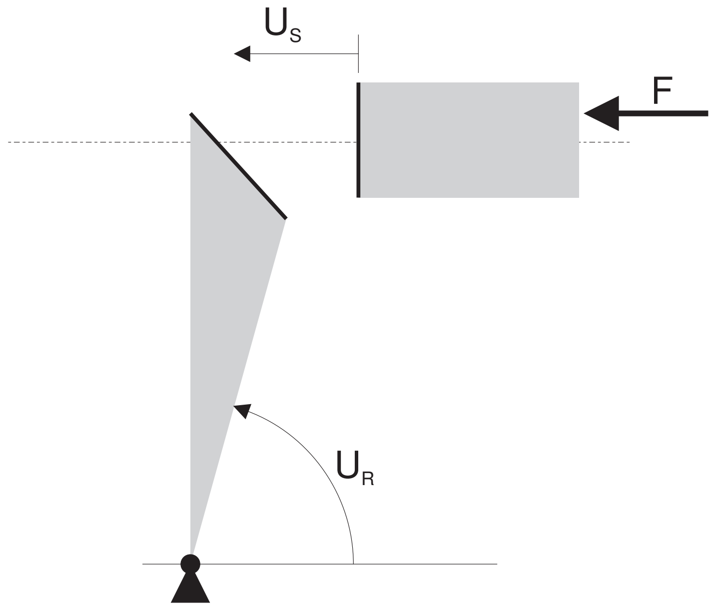


Figure 2-5: Force F pushes the slider to the left.

We can describe the motion of the rotor and slider as a sequence of configurations in c-space. We call this sequence of configurations the *trajectory* through c-space. Figure 2-6 shows the trajectory corresponding to the motion described above. In the initial configuration the rotor and slider are not touching, hence the initial configuration is in free space. When the motion begins, only the slider moves, and the trajectory is vertical. While the slider pushes the rotor, the configuration is a point on the cs-curve, thus the trajectory follows the cs-curve. Once the engagement is broken, only the slider moves; the trajectory is once again vertical.

In the physical world forces cause bodies to move, but in c-space forces cause the configuration to change. To compute dynamics directly from c-space we treat the configuration as a particle in the cs-plane and apply forces directly to this particle. Cs-curves act like physical surfaces that deflect the particle as it moves through c-space.

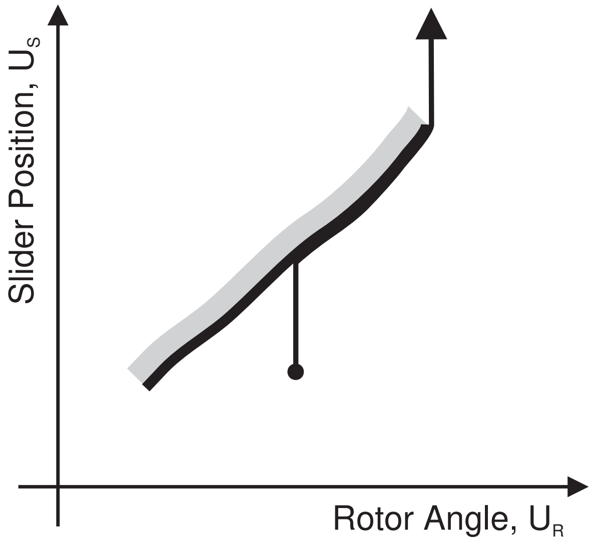


Figure 2-6: The motion of the rotor and slider can be represented as a trajectory through c-space. The initial configuration is shown as a dot.

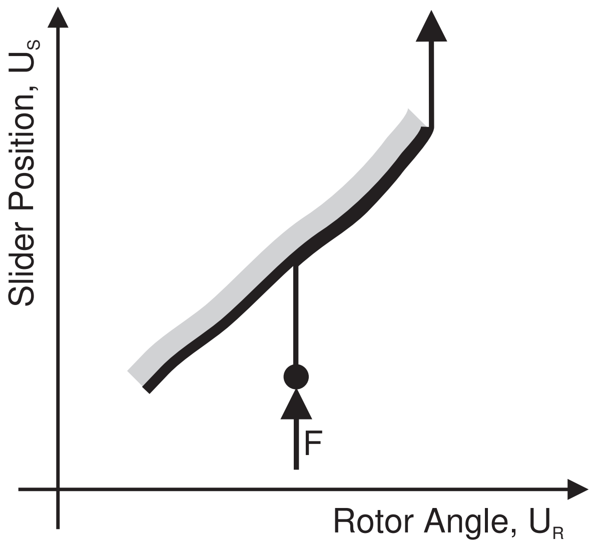


Figure 2-7: The motion of the rotor and slider can be computed by treating the *configuration* as a particle with forces applied to it.

Using the particle metaphor we can now compute the motion of the rotor and slider directly from their c-space, without any reference to the structure of the device. Figure 2-7 shows the particle we use to represent the configuration of the rotor and slider. The force applied to the slider appears in the cs-plane as a force in the direction of the slider's c-space coordinate (vertical). Similarly, any forces applied to the rotor would appear in the cs-plane in the direction of the rotor's c-space coordinate (horizontal). The initial configuration of the rotor and slider defines the initial location of the particle. The net force on the particle causes it to move vertically, until it strikes the cs-curve. The cs-curve deflects the particle as it continues to move upward. Finally, the particle reaches the end of the cs-curve and once again moves directly upward.

The example in Figure 2-7 illustrates a general principle: using the particle metaphor, it is always possible to compute the motion of the bodies in a device directly from the device's c-space description.

As this example has demonstrated, we make simplifying assumptions about the motion of bodies. We assume that collisions are inelastic, that motion is inertia-free, and that contacts are frictionless (we did not explicitly mention the frictionless assumption in this example). If the collision between the slider and rotor were actually elastic, the rotor would bounce off the slider and the trajectory in c-space might look like Figure 2-8. If there were appreciable inertia, the rotor would continue to move after it disengaged the slider, producing a trajectory like the one in Figure 2-9. If there were substantial friction between the rotor and slider, the slider might not be able to push the rotor and the device would jam.

We make these assumptions about the motion of bodies because they greatly simplify the trajectories through c-space and hence simplify reasoning about the trajectories. These assumptions are not overly restrictive: Sacks and Joskowicz [43] examined 2500 mechanisms in a catalog of mechanisms and found that 80% could be modeled accurately with these assumptions.

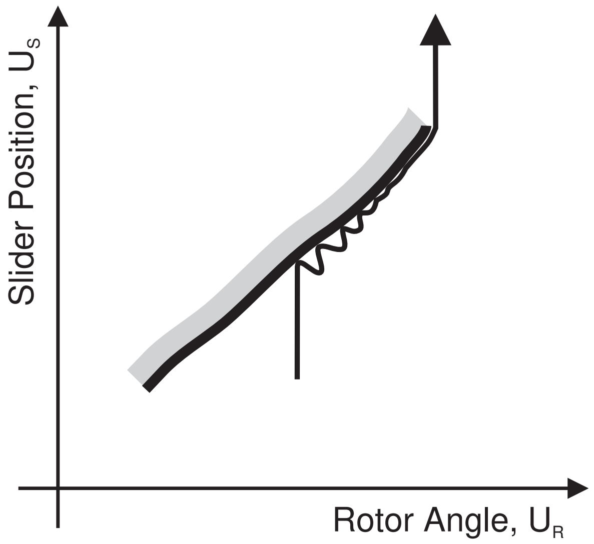


Figure 2-8: The trajectory of the rotor-slider device if collisions are elastic.

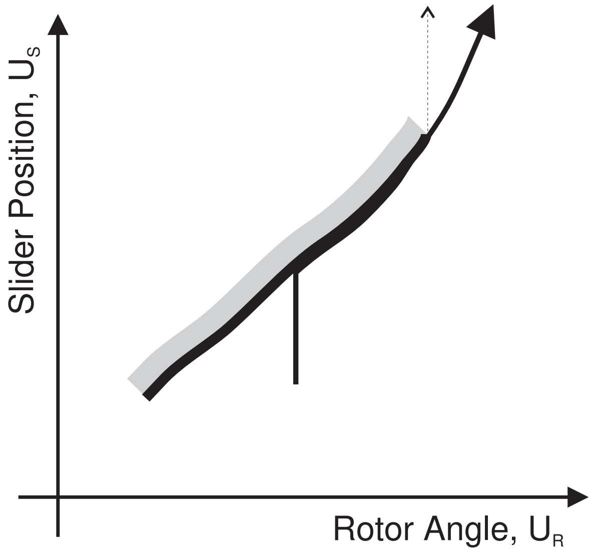


Figure 2-9: The trajectory of the rotor-slider device if there is appreciable inertia.

2.3 Qualitative Configuration Space

In the last section we showed how to represent in c-space the behavior of all of the components in SKETCHIT's domain. We also demonstrated that we can use c-space to reason about behavior: we can compute the overall behavior of a device directly from the c-space description of its parts. Hence, c-space has many of the properties we require of a behavioral representation.

To facilitate synthesizing new designs, we require a behavioral representation with one other essential property: it must generalize the design by abstracting away the implementation of the behaviors. As we mentioned at the outset, c-space does not adequately generalize the design, and hence in this section we extend c-space to produce a new representation that does. We call this new representation qualitative configuration space (qualitative c-space or qc-space).

To illustrate, we use the rotor-slider device from Figure 2-1. The cs-curve in Figure 2-2 represents the interaction between the face on the rotor and the face on the slider. Notice that if we extend the slider's face farther upward, we obtain the same cs-curve. Hence the particular cs-curve in Figure 2-2 represents many pairs of faces (we discussed just two of the possibilities) and in doing so abstracts away the particular geometry of the rotor and slider. However, the cs-curve is still too specific for our purposes.

To see why, we examine the c-space trajectory that describes the motion of the rotor and slider (Figure 2-6). As the trajectory moves vertically through the cs-plane, it is deflected to the right by the cs-curve. In physical terms, the slider pushes the rotor counterclockwise. The essential behavior—the slider pushing the rotor counterclockwise—is the physical manifestation of the cs-curve deflecting the trajectory to the right. Hence, any cs-curve that deflects the trajectory to the right will produce the same *qualitative* behavior (i.e., the slider pushing the rotor counterclockwise).

Consequently, to generalize the design far enough for our purposes, we want to identify a class of curves that displace the trajectory to the right. There are many curve shapes that would do this, but if we generalize only to the class of *monotonic* curves, we make tractable the task of computing trajectories through multi-dimensional c-spaces (see Chapter 4).[2] Hence, we generalize to monotonic curves.

In addition to its shape, a single cs-curve has two other properties that we can vary, the location of each end point. In this example there is only one constraint on the locations of the end points: one end point must be on either side of the initial vertical segment of the trajectory (otherwise the "particle" will miss the curve, i.e., the slider will miss the rotor). Hence, we generalize to monotonic curves that have one end point on either side of the initial segment of the trajectory.

Figure 2-10 shows our generalization of the c-space of the rotor and slider. We

[2] A multi-dimensional c-space occurs when a device is composed of many fixed-axis components.

represent the end points of the generalized cs-curve and the starting point of the trajectory with landmarks. Landmarks are quantities with constraints on their relative ordering, but which do not have absolute values. In Figure 2-10 for example, A, B, and C are landmarks on the U_R axis; all we know about their values is that A $<$ B $<$ C (which ensures that one end point of the curve is one either side of the initial segment of the trajectory). The generalized cs-curve is a monotonic curve with positive slope. (All of this assumes that blocked space is on the appropriate side of the curve as shown.) Any particular (i.e., numeric) cs-curve whose end points satisfy the conditions expressed by the landmark orderings, and whose shape is monotonic with positive slope, will produce the same qualitative behavior as the original.

Figure 2-10 is an example of what we call qualitative configuration space (qc-space). Qc-space, which is summarized in Figure 2-11, is SKETCHIT's primary representation of behavior. It inherits all of the desirable properties of c-space that we described in Section 2.2, and in addition adequately generalizes the design.

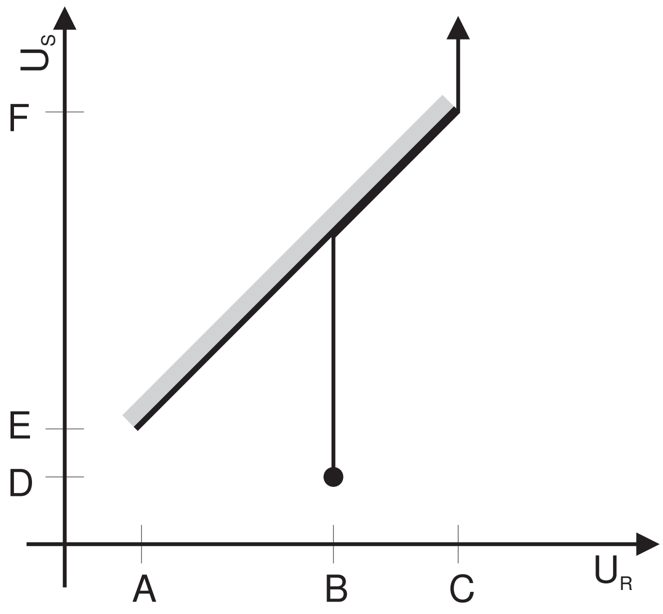


Figure 2-10: The generalization of the c-space for the rotor and slider. The cs-curve represents any monotonic curve, but is shown as a straight line for the sake of convenience. The values A through F are landmark values rather than specific numerical values. The landmark values are constrained to have the ordering shown.

In c-space the interaction between a pair of faces is a cs-curve, but in qc-space the interaction is a qcs-curve (i.e., a qualitative configuration space curve). A qcs-curve represents a family of monotonic cs-curves that all have the same qualitative slope. Qualitative slope is the obvious notion of describing curves as horizontal, vertical, Qualitative Configuration Space

- The interaction between a pair of faces is represented with a qcs-curve.
- Spring neutral positions, actuator motion limits, and interactions with fixed surfaces are represented by infinite boundaries.
- Qcs-curves are monotonic and have a qualitative slope.
- Landmark values denote the end points of qcs-curves and the axis crossings of infinite boundaries.
- The relative locations of qcs-curves and infinite boundaries are encoded by the ordering of the landmark values.

Figure 2-11: The Qualitative Configuration Space Representation.

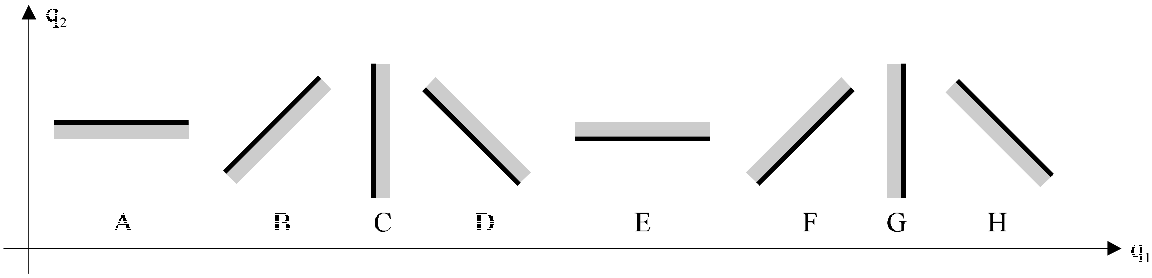


Figure 2-12: There are eight types of qcs-curves. For convenience, the diagonal curves are drawn as straight lines, but they represent any diagonal monotonic curve.

positively sloped, or negatively sloped. We often used the term diagonal slope to refer to both positively and negatively sloped curves. When the four qualitative slopes are combined with the two possible locations of blocked space, the result is the eight qualitatively different cs-curves in Figure 2-12.

In qc-space, the locations of qcs-curves and infinite boundaries are defined by landmark values: A pair of landmark values defines the location of each end point of a finite qcs-curve; one landmark value defines the axis crossing of each infinite boundary. The ordering of the landmark values encodes the relative locations of the qcs-curves and infinite boundaries.

Implicit in the qc-space representation is the assumption that each interaction is monotonic. This was true of the rotor and slider, but there are devices with non-monotonic interactions. Figure 2-13 shows one example in which the cs-curve is a piece of a sine curve. However, this interaction can be split into two monotonic interactions (one with negative slope and one with positive slope) by spitting the piston face into two faces. SketchIT requires that the user break non-monotonic interactions into a set of monotonic interactions.

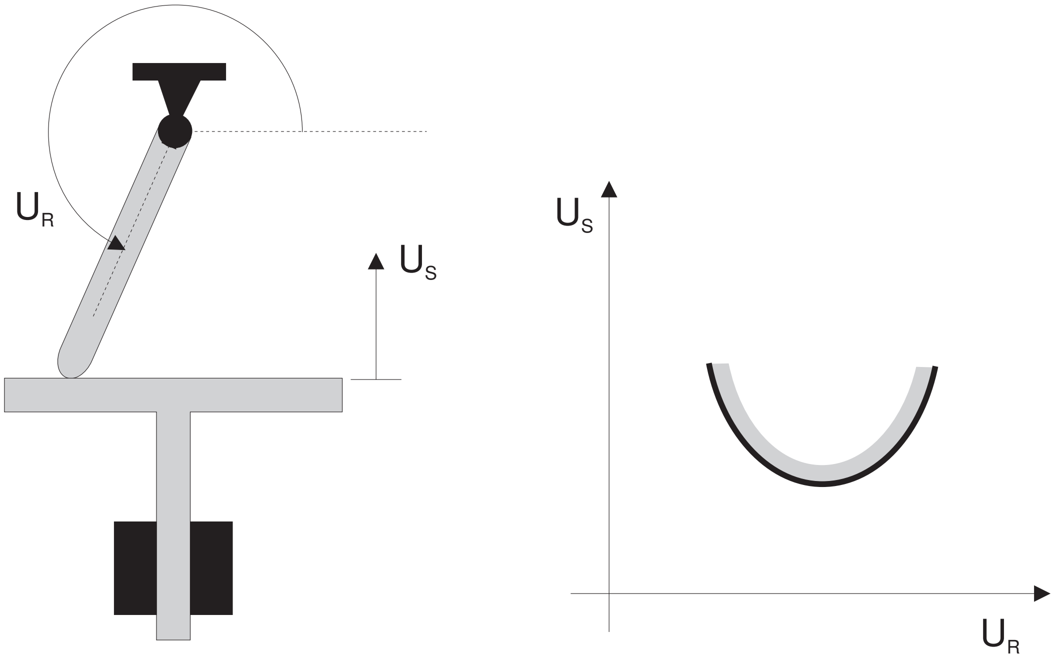


Figure 2-13: The c-space for this lever and piston is a section of a sine curve. If the face of the piston is split in half the c-space becomes two monotonic interactions.

2.4 Using QC-Space to Fix a Sketch

SketchIT uses qc-space as a tool for determining what behavior each part of the sketch ought to provide. The qc-space that SketchIT computes directly from the sketch is the program's first hypothesis about the behavior of each part. SketchIT tests this hypothesis by qualitatively simulating the overall behavior of the qc-space and comparing this to the desired behavior described with a state transition diagram (e.g., Figure 1-3b). If this qc-space produces the correct overall behavior, SketchIT is finished with this step. Otherwise SketchIT must modify the qc-space until it does produce the desired behavior (or until SketchIT determines that the qc-space is not capable of producing the desired behavior).

Figure 2-11 illustrates that there are only two ways of modifying a qc-space: changing the qualitative slope of a qcs-curve and changing the relative locations of two qcs-curves.[3] Hence the program searches for a working qc-space by applying just these two kinds of modifications. Once SKETCHIT finds a qc-space that produces the desired behavior, the program produces working geometry by using a library to map the qc-space back to real geometry. The library, described in Chapter 7, contains implementations for each kind of behavior that a qc-space can represent.

The remainder of this section provides an example of SKETCHIT in action, illustrating how it uses qc-space to transform a sketch into working geometry. The example, shown in Figure 2-14, consists of two translating blocks that interact through a pair of sloped faces. The desired behavior is for the spring to push block A against block B, driving B into the stop (the fixed surface).

For the particular geometry shown in the figure, the device does not work: block A cannot drive block B into the stop. The qc-space trajectory in the accompanying qc-space shows the actual behavior: the spring pushes block A into block B; A pushes B towards the stop; A and B disengage (before B reaches the stop); B comes to rest; and A continues to the right until the spring relaxes.

Because this qc-space does not produce the desired behavior, SKETCHIT must repair it. In this case, there is exactly one modification that will fix the qc-space (i.e., of all of the particular ways the program could apply the two general kinds of modifications, there is exactly one way that will fix the sketch): SKETCHIT must change the relative locations of the diagonal qcs-curve and the horizontal qcs-curve as shown in Figure 2-15.[4]

While there is only one particular modification that will repair the qc-space, there are many ways to map this modification to the geometry of the device. Figure 2-16 shows three possibilities: the face on A can be extended, the face on B can be extended, and the fixed surface can be lowered.

This example illustrates an important point: often many different changes to physical geometry map to the same change in qc-space. Thus, the number of ways to modify physical geometry is often much larger than the number of ways to modify qc-space. This is one of the reasons why qc-space is a convenient design space.

Also note the intuitive diagnosis of the failure "A doesn't push B far enough to reach the stop." In qc-space, "push" is a technically precise term meaning a diagonal qcs-curve. Similarly, "push far enough to reach the stop" means that the diagonal

[3]Changing the qualitative slope subsumes choosing which side of the curve is blocked space. For example, the slope of curve A in Figure 2-12 can be changed to produce curve E. In addition, for short hand convenience, we use the expression "the relative locations of the qcs-curves" to refer to the locations of both qcs-curves and infinite boundaries.

[4]At first glance, one might suggest two ways of repairing the qc-space: making the diagonal qcs-curve longer or lowering the horizontal qcs-curve. However, because locations in qc-space are relative rather than absolute, these are really the same repair, i.e., changing the relative locations of the diagonal qcs-curve and the horizontal qcs-curve (as shown in Figure 2-15.)

qcs-curve (representing the pushing interaction) must reach the horizontal qcs-curve (representing the stop). As this example illustrates, qcs-space represents behavior (e.g., "pushing" and "far enough") rather than implementation. Thus, by modifying qc-space we modify behavior directly. Once we find the desired behavior, we then map to implementation. As Figure 2-16 illustrates, there are many ways to map from behavior to implementation.

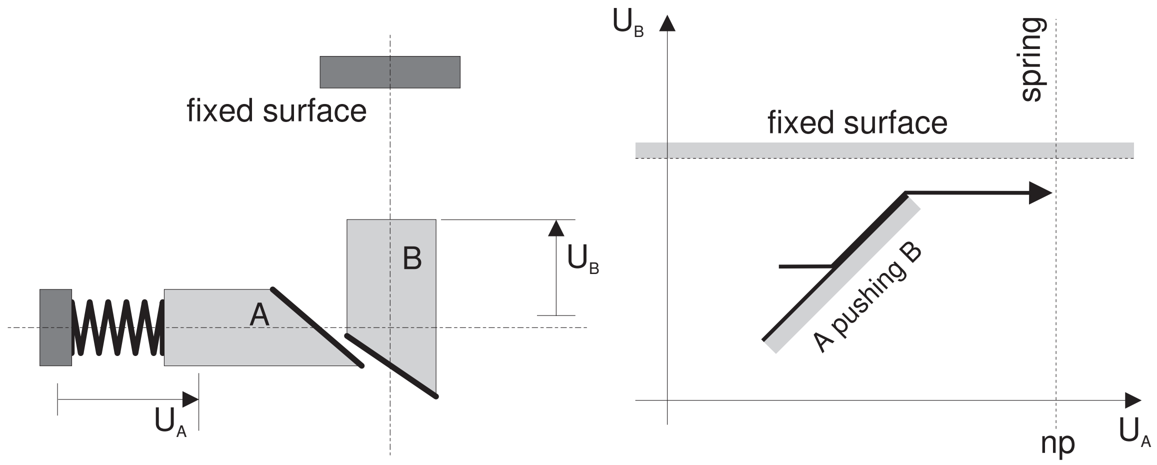


Figure 2-14: A simple mechanical device and its qc-space. Block A translates horizontally, B translates vertically, a spring pushes A to the right, B can bump into a fixed surface. The arrow in the qc-space shows the actual behavior of this device.

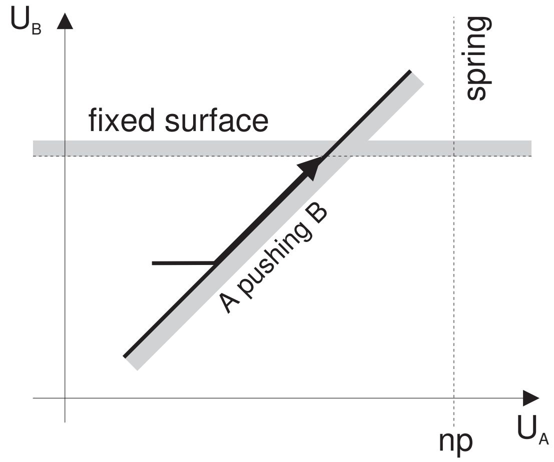


Figure 2-15: The qc-space of the two blocks modified to produce the desired behavior.

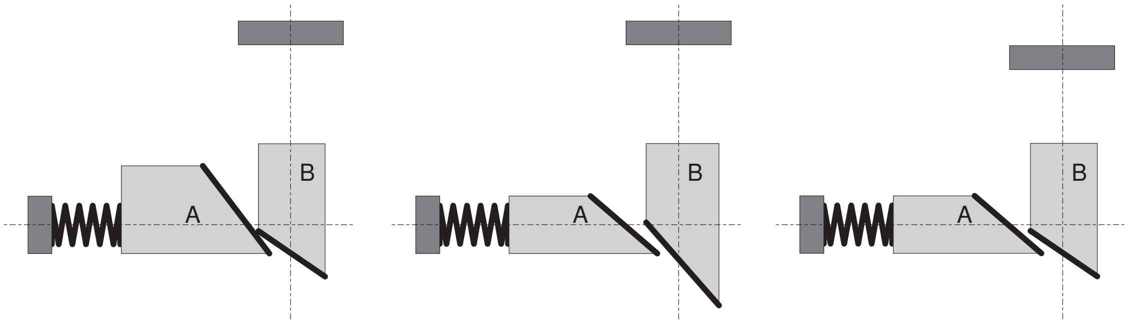
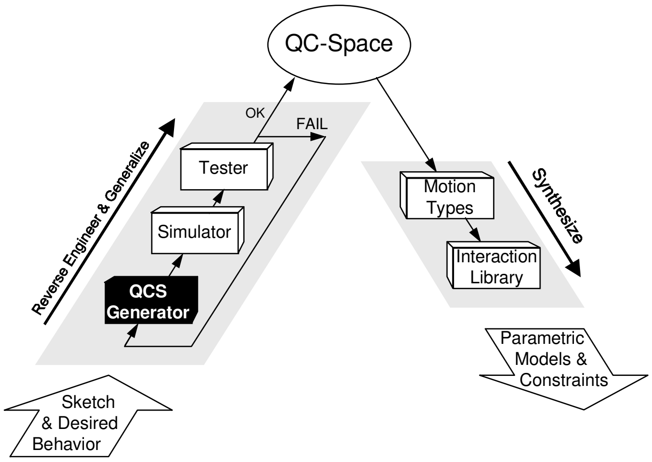


Figure 2-16: Three of the possible ways to implement the qc-space modification used in Figure 2-15: extend the face on A, extend the face on B, and lower the fixed surface.

Chapter 3

The QC-Space Generator



3.1 Introduction

SketchIT uses generate and test to reverse engineer the sketch of a device. The goal of reverse engineering is to determine what behavior each part ought to provide, and hence the generator's task is to produce hypotheses about the behavior of each part. The generator expresses each of its hypotheses as a candidate qc-space description of the design. For example, Figure 3-1 shows a candidate qc-space the generator produced for the circuit breaker from Chapter 1. To test a hypothesis, SketchIT qualitatively simulates the candidate qc-space and compares the result to the desired overall behavior.

This chapter describes how the generator produces candidate qc-spaces: The first candidate is an abstraction of the numerical c-space of the original sketch; the other candidates are modifications of the first.[1] There are only two kinds of modifications

[1]The generator performs a few tasks which are not essential to an understanding of the program's operation and which are more appropriately discussed in later chapters where their significance is apparent. See Sections 4.6.2, 4.6.3, and 4.7.

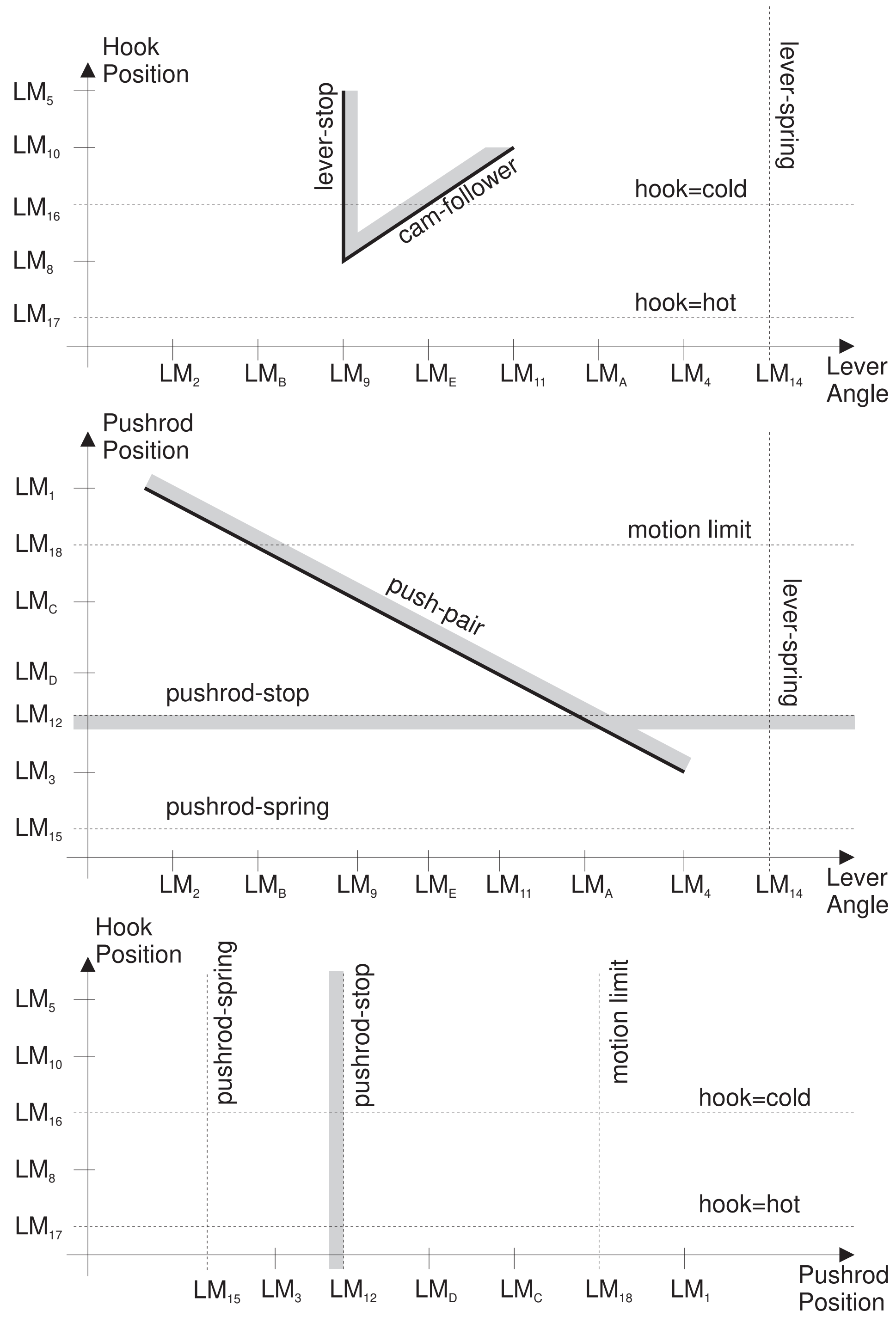


Figure 3-1: A candidate qc-space for the circuit breaker.

possible for a qc-space: changing the relative locations of two qcs-curves, or changing the qualitative slope of a qcs-curve.[2]

3.2 Abstracting C-Space to QC-Space

As noted, the first candidate qc-space is a qualitative version of the numerical c-space of the original sketch. A qc-space is defined by the qualitative slopes and relative locations of the qcs-curves. SKETCHIT obtains the qualitative slope of a qcs-curve by abstracting the slope of the corresponding numerical cs-curve: SKETCHIT computes the numerical slope of a straight line connecting the end points of the cs-curve, then matches this to one of the qualitative slopes in Figure 3-2. SKETCHIT takes the relative locations of the qcs-curves directly from the locations of the cs-curves in the numerical c-space diagram.

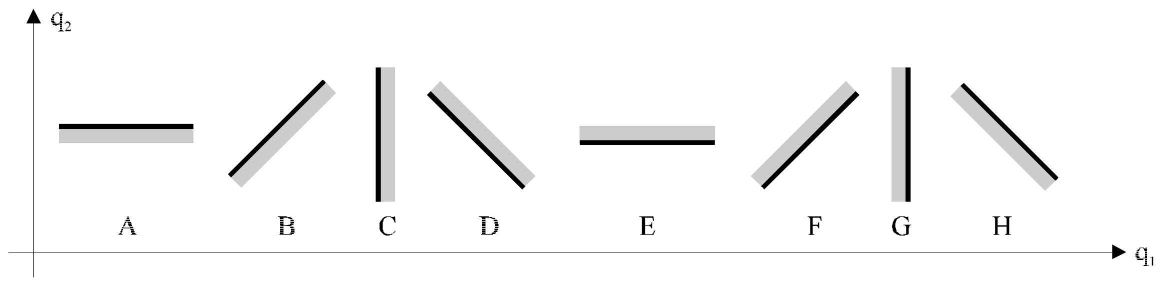


Figure 3-2: The eight different qualitative slopes. For convenience, the diagonal qcs-curves are drawn as straight lines, they can have any shape so long as they are monotonic.

We could have used general purpose techniques for computing configuration space (see, for example, [34], [1], [26], and [2]), but because SKETCHIT requires only the locations of the curve end points, we developed simplified, special purpose techniques that compute just this information. The simplifying insight was that for fixed-axis devices with flat faces, we need to handle only six different kinds of interactions: a rotating face interacting with a translating face, two rotating faces, two translating

[2]An implicit assumption in this discussion is that the number of parts in the device is fixed. If we relax this assumption, there are four additional ways to modify a qc-space: adding new qcs-planes by adding bodies to the device; adding new qcs-curves by adding pairs of engagement faces, springs, and actuators; deleting qcs-planes; and deleting qcs-curves. Relaxing this assumption would allow the program to generate a wider range of design variants. However, we use this assumption in order to limit search. In effect, we search only that part of the design space that is close to the original sketch.

faces, a rotating face interacting with a fixed face, and a translating face interacting with a fixed face.[3] Appendix C describes our techniques for two of these interactions.

3.3 Reducing Search

The brute force approach—exhaustively enumerating all possible values for the location and slope of each qcs-curve—is clearly impractical in any non-trivial design task. As a result we have identified two sources of knowledge that allow us to prune the search space: knowledge about the mapping from the sketch to qc-space and knowledge about trajectories in qc-space.

3.3.1 Mapping to QC-Space: Small Targets

There are circumstances in which small variations in the geometry of the sketch produce qualitative changes in the behavior. When the program detects these conditions, it asks the designer well focused questions about his intent.

One such circumstance occurs when a numerical cs-curve is nearly vertical or nearly horizontal. If the cs-curve really is vertical or horizontal, the interaction is stop behavior (i.e., one body limits the motion of another); but if the cs-curve is sloped even very slightly, the interaction is pushing behavior (i.e., one body pushes the other). A small change in the slope of the cs-curve produces a qualitative change in behavior. Thus, when the program identifies a numerical cs-curve that is nearly vertical or nearly horizontal, it asks the designer if the interaction is intended to provide stop behavior.

Another set of circumstances occurs when the end points of two numerical cs-curves are coincident, or nearly so. This is likely to have been intentional, because it corresponds to a common and useful piece of behavior: if two cs-curves connect end to end, the device can slip from one engagement pair to the next with no disengagement in between. Hence if two numerical cs-curves (nearly) connect end to end, the program asks the designer if a smooth transition between engagement pairs was intended.

The "V" in the top of Figure 3-1 illustrates the use of these questions in the circuit breaker example. The program notices that the cs-curve for the lever-stop engagement pair is nearly vertical and asks if we intended stop behavior; in this case we did. The program then asks if the end points of the lever-stop cs-curve and the cam-follower cs-curve are intended to be coincident; again they are, in order to allow the device to slip from one pair of engagement faces to the other without disengaging. Finally, the program notices that the other ends of the two cs-curves have a coordinate

[3] We implemented only the first and last of these, because that was enough for the three examples we used to test SketchIT, and because computing c-space curves is a relatively well understood problem that was not a focus of this work.

in common and asks us if this was intended, but this is just coincidental. The "V" in the figure reflects our answers to these questions.

The program asks questions when it detects small targets in c-space, but notice that these questions are very sharply focused. For example, when the program notices a vertical or horizontal curve, it does not ask the user "what did you mean?", but rather asks the very specific question, "is this intended to be a stop?"

The generator examines the c-space of the sketch for the presence of small targets before it abstracts the c-space to qc-space.

3.3.2 Debugging Rules

In Section 2.2.4 we demonstrated that the behavior of a device can be described as a trajectory through c-space. For example, Figure 2-6 shows the trajectory describing the behavior of a device composed of a rotor and slider.

Because we can describe the behavior of a device as a trajectory through c-space, we can describe the *desired* behavior as a *desired* trajectory. The topology of the c-space can have a strong influence on whether the desired trajectory (and hence the desired behavior) is easy, or even possible. This is equally true in qc-space.

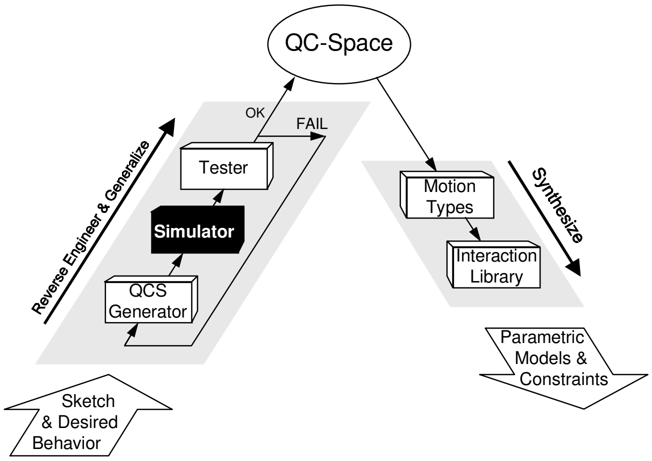
We have begun exploring the use of (but have not yet implemented) debugging rules that examine *why* a particular qc-space fails to produce the desired trajectory, based on its topology. If we can diagnose these kinds of failures to produce the desired trajectory, we may be able to generate a new qc-space by judicious repair of the current one. Several of the debugging rules we have explored are:

- If the qc-space has a funnel-like topology that "traps" the trajectory, the qcs-curves should be moved to eliminate the funnel.
- Any qcs-curve that is never touched by the trajectory is useless, and should be moved.
- If a trajectory terminates at a spring neutral position before it reaches a desired encounter with a particular qcs-curve, the neutral position should be moved to the other side of that qcs-curve.

Another kind of debugging strategy is to try to determine which qcs-curves are likely to be correct (i.e, provide a suitable behavior), so that the modifications can be focused on the other qcs-curves. One approach to this strategy is to simulate each arc of the state transition diagram individually to determine which arcs produce the desired transitions (i.e., for each state in the diagram, check if the external inputs for the arc leaving that state causes a transition to the next state). The qcs-curves used to achieve the desired transitions are likely to be correct. Modifications should be focused on the other qcs-curves.

Chapter 4

The Simulator



4.1 Introduction

Chapter 3 described the qc-space generator, which produces a sequence of candidate qc-spaces for the sketch. Each of these candidates describes one set of possible behaviors for each part of the sketch.

A qc-space is an implicit description of behavior: qc-space describes the kinematics of the device, that is, it describes all possible *positions* the device's parts can occupy. This chapter describes the simulator SKETCHIT uses to make explicit the behavior of a candidate qc-space. The behavior is characterized by the *motion* the device's parts exhibit in response to the external inputs specified by the state transition diagram.

The simulator computes all possible global behaviors of a candidate qc-space (i.e., all possible motions of the parts) so that the program can determine if the candidate is capable of providing the desired behavior. If so, during the synthesis process, the program will generate working implementations from the qc-space.

The simulator begins by identifying the forces on each part, then uses a qualitative version of Newton's laws to compute the motion of each part. That motion continues until some event (e.g., a collision) changes the nature of the forces. At this point the simulator stops, recomputes the forces, then continues simulating. Because this is a qualitative simulation, ambiguities can arise concerning what will happen next; our simulator produces an envisionment by determining all possible sequences of events.

To use Newton's laws (i.e., the Newtonian formulation of the equations of motion) one begins by drawing a free-body diagram for each body. A free-body diagram is a graphical depiction of a body in which it is isolated from the other bodies it engages by replacing the engagements with force vectors. Newton's laws then produce a system of algebraic and differential equations that relate the sum of the forces to the time rate of change of momentum of the body. The solution to this system of equations contains values for the forces and the motion of the body.

Unfortunately, Newton's laws do not work as easily with a qualitative representation. If we simply used qualitative versions of the engagement force vectors, there would be a significant amount of ambiguity in the force sums.[1] Hence, computing motion with a qualitative representation introduces new difficulties and requires special techniques. One technique we use is to represent each engagement by the type of constraint it applies to a body, rather than representing the engagement with an engagement force.

To illustrate what we mean by type of constraint, consider the three blocks in Figure 4-1. The spring pushes block A to the right, the actuator pushes block C to the left. In SKETCHIT's world, all actuators are assumed to be motion sources, that is, they assign position as a function of time. Block B, the block in the middle, experiences two engagement forces, one from A and one from C. Because the forces are in opposite directions, the qualitative sum of these forces is ambiguous.

However, we know that B will move to the left. Why is this? We know that the force C applies to B is whatever force is necessary for C to achieve its assigned motion. We call this kind of engagement a "motion constrained engagement" because it constrains the motion of the body to which it is applied.

The spring's deflection determines the spring's force on A. In an inertia-free world, A will transmit the spring force to B. Thus, A applies a known force to B. In contrast to a motion constrained engagement which assigns motion, this kind of engagement assigns force. We call this kind of engagement a "compliant engagement" because it assigns force in the same way that a compliant member (e.g., a spring) does.

One of our basic principles is that *a motion constrained engagement overpowers a compliant engagement.* Hence, because there is a motion constrained engagement pushing B to the left and a compliant engagement pushing B to the right, B must move to the left.

This example illustrates the kinds of techniques our simulator uses to compute

[1] Each component of a qualitative vector is a qualitative scalar. A sum of qualitative scalars is subject to ambiguity. Because a qualitative vector has many scalar components, there are many ways for a sum of qualitative vectors to be ambiguous.

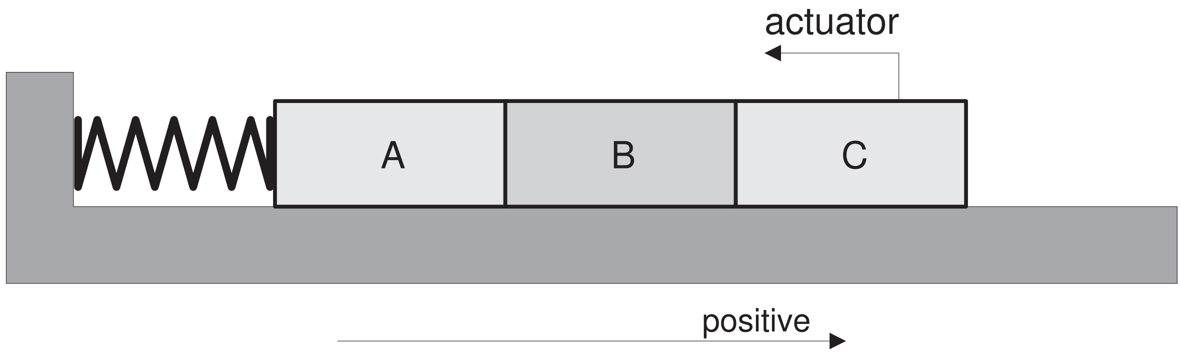


Figure 4-1: The three blocks slide along a frictionless horizontal surface. The spring pushes block A to the right, the actuator pushes block C to the left

the behavior of a device. In the following sections, we will describe the simulator in more detail, describing our representation for forces and engagements, our techniques for computing motion, and our techniques for determining when an event changes the motion. We initially focus on devices whose rotating parts turn less than a full revolution; then in Section 4.7 we describe additional techniques used to simulate devices containing parts that do rotate through full revolutions.

4.2 Forces

As the example in Figure 4-1 illustrated, computing motion in a qualitative world requires special techniques for reasoning about forces. In this section, we describe some of the special techniques we use. First we describe how to simplify forces by reasoning about their projections on the degrees of freedom of the bodies to which they are applied. Then we describe the particular issues involved in representing the two primary kinds of forces in SketchIT's world: spring forces and engagement forces.

4.2.1 Force Projections

Because we are restricting our attention to fixed-axis bodies, the only component of a force that has any effect on the motion of a body is the projection of the force along the degree of freedom of the body. Hence, we can simplify forces, without introducing inaccuracies, by reasoning only about their projections. The advantage of this simplification is that it greatly reduces the ambiguity in force sums.

We describe a projection with two properties, direction and magnitude. Because we are using a qualitative representation, we describe the direction as having one of the three standard values of `+`, `-`, or `0`. The direction is `+` if the force has a

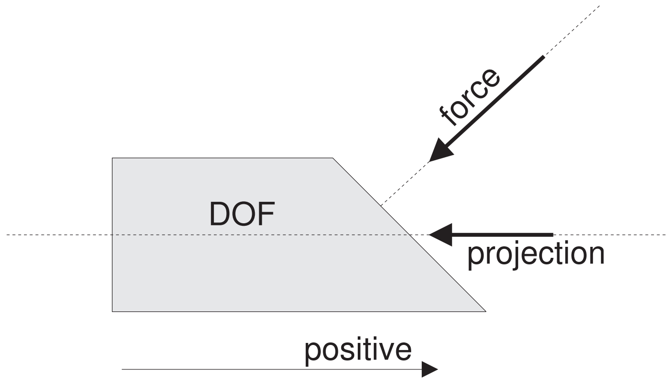


Figure 4-2: The block has a translational degree of freedom: it is constrained to translate along the horizontal dotted line.

component in the direction of positive motion, he direction is `-` if the force has a component in the direction of negative motion, and the direction is `0` if the force is perpendicular to the direction of motion. Similarly, we describe the magnitude as `0` if the force has zero magnitude, or `1` if the force has non-zero magnitude. For brevity we use the term qualitative force to refer to the projection of a force onto the degree of freedom of the body to which the force is applied.

Consider, for example, the force applied to the block in Figure 4-2. Because the block is constrained to move horizontally, only the horizontal projection of the force has any effect on the block's motion. Because the projection points in the negative direction and has non-zero magnitude, we describe the force qualitatively as (`-`, `1`).

The notion of projecting a force onto the degree of freedom of a body works just as well if the body rotates instead of translates. In this case, we first compute the moment of the force about the axis of rotation, then project this moment onto the axis of rotation.

A torque is the moment of a set of forces which produce no *net* force. An example is the torque that the stator of an electric motor applies to the rotor. The torque causes the rotor to spin, but there is no net force that would cause the rotor to translate. The projection of a torque onto the degree of freedom of a rotating body is, as one would expect, the projection of the torque onto the axis of rotation. The projection of a torque on the degree of freedom of a translating body is zero.

When computing the motion of rotating bodies, we reason about moments and torques; when computing the motion of translating bodies, we reason instead about forces. However, in both cases the same set of rules apply. Hence we express the rules in terms of "generalized forces." The term generalized force is a generic name for the class consisting of forces, torques, and moments. For brevity, we refer to generalized forces as simply forces.

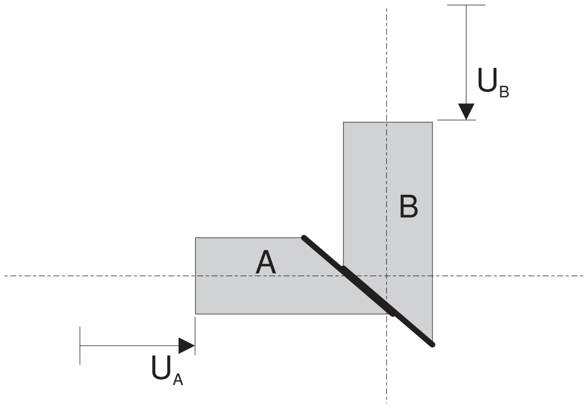


Figure 4-3: A translates horizontally, B translates vertically.

4.2.2 Spring Forces

In our world, springs always have one end fixed to ground and one end attached to a movable component. The spring is relaxed when the movable component assumes a specific location called the neutral position. We compute the magnitude of a spring force in the obvious way: If the movable component's position is greater than the neutral position, the spring applies a negative force (represented qualitatively as (`-`, `1`)). If the component's position is less than the neutral position, the spring applies a positive force (represented qualitatively as (`+`, `1`)). Finally, if a spring is relaxed, (`+, 0`), (`-, 0`), and (`0, 0`) are all valid qualitative representations of its force.

4.2.3 Engagement Forces

Engagement forces prevent one engagement face from penetrating the other. The direction of an engagement force is opposite the direction of motion that would cause penetration. For example, if block A in Figure 4-3 moves in the positive direction, it will penetrate B, assuming B does not move. However, if B applies a sufficient negative force, it will prevent the penetration. Thus, the engagement force B applies to A has a qualitative direction of "`-`."

Unlike a spring force whose direction changes with the state of the spring, the direction of an engagement force is a fixed property of the geometry. However, just as with a spring force, the magnitude of an engagement force may be either `0` or `1`.

The simulator determines which direction of motion causes penetration, and hence the direction of an engagement force, by examining qc-space. In qc-space (and c-space), configurations in which one object penetrates another are represented as blocked space. The motion that would cause penetration is the motion that would take the configuration into blocked space. For example, Figure 4-4 shows the qcs-curve for the interaction of blocks A and B from Figure 4-3. If A and B are engaged

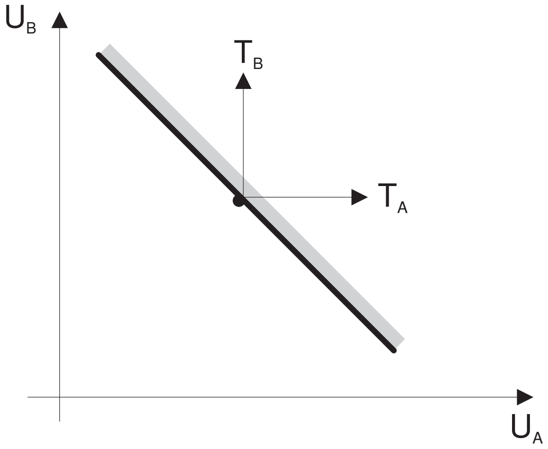


Figure 4-4: The current configuration is shown as a dot. T_A is a trajectory that would cause A to penetrate B. T_B is a trajectory that would cause B to penetrate A.

and A moves in the positive direction shown by trajectory T_A, the configuration will enter blocked space (i.e., the parts would penetrate). To prevent this penetration, B must apply a force in the negative direction. Hence the force has a qualitative direction of "-". Similarly, if B moves in the positive direction shown by trajectory T_B, the configuration will enter blocked space. To prevent this penetration, A must apply a force in the negative direction. Hence the force has a qualitative direction of "-".

Figure 4-5 shows the directions of the qualitative engagement forces superimposed on the qcs-plane. Because qualitative forces are the projections of the real forces onto the degrees of freedom, they are always parallel to the coordinate axes of the qcs-plane. The figure also shows the outward normal to the qcs-curve. Conveniently, the outward normal is always the vector sum of the directions of the two qualitative engagement forces. Hence, a simple way to compute the directions of the qualitative engagement forces is to take the vector components (i.e., the components along the qc-space axes) of the normal to the qcs-curve.

As an example of using the qcs-curve normal to compute force directions, consider the pair of blocks and qc-space shown in Figure 4-6. Because the normal to the qcs-curve has no vertical component, the engagement force that A applies to B has no component along B's degree of freedom. In our qualitative force vocabulary, the force has direction "0". Similarly, because the horizontal component of the normal is in the negative direction, the force of B on A has qualitative direction "-".

Notice that this analysis is consistent with the fact that the interaction is a stop (which can be determined from the fact that the cs-curve is vertical). For a stop, there is always one body that can move freely and one body whose motion is limited. The body that moves freely is the one that experiences the force with a qualitative direction of "0" (in this case, block B).

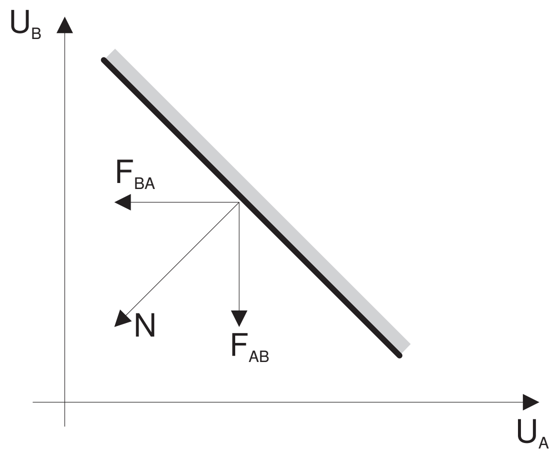


Figure 4-5: N is the normal to the qcs-curve, F_{AB} is the force of A on B, F_{BA} is the force of B on A. The normal is the vector sum of the directions of these two forces.

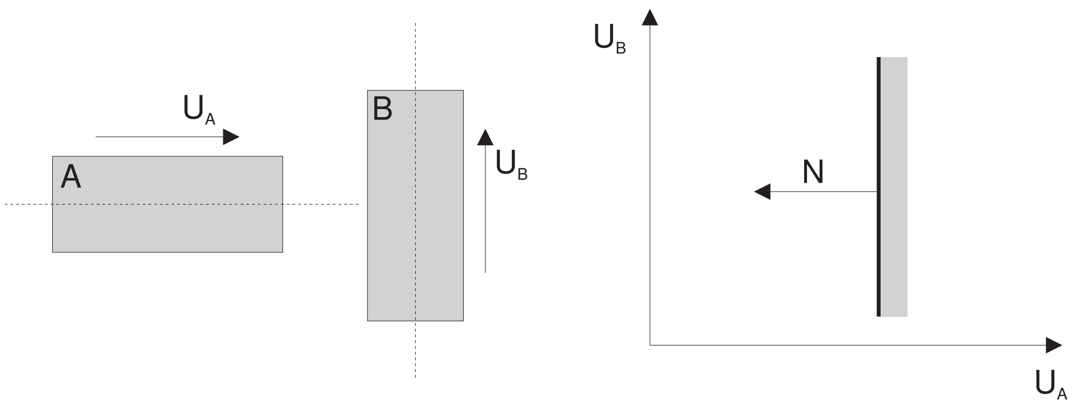


Figure 4-6: A pair of blocks and their qc-space. Block B translates vertically, A translates horizontally. N is the normal to the qcs-curve.

4.2.4 Engagement Types

In Section 4.1 we introduced the concept of engagement type. In this section we describe how the simulator determines the type of an engagement so that it can use the principles developed in Section 4.1 to determine the motion of an object from the types of its engagements. First, however, we must add to the concept of engagement type: Each engagement actually has two engagement types, one from the viewpoint of each of the two bodies. For example, in Figure 4-1 the engagement between B and C is motion constrained from B's viewpoint, and compliant from C's viewpoint.

The simulator begins determining engagement types by identifying the two kinds of engagements which are easily identifiable as motion constrained. First, if an engagement is a stop, it is motion constrained from the viewpoint of the body whose motion is limited. Figure 4-6 shows an example in which block B limits A's motion; the engagement is motion constrained from A's viewpoint. Second, if a body engages another body that is positioned by an actuator, the engagement is motion constrained from the viewpoint of the first body. In Figure 4-1, for example, the engagement between B and C is motion constrained from B's viewpoint because C is positioned by an actuator.

After the simulator identifies all of the easily identifiable motion constrained engagements, it uses propagation to identify the remaining ones. Figure 4-7 illustrates this propagation technique. Fixed surface S acts like a stop and applies a motion constrained engagement to A. P'_{mce} is the actual engagement force that the fixed surface applies to A, and P_{mce} is the qualitative version of this force (i.e., P_{mce} is the projection of the actual force onto A's degree of freedom). Similarly, F' and F are the actual and qualitative forces, respectively, that block A applies to B, and M' and M are the actual and qualitative forces, respectively, that block B applies to A. (The forces with primes are the actual forces, those without are the qualitative forces.) According to Newton's second law, F' and M' are equal and opposite.

Block A will apply a motion constrained engagement to B if A can propagate P'_{mce} to B. This requires that $P'_{mce} \cdot F' > 0$. Because F' and M' are equal and opposite, we can rewrite this expression as $P'_{mce} \cdot (-M') > 0$. Finally, by inspection of Figure 4-7 we see that $P'_{mce} \cdot (-M') > 0$ if and only if M and P_{mce} have opposite qualitative directions. Thus, to test if A applies a motion constrained engagement to B, we simply check if M and P_{mce} have opposite qualitative directions, which indeed they do.

We can generalize this example into a rule:

Motion Constrained Rule

To test if engagement E between body-1 and body-2 is motion constrained from body-2's viewpoint, identify the qualitative force M that body-2 applies to body-1 at engagement E. If there is a motion constrained engagement that applies a qualitative force P_{mce} to body-1, and M and P_{mce} have opposite qualitative directions (i.e., one is + and the other is -), then E is motion constrained from body-2's viewpoint.

The simulator applies this rule with a forward chaining rule engine. In effect,

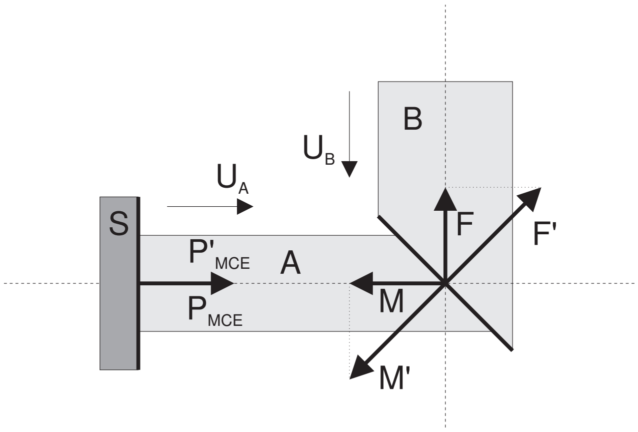


Figure 4-7: A translates horizontally, B vertically. Fixed surface S engages A.

the simulator examines all engagements and for each engagement examines both viewpoints. If the engagement type is unknown and the motion constrained rule is satisfied, the simulator labels the engagement as motion constrained from that viewpoint. The simulator repeats this procedure until no new labels are generated. If all of the motion constrained engagements produced by stops and bodies positioned by actuators are labeled to start with, this procedure will identify and label all of the remaining motion constrained engagements. Any unlabeled engagements must be compliant because there is no other choice.

To illustrate this procedure, we apply it to the blocks in Figure 4-1. We begin by labeling the simple cases: the engagement between B and C is motion constrained from B's viewpoint because C is positioned by a motion source. Using the propagation rule, we determine that the engagement between A and B is motion constrained from A's viewpoint. The rule cannot be applied to any of the other engagements. Hence, the engagement between A and B is compliant from B's viewpoint, and the engagement between B and C is compliant from C's viewpoint.

4.3 Interface Motion

In the example of Figure 4-1 we decided that block B will move to the left because there is a motion constrained engagement pushing it in that direction. Now imagine that the actuator attached to block C pushes C to the right instead of the left. C will still apply a position constraint to B, and thus the engagement is still a motion constrained engagement. The force of C on B will still be negative (i.e., the direction of the engagement force will still be -). However, block B will move to the right.

A convenient way to describe the difference between the two situations is in terms of the relationship between the direction of C's motion and the direction of the force

C applies to B: With the actuator moving to the left, they are in the same direction. In this case we say that C is applying an "advancing engagement." With the actuator moving to the right, C's motion and the force C applies to B are in opposite directions, and we say that C is applying a "retreating engagement." (If the actuator holds C fixed, we say that C is applying a "stationary engagement.")

Just as the type of the engagement is defined from the viewpoint of a particular body, the interface motion is defined from the viewpoint of a particular body. For example, when the actuator pushes C to the left in Figure 4-1, the engagement between B and C is advancing from B's viewpoint and retreating from C's.

Now we have four properties with which to describe an engagement: the interface motion—whether the engagement is advancing, retreating, or stationary; the direction of the engagement force—[`+`, `-`, or `0`]; the magnitude of the engagement force—[`0` or `1`]; and the type of the engagement—motion constrained or compliant. All four properties are necessary for computing the motion of a body.

4.4 Computing Motion

Because we use a qualitative representation, and because bodies are fixed-axis, we describe the motion (velocity) of a body with the standard values of `+`, `-`, or `0`, paralleling our representation for forces.

We have three basic principles for computing motion. First, the motion is always in the direction of the net force. This principle is derived from the inertia-free assumption (with inertia, the acceleration is in the direction of the net force but the velocity need not be). Second, a motion constrained engagement overpowers a compliant engagement. Now that we have four properties with which to describe an engagement, we can in fact be more precise about the second principle: An advancing motion constrained engagement causes motion in the direction of the engagement force; a stationary motion constrained engagement prevents motion opposite the engagement force; and a retreating motion constrained engagement allows motion opposite the engagement force, not faster than the speed of retreat. Third, a pair of engagement faces that are touching, but instantaneously separating, apply no engagement force to each other. This happens if the engagement is retreating from both viewpoints, or stationary from one viewpoint and retreating from the other.

Using these principles, we can derive all consistent combinations of applied engagements and the motion resulting from each consistent combination. Table 4.1 shows these results. For example, because block B in Figure 4-1 experiences a motion constrained, `-`, advancing engagement (from C) and a compliant, `+`, retreating engagement (from A), rule 8 tells us that B's motion is `-`.

The rules in the table that contain "BI" (bad input) in the motion column indicate that the user specified a physically unrealizable set of inputs. There are two kinds of physically unrealizable inputs: an actuator pushing a body into a stop, and two actuators pushing against each other. Any combination of engagements not listed in

rule	spring, external force		engagement												motion
			motion constrained						compliant						
			+			−			+			−			
	+	−	adv	sta	ret	adv	sta	ret	adv	sta	ret	adv	sta	ret	
1	○	○	□	○	○	□	○	○	○	○	○	○	○	○	BI
2	○	○	□	○	○		□	○	○	○	○	○	○	○	BI
3	○	○	□	○	○			□	○	○	○			○	BI / +
4	○	○		□	○	□	○	○	○	○	○	○	○	○	BI
5	○	○			□	□	○	○			○	○	○	○	BI / −
6	○	○		□	○		□	○		○	○		○	○	0
7	○	○	□	○	○				○	○	○			○	+
8	○	○				□	○	○			○	○	○	○	−
9	□			○	○			○	○	○	○			○	+
10	○			○	○			○	□	○	○			○	+
11		□			○		○	○			○	○	○	○	−
12		○			○		○	○			○	□	○	○	−
13		○		□	○		○	○		○	○		○	○	0
14	○			○	○		□	○		○	○		○	○	0
15				○	○		○	○		○	○		○	○	0

Table 4.1: A body's motion is consistent with the engagements it experiences if the motion and the engagements satisfy one of these rules. Any combination of engagements not listed is inconsistent. □= one or more. ○= zero or more. blank = none. adv = advancing, sta = stationary, ret = retreating. "BI" (bad input) indicates that the user specified incompatible inputs: an actuator pushing against a stop or two actuators pushing against each other. The magnitude of spring forces and external forces must be 1. The magnitude of engagement forces is unspecified.

the table is inconsistent.

If we know the properties of all of the engagements for a particular body, we can use the table to determine the motion of that body. The problem with this approach is that before we can use the rules in the table to determine the motion of a body, we must know the interface motion of every engagement applied to that body. The interface motion of an engagement depends on the motion of the body that applies the engagement. Hence, to compute the motion of a body, we must know the motion of every body which engages that body. This leads to a chicken and egg problem. Consider, for example, using the table to compute the motion of the blocks in Figure 4-1. We cannot compute the motion of block B until we know A's (and C's) motion, but we cannot compute A's motion until we know B's.

We can use search to alleviate this problem. For example, we could choose a motion (+, -, or 0) for each body, compute the resulting interface motion for each engagement, and check Table 4.1 to determine if each body's motion was consistent with its engagements. If there were inconsistencies, we would choose different motions and repeat.

We do use search, but the technique we actually use is much more efficient than the brute force approach just suggested. We use an approach variously referred to as "opportunistic search" or "constraint propagation with assumed states and back-tracking." The remainder of this section describes our search technique.

The basic principle behind our search technique is that if the simulator can, by some means, compute the motion of a particular body, the simulator may then be able to compute the motion of other bodies that that body engages. One can think of this as the propagation of motion from one body to the next. The simulator starts this propagation process by computing the motion of each body positioned by an actuator (the motion of the body is that of the actuator). Then the simulator attempts to propagate motion from the bodies with known motion to those they touch. Each time the simulator computes the motion of a new body, it repeats the process of propagating motion from the bodies with known motion to those they touch.

The propagation process may stall before the simulator has computed the motion of all of the bodies. To prevent stalling, the simulator assumes a value for the motion of bodies that match a particular criteria we describe below. After it has computed or assumed a value for the motion of each body, the simulator checks to see if the assumed motions are consistent with the engagements. If not, the simulator repeats the entire process, but assumes different motions for the bodies whose assumed motion was inconsistent.

The simulator's criteria for deciding when to assume (rather than compute) a motion depends on two quantities that we call the "mce-direction" and "k-direction" of the body. The "mce" in mce-direction stands for "motion constrained engagement." The "k" in k-direction comes from the variable commonly used to denote a spring constant, and hence is a mnemonic reference to the term compliant engagement.

The mce-direction represents the *directions* of all of the motion constrained engagement forces applied to a body: If all the motion constrained engagements are `+`, the mce-direction is `+`; if all the motion constrained engagements are `-`, the mce-direction is `-`; if there are both `+` and `-` motion constrained engagements, the mce-direction is $\pm$; and if there are no motion constrained engagements, the mce-direction is `0`. In a similar way, the k-direction represents the directions of all of the compliant engagement forces, spring forces, and external forces applied to a body.

Our search technique relies on six basic strategies for computing the motion of a body. These strategies are based on our assumption that it is always possible to compute the interface motion of motion constrained engagements, but sometimes it is not possible to compute the interface motion of compliant engagements. These assumptions are true of the examples we tried, but they are not in general true: it is not always possible to compute the interface motion of motion constrained engagements.[2]

[2] We discuss more general approaches later.

Strategies for computing motion:

1) The mce-direction and the k-direction represent the directions, not the magnitudes, of all of the forces applied to a body. Hence, even if the mce-direction and the k-direction are the same direction, that does not mean there is a net force (i.e., a non-zero force) in that direction. However in this case, the simulator assumes by default that there is a net force, and hence motion, in that direction. The simulator makes this assumption without checking if it already knows enough about the forces to actually determine their magnitudes.

2) If the mce-direction is opposite the k-direction, the simulator waits until it knows the interface motion of every motion constrained engagement before computing the motion of the body. (This follows from our assumption that it is always possible to compute the interface motion of motion constrained engagements.) If there is an advancing motion constrained engagement, the body's motion is in the mce-direction. If there is a stationary motion constrained engagement but no advancing ones, the motion is `0`. If there are only retreating motion constrained engagements, the simulator assumes that the motion is in the k-direction. Again, the simulator makes this assumption without checking if it already knows enough about the forces to actually determine if any of the forces in the k-direction have a non-zero magnitude.

3) If the mce-direction is $\pm$ it is likely that there is a bad input (e.g., an actuator pushing against a stop) and the simulator waits until it knows the interface motion of every motion constrained engagement so that it can apply rules 1–6 in Table 4.1 to determine if there actually is a bad input. If those rules do not apply, there are no bad inputs and the simulator recomputes the mce-direction by ignoring the retreating motion constrained engagements. To see why it does this, we return to the example in Figure 4-1. Imagine that the actuator pulls C to the right and there is no spring attached to A. In this case block B will not move. If we put the spring back, B will move to the right because of the spring force. We see that the retreating engagement force that C "applies" to B does not determine B's direction of motion. In general, if we ignore retreating engagements, we still compute the correct direction of motion. However, in this case, if the simulator ignores the retreating motion constrained engagements, it can compute the motion using strategy 1 or 2 (if the k-direction is $\pm$ see strategy 4).

4) If the k-direction is $\pm$, and the mce-direction is `+` or `-`, the simulator reduces this to one of the previous strategies by ignoring retreating compliant engagements and recomputing the k-direction. (Rule 3 describes why the simulator can safely ignore retreating engagements.)

5) If there is no force applied to a body, the body's motion is `0`.

6) If the body is positioned by an actuator, the body's motion is that of the actuator.

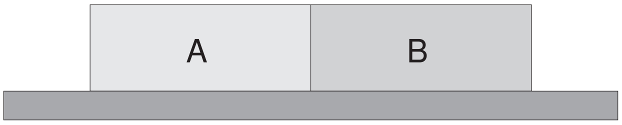


Figure 4-8: Two blocks, A and B, which translate horizontally.

After it has computed or assumed the motion of each body using these six strategies, the simulator checks to see if the assumed motions are consistent with the engagements. If not, the simulator repeats the entire process one more time, but assumes that motion is `0` for bodies whose previously assumed motion was inconsistent with the engagements. If the second attempt to compute motion produces inconsistencies, the simulator fails. For the examples we have tried the simulator always succeeded in computing motion. We chose our method because it is efficient and relatively simple, however, there are other available approaches which are complete as well as efficient (see [50]).

To illustrate our strategies, we compute the motion of the blocks in Figure 4-1 for the case in which the actuator pushes C to the left. Because there is an actuator pushing C, we compute C's motion with strategy 6. Block B has a motion constrained engagement pushing it to the left and compliant engagement pushing it to the right. Using strategy 2, B's motion is to the left because the motion constrained engagement from C is advancing. Similarly, we use rule 2 to determine that A moves to the left.

Now imagine that there is no actuator attached to C. In this case, we use strategy 1 and assume that C moves to the right, in the direction of the compliant engagement force from B. B experiences opposing compliant engagement forces, but because we know that the force from C is retreating, we use strategy 4 to determine that the net force on B is to the right (i.e., we ignore the force from C). Then, using strategy 1, we assume that B moves to the right. Similarly, using strategies 4 and 1, we assume that A moves to the right. Checking our assumptions, we find that all of the motions are consistent with the engagements.

For another example, consider the two blocks in figure Figure 4-8. The engagement between A and B is compliant from both viewpoints. Using strategy 1 we assume that A moves to the left. With a second application of strategy 1 we assume that B moves to the right. Checking our assumptions, we find that they are inconsistent: A cannot push B to the right if A is moving to left, and B cannot push A to the left if B is moving to the right. Hence, we must change our assumptions and try again. Using strategy 1, and remembering that the previously assumed motion was inconsistent, we assume that A's motion is `0`. Similarly, we use strategy 1 to determine that B's motion is `0`. This time our answer is consistent: each body experiences a stationary, compliant engagement which cannot cause motion.

If the k-direction is $\pm$ because the sum of the compliant engagement forces, spring forces, and external forces is genuinely ambiguous, strategy 4 cannot help us compute the motion. For example, if the actuator in Figure 4-1 is replaced by a spring pushing C to the left, the k-direction of all three blocks is $\pm$. We cannot compute the motion unless we know which spring force is larger. In this case, the simulator must branch to consider all three possibilities: A's spring force is larger, C's spring force is larger, the two spring forces are equal. Our simulator assumes that a body will not experience a genuinely ambiguous force sum, and hence the program returns failure if one does.

4.5 The Next Event

Because the motion of a rigid body depends on the applied forces, the motion changes when and only when there is a change in the nature of the forces. There are four kinds of events that can change the nature of the forces: an engagement is broken, an engagement is made, a spring passes through its neutral position, or an actuator reaches its limit. The first kind of event occurs when a trajectory in qc-space following a qcs-curve reaches the end of that curve;[3] the other kinds of events occur when the trajectory hits a new curve.

Because, in our world, the motion of every rigid body remains either strictly positive, strictly negative, or zero over a time step, all trajectories in qc-space are monotonic over a time step. Because the trajectory through a qcs-plane is a projection of a monotonic trajectory (i.e., a projection of the trajectory through the full multi-dimensional qc-space), the trajectory through a qcs-plane is also monotonic. To determine which events happen next, the simulator first determines which events would happen if the monotonic trajectory in each qcs-plane continued until an event occurs in that plane. Then, the simulator uses constraint propagation to determine which of the events predicted by the individual planes can happen first.

4.5.1 Events in the Plane

The simulator selects methods for computing the events in a qcs-plane based on the kind of trajectory in that plane. The simulator distinguishes three kinds of trajectories: trajectories along finite qcs-curves, horizontal or vertical trajectories, and diagonal trajectories.

[3]There are two ways that an engagement can break: the trajectory in the qcs-plane moves along the qcs-curve until it reaches the end, or the trajectory moves perpendicular to the qcs-curve. In the former case, the two engagement faces that comprise the engagement maintain contact pressure and slide along one another until one face slides past the other. Figure 4-10 shows an example in which the sloped face on the lever slides past the sloped face on the hook. In the later case, there is no contact pressure, and the two engagement faces separate without sliding. An example is the opening of a pair of electrical contacts. This kind of broken engagement is detected by examining the interface motion of the engagement faces (Section 4.8.1).

Horizontal and Vertical Trajectories

If the trajectory is horizontal or vertical, the simulator determines the next event by looking along the direction of motion to see which qcs-curve is reached first. In effect, the simulator computes the point of intersection, if any, between the trajectory and each qcs-curve in the plane. If the trajectory does not intersect any qcs-curves, then no event will occur in this plane (and thus the motion will continue without bound). If there are intersections, the simulator sorts them by landmark value to see which one is reached first.

Figure 4-9 shows an example of a horizontal trajectory in which the lever moves to the left and the hook remains stationary. The first event occurs when the trajectory reaches the cam-follower qcs-curve, that is, when the lever hits the hook.

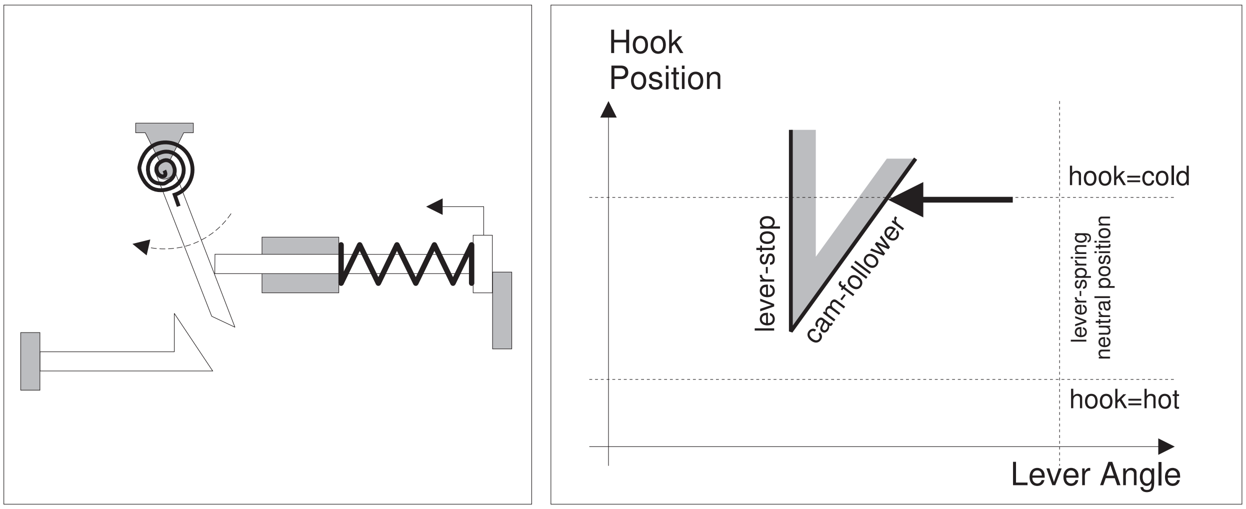


Figure 4-9: The reset input pushes the pushrod to the left, the pushrod pushes the lever clockwise, and the hook is relaxed at its cold neutral temperature (left figure). The trajectory in qc-space (thick arrow) is horizontal (right figure).

Trajectories Along a Finite Qcs-curve

If the trajectory in the plane follows a finite qcs-curve, the next event occurs when the trajectory reaches the end of the curve or when the trajectory reaches the intersection between that qcs-curve and another qcs-curve. We call the point of intersection between two qcs-curves an "i-point" (short for intersection point). Prior to computing a simulation, SketchIT computes the locations of all of the i-points. Section 4.6 describes the details of this process, but for our purposes here it is necessary to know only that the i-points exist.

Because the locations of the i-points are precomputed, the simulator computes the next event by simply looking along the trajectory to determine what is reached first, an i-point or an end point. If the trajectory reaches an i-point first, there is only one next event, namely the trajectory hitting the qcs-curve that intersects the engaged qcs-curve at that i-point.

If the trajectory reaches an end point first, there may be multiple next events. There is always at least one next event, namely the trajectory reaching the end of the engaged qcs-curve. Additional next events can occur if several qcs-curves have this end point in common (i.e., this point is the end point of multiple qcs-curves), because, as noted earlier, a trajectory reaching a new curve is an event. Hence, if an end point is reached first, the simulator checks if there are multiple next events.

Figure 4-10 shows an example of a trajectory along a qcs-curve. This trajectory occurs when the lever is moving to the left, pushing the hook out of the way. The first event occurs when the trajectory reaches the end of the cam-follower qcs-curve, that is, when lever slips off the left edge of the hook. Because the lever-stop qcs-curve has a common end point with the cam-follower qcs-curve, the simulator considers the lever-stop qcs-curve to be a new engagement, albeit only an instantaneous one. (In the next step of simulation, the motion computation will determine that the trajectory instantly leaves the end of the lever-stop qcs-curve.)

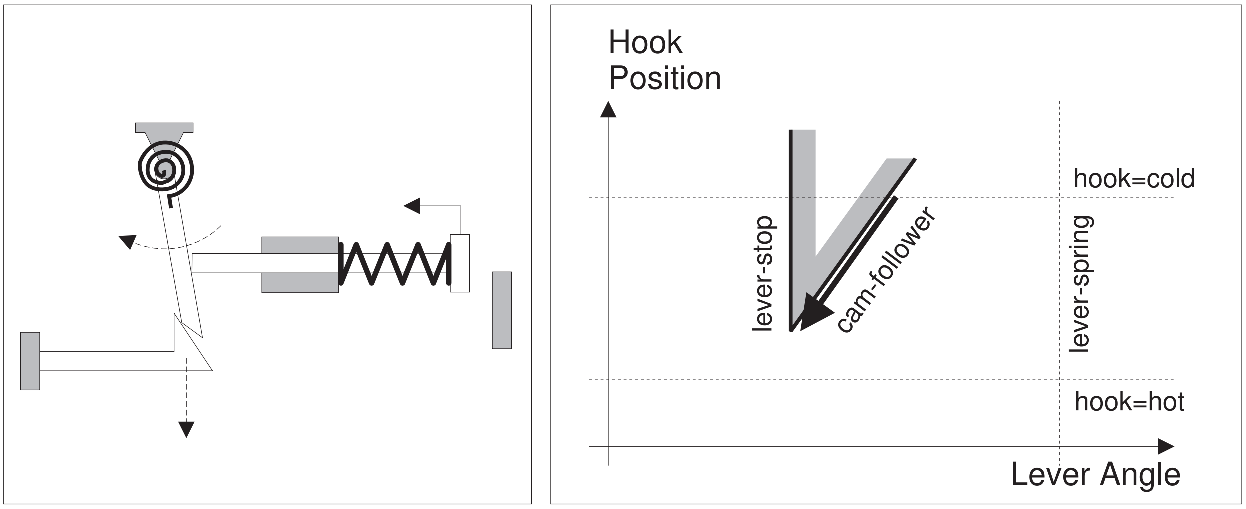


Figure 4-10: The reset input pushes the pushrod to the left, the pushrod pushes the lever clockwise, and the lever pushes the hook down (left figure). The trajectory in qc-space (thick arrow) follows the cam-follower qcs-curve (right figure).

Diagonal Trajectories

Consider next the situation in Figure 4-11: the trajectory is diagonal and does not follow a qcs-curve. To determine which qcs-curves the trajectory reaches, and hence what events happen next, we need to reason geometrically. The simulator uses a process of elimination to identify which qcs-curves can be reached: it determines geometrically all of the qcs-curves that cannot be reached; any remaining qcs-curves can be reached.

The simulator begins by defining a new coordinate system with its origin at the starting point of the trajectory. Because the trajectory is monotonic, it will remain entirely in one quadrant of the new coordinate system; we call this the active quadrant. Because the trajectory remains inside the active quadrant, it cannot reach any qcs-curve that lies completely outside the active quadrant. Thus, the first step in the process of elimination is to eliminate any curve that lies completely outside the active quadrant. Figure 4-11 shows an example in which the lever is moving to the right and the hook is moving upward. The first quadrant (i.e., the region above and to the right of the heavy black coordinates axes) is the active one.

Now imagine a "chain" of qcs-curves that divides the active quadrant into two regions, only one of which includes the origin. We call the region that includes the origin the active region. Figure 4-11 shows an example in which the chain is composed

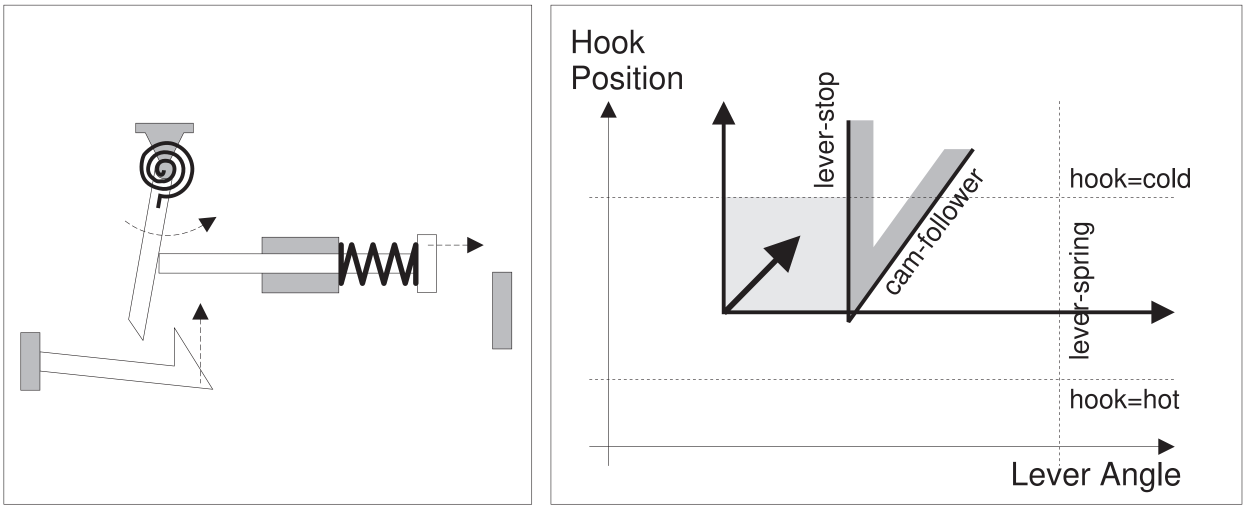


Figure 4-11: The pushrod-spring pushes the pushrod to the right, the lever-spring pushes the lever counterclockwise, and the hook moves towards its cold neutral position (left figure). The hook is high enough that the lever cannot miss it. The trajectory in qc-space (thick arrow) is diagonal (right figure). The active region of the qcs-plane is shaded.

of the lever-stop and the "hook=cold" qcs-curves; the active region is shown as a shaded rectangle. The trajectory cannot reach any qcs-curve that lies completely outside the active region.

There may be more than one chain that separates the active quadrant into two regions. For example, in Figure 4-11, there are three chains: the lever-stop and "hook=cold" chain, the cam-follower and "hook=cold" chain, and the lever-spring and "hook=cold" chain. We call the chain with the smallest active region the smallest chain. In this example, the lever-stop and "hook=cold" chain is the smallest. By definition, all other chains lie completely outside the active region of the smallest chain. Any qcs-curve that lies completely outside the smallest active region cannot be reached. Thus, the second step in the process of elimination is to eliminate any qcs-curves that lie completely outside the smallest active region.

To construct the chains, the simulator must know which qcs-curves intersect each other, and where those intersections are located. This is another reason why SKETCHIT precomputes the locations of all of the intersection points (i-points).[4]

When a qcs-curve is partially inside and partially outside the active quadrant, it casts a "shadow," a region that cannot be reached by a monotonic trajectory. Because the trajectory cannot enter the shadow, any qcs-curve whose intersection with the active region lies completely inside the shadow cannot be reached. Thus, the third step in the process of elimination is to eliminate any qcs-curves that are occluded by shadows.[5] Figure 4-12 shows an example in which qcs-curve C1 casts a shadow shown as a shaded region. Because qcs-curve C2 is inside that shadow, it cannot be reached by the monotonic trajectory.

To summarize, any qcs-curve that lies completely outside the active quadrant cannot be reached. If there is are chains that divide the active quadrant so as to define active regions, then any qcs-curve that lies completely outside that active region of the smallest chain cannot be reached. Finally, any qcs-curve that is occluded by a shadow cannot be reached. All of the remaining qcs-curves, including the ones that compose the smallest chain, can be reached.

In the example of Figure 4-11, all curves except the "hook=cold" and the lever-stop curves cannot be reached. There are three possible events: The trajectory can reach the lever-stop qcs-curve, in which case the lever strikes the hook; the hook can reach the "hook=cold" qcs-curve, in which case the hook relaxes; and the trajectory can reach the intersection of these two qcs-curves, in which case the lever strikes the

[4]See Section 4.6

[5]In its current implementation the simulator does not check if qcs-curves are occluded by shadows, and thus the simulator may predict events that are not physically possible. In the worst case, this could cause the program to accept solutions that do not produce the desired behavior. However, because shadows occur only infrequently (e.g., there were no shadows in the examples we tried), this deficiency in implementation should not cause any difficulties in practice. If this should in the future prove to be a source of difficulty, the program can be easily extended to check for occluded qcs-curves.

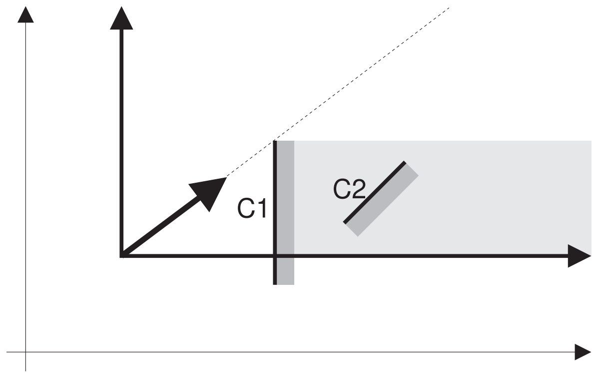


Figure 4-12: A diagonal trajectory in qc-space (thick arrow). Because qcs-curve C1 is partially inside the active quadrant and partially outside, it casts a "shadow" (shaded region). Because qcs-curve C2 is inside the shadow, it cannot be reached by the monotonic trajectory.

hook at the same instant the hook relaxes. Note that these three events are exactly the ones that common sense would predict if we performed a mental simulation of the device in Figure 4-11.

4.5.2 Computing Which Events Happen First

After computing the possible events for each qcs-plane, the simulator uses constraint propagation to determine which of these events can happen first. If, for example, the lever moves to the right in the lever-hook qcs-plane shown in Figure 4-13, then the lever must also move to the right in the lever-pushrod qcs-plane shown in Figure 4-14. If, for simplicity's sake, we assume the pushrod is at rest against the pushrod-stop, the next event in the lever-pushrod qcs-plane occurs when the trajectory reaches the push-pair qcs-curve, that is, when lever hits the pushrod. (The previous section describes the next events for the lever-hook qcs-plane.) Now that we have the next events for these two qcs-planes, lets examine some of the possible choices for which of these events can actually happen first.

Lets assume that in the lever-hook qcs-plane, the trajectory reaches the lever-stop qcs-curve (i.e., the lever hits the hook), and in the lever-pushrod qcs-plane the trajectory reaches the push-pair qcs-curve (i.e., the lever hits the pushrod). If the lever hits the hook, the lever's position must be landmark A. If the lever hits the pushrod, the lever's position must be landmark B which is greater than landmark A. Because the lever cannot be in two places at once, we must conclude that these two events, the lever hitting the hook and the lever hitting the pushrod, cannot both happen.

Now assume that the lever hits the hook but does not yet reach the pushrod. As

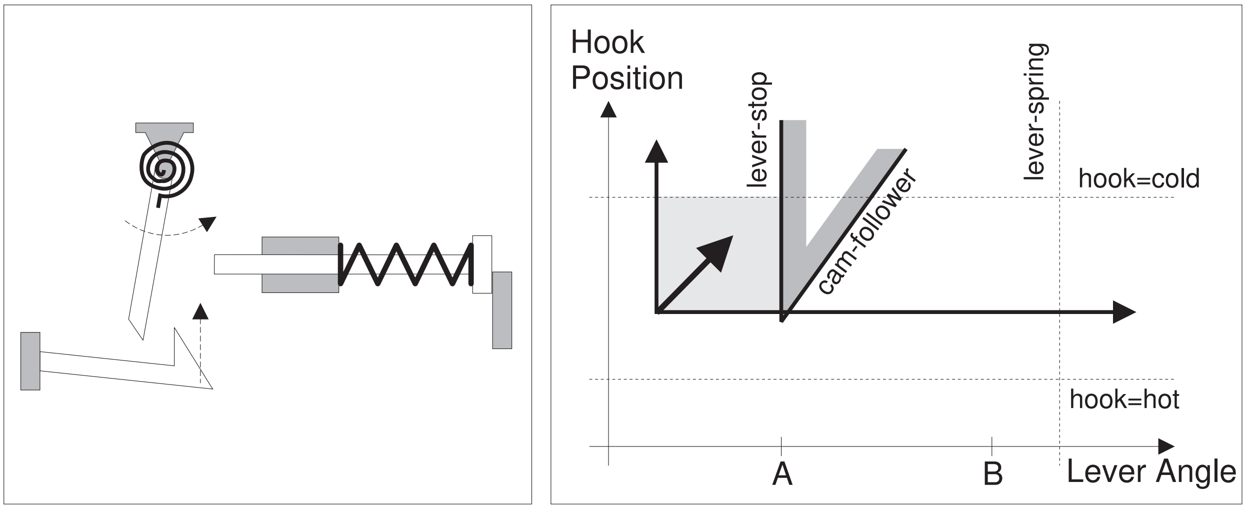


Figure 4-13: The pushrod is at rest against its stop, the lever-spring pushes the lever counterclockwise, and the hook moves towards its cold neutral position (left figure). The hook is high enough that the lever cannot miss it. The trajectory in qc-space (thick arrow) is diagonal (right figure). The active region of the qcs-plane is shaded.

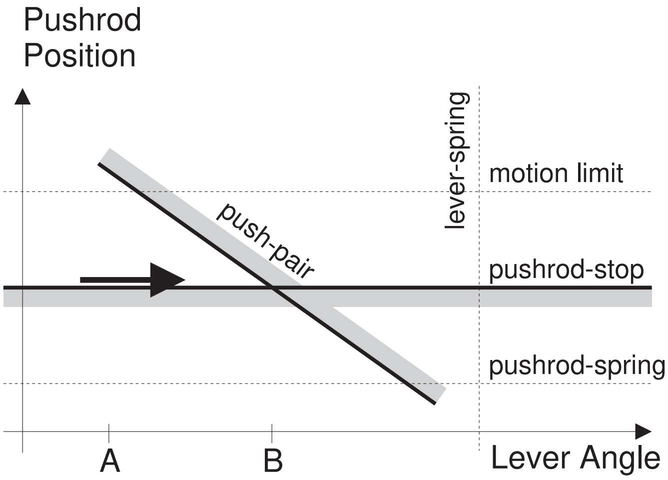


Figure 4-14: When the lever moves to the right in the qcs-plane of Figure 4-13, it must also move to the right in this qcs-plane.

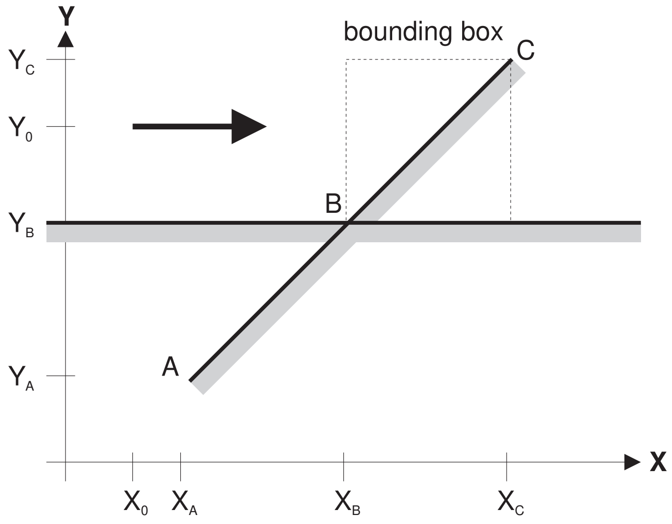


Figure 4-15: A horizontal trajectory (thick arrow) in the qcs-plane. Because the diagonal qcs-curve is monotonic (although we draw it as a straight line for convenience), the part of the curve between points B and C can be located anywhere inside the bounding box. The next event occurs when the trajectory strikes the diagonal qcs-curve, requiring that the trajectory ends inside the bounding box. For no event to occur, the trajectory must not reach the bounding box.

before, if the lever is to hit the hook, the lever's position must be landmark A. For the landmark to not reach the pushrod, the lever's position must be to the left of landmark B. Because landmark A is to the left of landmark B, our assumptions are consistent. Hence, we conclude that the next event occurs when the lever strikes the hook. In this case, an event occurs in the lever-hook qcs-plane, but none occurs in the lever-pushrod qcs-plane.

To compute which events can happen next, the simulator picks one event, possibly a null event, for each qcs-plane and checks if this set of events places consistent constraints on the positions of the bodies. If the constraints are consistent (i.e., the bodies can all be in the required locations at the same time), this set of events can happen simultaneously, and hence this set of events can happen next. The simulator repeats this process for each possible combination of events obtained by choosing one event for each plane.[6] If there is more than one consistent set of events, the simulator generates them all, producing an envisionment.

The position constraints for an event are simple equalities and inequalities between landmarks. Consider, for example, Figure 4-15 which shows the qc-space trajectory

[6]The simulator does, however, throw out the combination that consists only of null events, because that combination corresponds to the current state.

of two bodies whose positions are X and Y respectively. The trajectory is horizontal and starts from the point (X_0, Y_0). The first event occurs when the trajectory reaches the diagonal qcs-curve. For this event to occur, the trajectory must be somewhere in the bounding box of the portion of the qcs-curve between point B and point C:

$X_B < X < X_C$

$Y = Y_0$

For the null event to occur, the trajectory must not yet reach the bounding box:

$X_0 < X < X_B$ and $Y = Y_0$

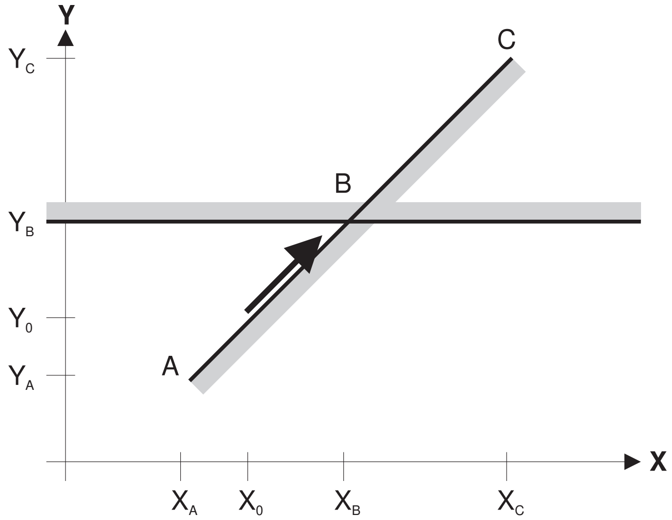


Figure 4-16: A diagonal trajectory (thick arrow) along the diagonal qcs-curve. The next event occurs when the trajectory reaches the horizontal qcs-curve at point B.

As a second example of constraints, consider Figure 4-16 which shows a trajectory following a finite qcs-curve. The first event occurs when the trajectory reaches i-point B:

$X = X_B$ and $Y = Y_B$

For the null event to occur, the trajectory must not yet reach i-point B:

$X_0 < X < X_B$ and $Y_0 < Y < Y_B$

As a third example of constraints, consider the diagonal trajectory shown in Figure 4-17. There are six possible events.

- The trajectory can reach C1:

  $X_0 < X < X_C$ and $Y = Y_C$

  The point $X = X_0$, $Y = Y_C$ is not within the constraints because the trajectory would have to be vertical, not diagonal, to reach this point. The point $X = X_C$, $Y = Y_C$ is not within the constraints because that is a separate event in which the trajectory simultaneously reaches C1 and C2.

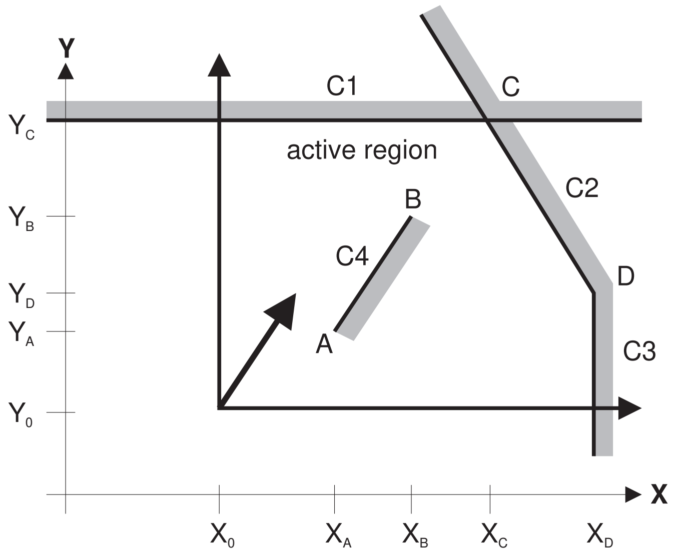


Figure 4-17: The diagonal trajectory (thick arrow) is enclosed by a chain consisting of qcs-curve C1, C2, and C3. Qcs-curve C4 is inside the active region of the chain.

- The trajectory can reach qcs-curve C2. For this to happen, the trajectory must be in the bounding box of the portion of C2 between points C and D:

  $X_C < X < X_D$ and $Y_C < Y < Y_D$

- The trajectory can simultaneously reach C1 and C2:

  $X = X_C$ and $Y = Y_C$

- The trajectory can reach C3:

  $X = X_D$ and $Y_0 < Y < Y_D$

- The trajectory can simultaneously reach C2 and C3:

  $X = X_D$ and $Y = Y_D$

- The trajectory can reach C4. For this to occur, the trajectory must be inside the bounding box. In this case, because there are no other qcs-curves with end points coincident with the end points of C4, the constraints include the end points:

  $X_A \leq X \leq X_B$ and $Y_A \leq Y \leq Y_B$

In this example, the constraints for the null event require a little more work because of the irregular shape of the active region, and the presence of qcs-curve C4 in the middle of the active region. Here, the simulator constructs the constraints

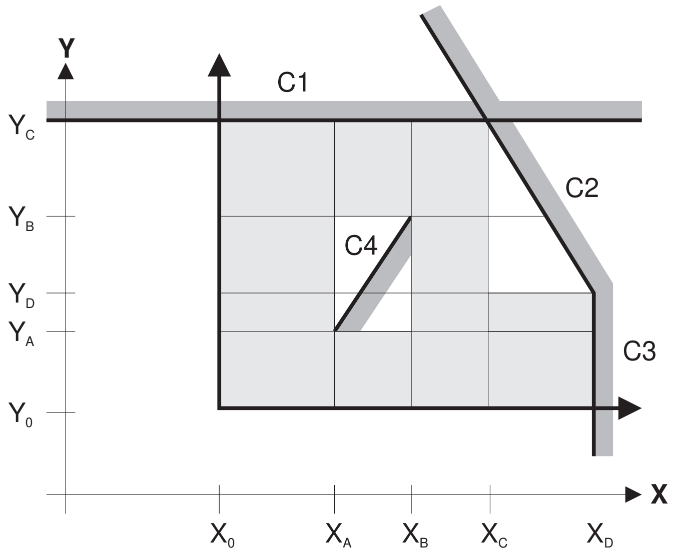


Figure 4-18: To compute the position constraints for the null event, the simulator tiles the active region of the chain. Each shaded tile is a range of positions consistent with the null event.

as a disjunction of simple constraints. This process is equivalent to tiling the active region as shown in Figure 4-18. Each tile contributes one pair of constraints to the disjunction. For example the lower left tile contributes:

$X_0 < X \leq X_A$ and $Y_0 < Y \leq Y_A$

and the tile second from the bottom, on the left contributes

$X_0 < X < X_A$ and $Y_A < Y \leq Y_D$

The simulator uses a very simple tiling technique: There is one column of tiles for each horizontal landmark from an end point or i-point that lies inside the active region or is part of the smallest chain. Here there are four such landmarks: X_A, X_B, X_C, and X_D. Similarly, there is one row of tiles for each vertical landmark from an end point or i-point that lies inside the active region or is part of the smallest chain. Here there are four such landmarks: Y_A, Y_B, Y_C, and Y_D. For ease of tiling, the simulator assumes that the active region is convex.[7]

[7]The constraints that the simulator computes from the tiles may erroneously include some 0-dimensional regions and erroneously exclude some 1-dimensional regions. As an example of the former error, point (X_A,Y_A) is counted as part of the null event (i.e., as part of the lower left tile) but is really part of the event in which the trajectory reaches qcs-curve C4. As example of the latter error, the left edge of the bounding box of qcs-curve C2 (excluding point (X_C,Y_C)) should be counted as part of the null event but is not. We can prevent these kinds of errors with more careful bookkeeping; we have not done this because these kinds of errors do not occur in the examples we tried.

For a diagonal trajectory for which there is no chain to define an active region, the simulator can still use the tiling approach to compute the constraints for the null event. To do this, the simulator defines two artificial infinite boundaries located at infinity. This guarantees that there will be a chain. The simulator also uses the artificial chains for determining which qcs-curves cannot be reached by the trajectory (see Section 4.5.1): any qcs-curve that lies completely outside the active region of the artificial region cannot be reached. Figure 4-19 shows an example in which qcs-curve C1 cannot be reached because it lies completely outside the active region of the chain composed of the artificial horizontal boundary and infinite boundary C2.

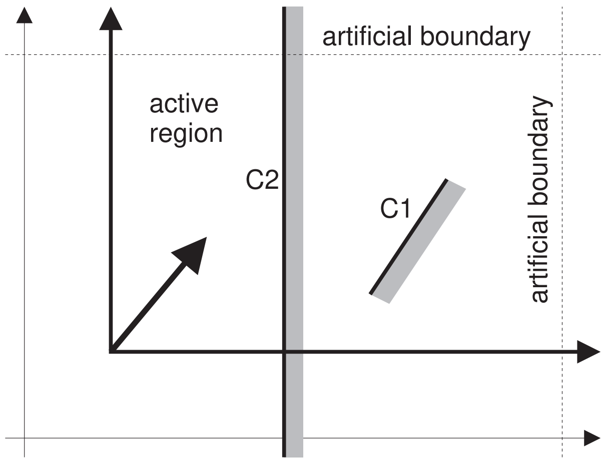


Figure 4-19: The two artificial boundaries are located at infinity. The horizontal, artificial boundary and qcs-curve C2 define the active region. Qcs-curve C1 cannot be reached by the trajectory (thick arrow) because C1 is outside the active region.

For each consistent set of events, the simulator's solution to the constraints defines the positions that the bodies occupy when that set of events occurs. These positions are the initial conditions for the next step of simulation. To consider all possible qualitatively distinct initial conditions, and hence all possible motions, the simulator must find all tiles that are consistent with a given set of events. For each tile that is part of a consistent set of events, the simulator must branch the simulation.[8]

When there is a trajectory following a diagonal qcs-curve, the net constraint for a particular choice of events may be satisfiable only for particular shapes of the diagonal

[8]Because of the simple tiling technique we use, this can lead to unnecessary branching. For example, if each of the tiles in the left column in Figure 4-18 is consistent with a particular set of events, the simulator will have to grow four branches. However, that column could be easily merged into a single tile, eliminating three branches. Because the examples we tried required only a very small number of tiles, there was no problem with excessive branching, and hence we did not implement a subroutine to merge tiles.

qcs-curve. This happens when the range of allowed positions (i.e., the positions consistent with the constraints) is a subset of the interior of the diagonal curve's bounding box, and that subset contains none of the curve's i-points. Figure 4-20 shows an example in which the curve has no i-points and the net constraint is shown as a rectangle. For the constraint to be satisfied, the curve must pass through the rectangle. If, for example, the curve has the shape shown by the dotted line, the net constraint cannot be satisfied. For the examples we have tried, there were no situations that required constraints on the shapes of the diagonal curves.

The simulator does not currently keep track of the kind of constraint on shape required in the example of Figure 4-20, and hence, if this kind of constraint is required for a particular device, the final BEP-Model the program generates will be insufficient to ensure the desired behavior. A simple remedy is for the program to reject any designs that require these constraints. A better remedy is for the program to keep track of these constraints and add them to the BEP-Model for the device.

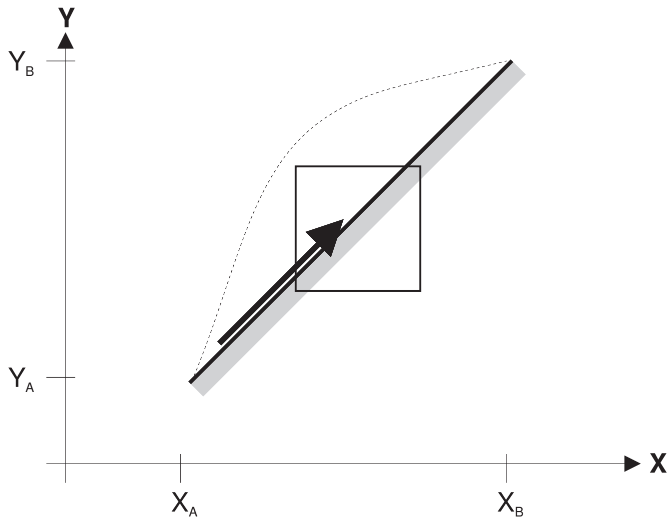


Figure 4-20: The trajectory (thick arrow) follows the diagonal qcs-curve. The rectangle shows the range of positions consistent with a particular choice of next events.

4.5.3 Unordered Landmarks

Section 4.5.1 discussed how the simulator computes the next events for each qcs-plane. This section attends to one additional detail necessary to complete that discussion.

Sometimes the landmarks for the initial positions of the bodies (i.e., the positions at the beginning of a step of simulation) may not be sorted with respect to the landmarks of the end points and i-points of the qcs-curves. (Recall that an i-point is the intersection between two qcs-curves.) When this happens, the simulator may have to enumerate the possible orderings before computing the next events in each qcs-plane.

Consider, for example, a pair of bodies whose qcs-plane is shown in Figure 4-21. Imagine that on step N-1 the trajectory is diagonal. Imagine further that step N-1 results in the null event in this plane, i.e., the trajectory does not hit any qcs-curves. In this case the trajectory must end somewhere inside the shaded region (i.e., the active region). Now imagine that on the next step, step N, the trajectory is horizontal and to the left. Because the initial position can be anywhere in the shaded region, the trajectory is not located with respect to qcs-curve C: the trajectory could hit the curve or pass above or below it. As a result, the simulator must branch to consider all three possibilities.

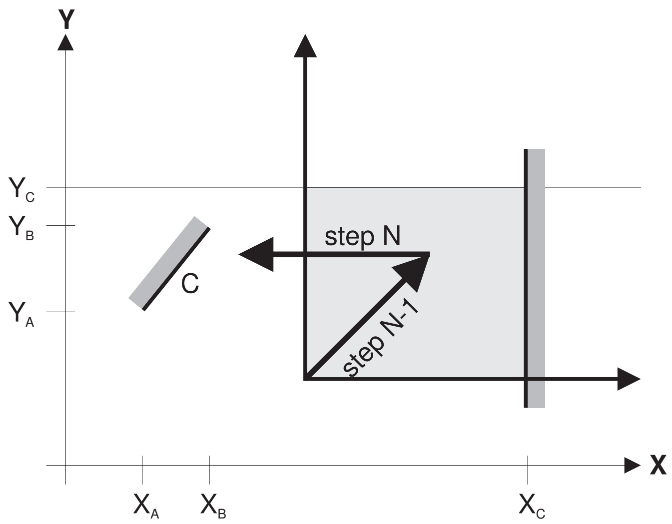


Figure 4-21: The trajectory through a qcs-plane for two consecutive time steps. During step N-1, the trajectory is diagonal with an active region shown by the shaded rectangle. During step N the trajectory is horizontal.

Now imagine instead that during step N the trajectory is vertical rather than horizontal. Because the shaded region is to the right of qcs-curve C, and the new trajectory is vertical, the trajectory cannot reach C. Hence, in this case, even though the landmarks for the initial position are not sorted with respect to the landmarks of qcs-curve C, the simulator need not branch.

As this example illustrates, the simulator may sometimes have to branch to consider possible landmark orderings. The general rule is: If the trajectory is in the direction of a qcs-curve, and the landmarks of the starting point of the trajectory are not ordered with respect to some of the landmarks of the end points or i-points of the qcs-curve, the simulator must branch to consider all possible orderings.[9]

4.6 Intersection Points

In Section 4.5.1 we described how the simulator computes the next events in a qcs-plane. As we noted there, the simulator must know the locations of all of the intersections between qcs-curves (i-points) in order to compute the next events. In this section we describe how the simulator computes the location of the intersection points.

The intersection between a horizontal and a vertical qcs-curve is a well defined point: The x-coordinate of the point is the landmark defining the horizontal position of the vertical curve, and the y-coordinate is the landmark defining the vertical position of the horizontal curve. Unfortunately, because finite qcs-curves are monotonic rather than straight lines, the location of the intersection between a diagonal qcs-curve and any other kind of qcs-curve may be ambiguous.

In fact, if two diagonal qcs-curves intersect each other, the potential ambiguity is so great that SketchIT does not allow two diagonal qcs-curves to intersect (except at their end points).[10] In the next section we describe the sources of ambiguity and how the program prevents this ambiguity by ensuring that two diagonal qcs-curves cannot intersect.[11]

SketchIT does however allow infinite boundaries to intersect diagonal qcs-curves. Section 4.6.3 describes how SketchIT computes the location of these intersections.

4.6.1 The Intersection Between Two Diagonal Qcs-Curves

Consider the intersection of two diagonal qcs-curves, Q_1 and Q_2 shown in Figure 4-22. Because the curves can have any shape as long as they are monotonic, the point of intersection can lie (almost) anywhere inside the intersection of their bounding boxes. The figure shows two possible locations of the intersection point (i.e., the point (M′, N′)) corresponding to two different shapes of qcs-curve Q_2. For one shape of the curve, the intersection is to the left of landmark D (the end point of curve Q_3), for the other it is to the right. Hence, the location of the intersection is ambiguous, even in qc-space.

If two finite qcs-curves intersect, one end of each qcs-curve will be useless because it lies in the blocked space of the other curve (i.e., the trajectory cannot reach that part of the curve). In the example in Figure 4-22, the portions of the curves below

[9] In its current implementation our simulator does not branch to consider all possible orderings, and hence it could fail to compute all of the possible behaviors of a device. While the current implementation is adequate for the examples we tried, for it to be useful for a wider range of problems, it must be extended to enumerate all possible orderings.

[10] i.e., the program does not produce candidate qc-spaces with intersecting diagonal qcs-curves.

[11] SketchIT actually ensures that all finite qcs-curves intersect each other only at their end points.

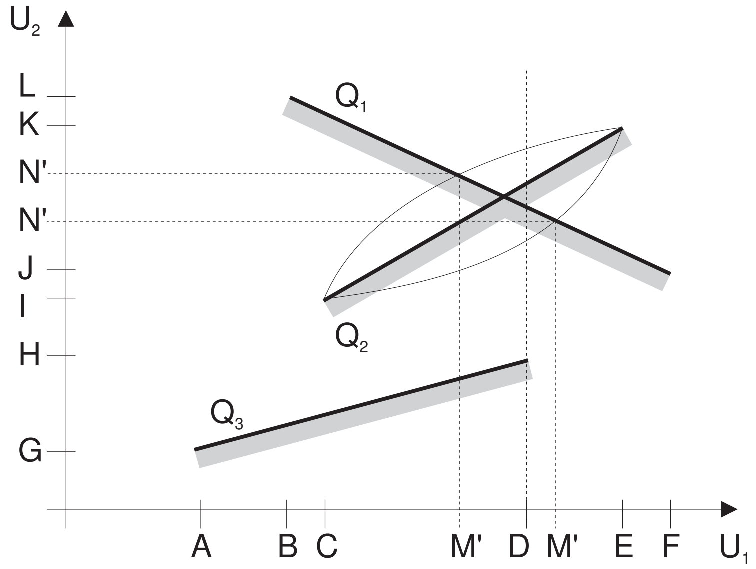


Figure 4-22: Depending on the particular shape of qcs-curve Q_2, the intersection between Q_1 and Q_2 may be either to the left or the right of landmark D.

the intersection point are the useless portions. If we discard these portions, there will be no change in the behavior of the device because the device configuration can never reach these portions. However the ambiguity will be eliminated: the qcs-curves will intersect at their end points (Figure 4-23) and the locations of the end points of qcs-curves are always well defined by a set of ordered landmarks.

Hence, SKETCHIT requires that qcs-curves are trimmed so that they intersect only at their end points. One way to do this is to trim the numerical cs-curves before abstracting the numerical c-space to qc-space.[12] Although the location of the intersection is ambiguous in qc-space, its location in (numerical) c-space is not. Thus, there is no difficulty in trimming the curves in c-space.

If the bounding box of a diagonal qcs-curve intersects the bounding box of another finite qcs-curve, but the qcs-curves themselves are not intended to intersect (i.e., they do not intersect in numerical c-space), SKETCHIT must enforce special constraints to ensure that the curves do not actually intersect. Section 4.6.3 describes how the program does this.

[12]In the examples we tried, none of the qcs-curve required trimming, and hence we did not implement a subroutine to trim numerical cs-curves. However, this is a straightforward extension.

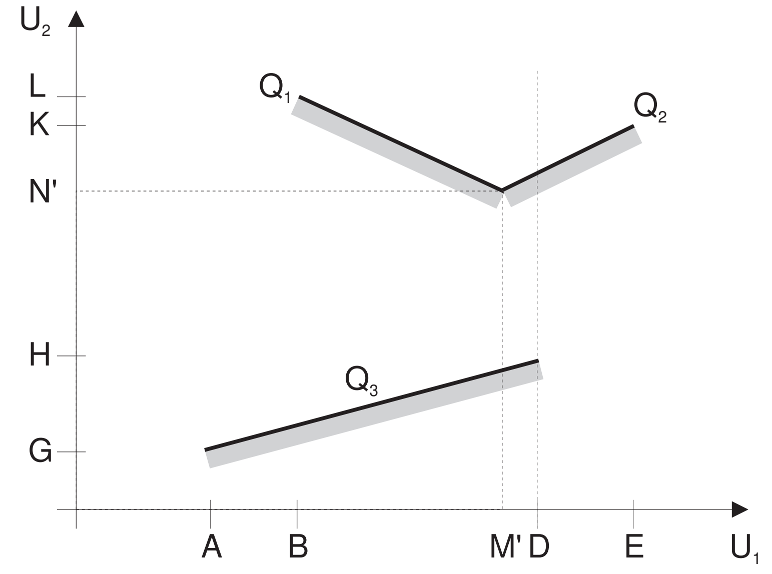


Figure 4-23: When the useless ends of Q_1 and Q_2 are removed, the curves intersect at an end point.

4.6.2 The Intersection Between a Diagonal Qcs-Curves and an Infinite Boundary

The point at which a diagonal qcs-curve intersects an infinite boundary may also be ambiguous. Figure 4-24 shows an example in which qcs-curve Q is intersected by infinite boundary IB. We define a new landmark K′, giving the intersection point (K,K′). Because Q can have any shape as long as it is monotonic, K′ can lie (almost) anywhere between landmarks B and D (the end points of Q). However, landmark C (the end point of another qcs-curve) is also between B and D, and thus K′ can lie either to the left or the right of C.[13]

We call the point of intersection between an infinite boundary and a finite qcs-curve an i-point. Previously, we used the term i-point to refer to the intersection between any two qcs-curves, but because SKETCHIT does not allow finite qcs-curves to intersect, we now use this more restrictive definition.[14]

As noted in Section 4.5.1, SKETCHIT uses the locations of i-points when reasoning about diagonal trajectories along a qcs-curves. The program also uses the locations of i-points to construct "chains" when reasoning about a diagonal trajectory through

[13]It is also possible for K′ and C to be the same landmark, but we do not consider this case.

[14]SKETCHIT does additionally allow infinite boundaries two intersect other infinite boundaries and horizontal and vertical, finite qcs-curves. However, the locations of these intersections are not ambiguous. We reserve the term i-point for the allowed intersections that are subject to ambiguity.

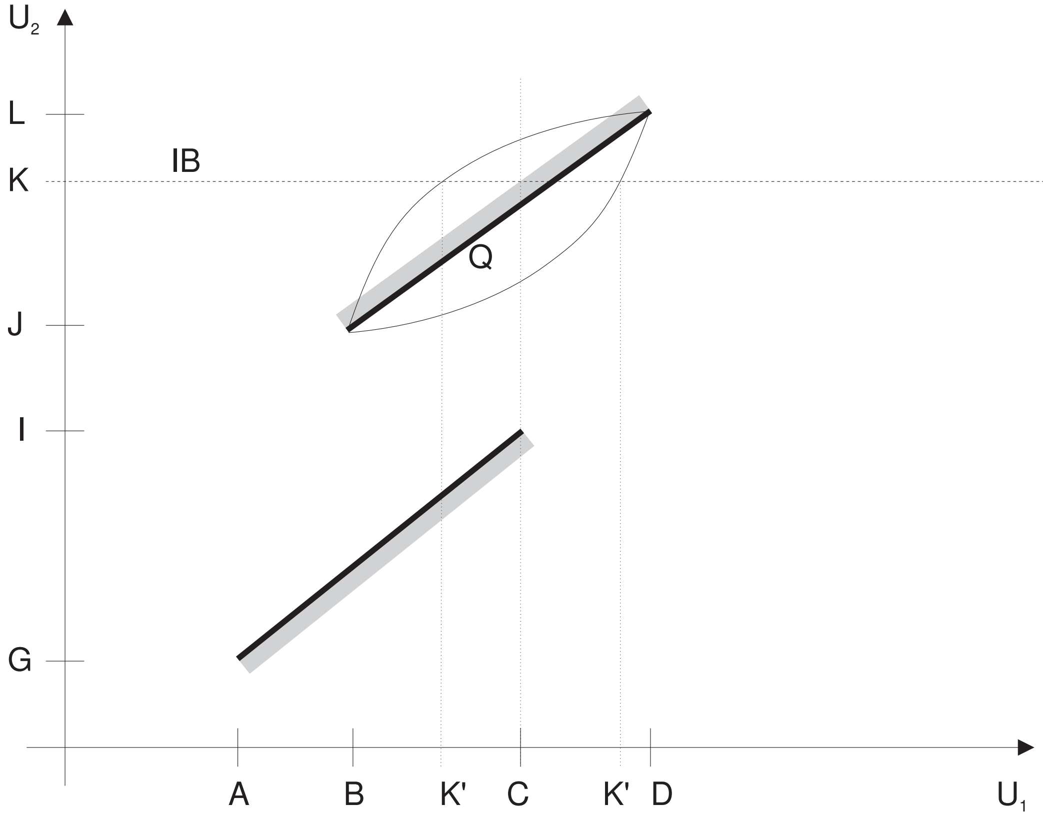


Figure 4-24: Qcs-curve Q can have any shape as long as it is monotonic. Here we show three possible shapes: bowing upward, straight, and bowing downward. The location (K′) of the intersection between qcs-curve Q and infinite boundary IB depends on the specific shape of Q.

the qcs-plane. Hence, ambiguities about the locations of i-points arise when reasoning about these two kinds of trajectories. When these ambiguities arise, the simulator could resolve the ambiguities by enumerating the possible locations of the i-points and branching to consider each choice. However, the bookkeeping is much easier if the generator enumerates all of the choices up front and generates a candidate qc-space for each choice. Then each candidate qc-space passed to the simulator includes unambiguous locations of the i-points, and hence the simulator encounters no ambiguity. This is the approach that SKETCHIT uses.[15]

We call the landmarks that define the end points of qcs-curves and the locations of the infinite boundaries the primary landmarks. The new landmark that we create for each i-point is called a secondary landmark. For the purposes of the simulator, the i-points must be completely ordered with respect to the primary landmarks. Also, all of the i-points for a single qcs-curve must be ordered with respect to each other. However, they need not be ordered with respect to the i-points of other qcs-curves. In Figure 4-25, for example, landmark H′ need not be sorted with respect to K′.

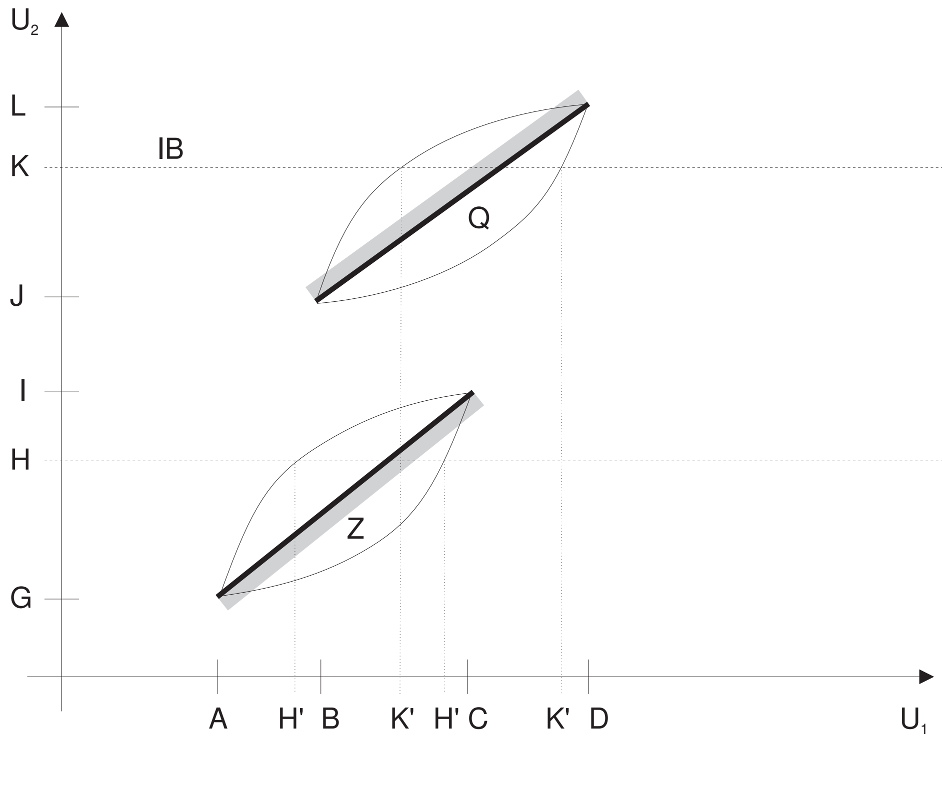


Figure 4-25: H′ is not ordered with respect to K′.

[15]Thus, the generator we initially described in Chapter 3 performs a task that we did not describe there: it enumerates the possible locations of the i-points.

4.6.3 Enumerating Possible I-Point Locations

In this section, we describe the method by which SKETCHIT enumerates the possible locations for the i-points: For each diagonal qcs-curve, SKETCHIT defines an i-point (i.e., SKETCHIT creates a new landmark) for each infinite boundary that intersects that qcs-curve. SKETCHIT also defines an "imaginary" i-point for each primary landmark spanned by the curve: For each primary landmark that lies in the span of the curve along the x-axis, SKETCHIT creates an i-point by intersecting the qcs-curve with a vertical line through the landmark. Similarly, for each primary landmark that lies in the span of the curve along the y-axis, SKETCHIT creates an i-point by intersecting the qcs-curve with a horizontal line through the landmark.

Every i-point (including the regular, non-imaginary kind) has one primary landmark and one new (secondary) landmark. Because the primary landmarks are totally ordered, and because each i-point has a primary landmark as either its x-coordinate or its y-coordinate, the i-points for a qcs-curve are partially ordered: Those i-points whose x-coordinate is a primary landmark form one ordered set, those i-points whose y-coordinate is a primary landmark form another. To enumerate all possible total orderings of the i-points, SKETCHIT simply enumerates all possible ways of interleaving the elements of these two sets (while, of course, maintaining the two partial orderings).

SKETCHIT creates the imaginary i-points for two reasons. First, this allows the program to easily enumerate the possible locations of the regular i-points simply by interleaving two partially ordered sets. Second, SKETCHIT uses the imaginary i-points to ensure that finite qcs-curves with intersecting bounding boxes do not intersect except possibly at their end points. The program does this by placing constraints on the relative locations of the end points and imaginary i-points of the two curves. Consider, for example, the two curves in Figure 4-26. Here the appropriate constraints are between end point (A, B) of curve Q_1 and the imaginary i-points of the other curve, Q_2. Using the landmark names given in the figure, the constraints are: $A' < B$ and $B' > A$.

The generator is the part of the program that actually enforces the non-intersection constraints (i.e., the kinds of constraints used in the example of Figure 4-26). It does this by simply rejecting candidate qc-spaces that do not satisfy the constraints.[16]

4.7 Periodic Boundary Conditions

So far we have considered only pairwise qc-spaces[17] that are planar. In this section we consider cylindrical ones. A pairwise qc-space is a cylinder if one of the two

[16] Thus, the generator we initially described in Chapter 3 performs a task that we did not describe there: it filters out and rejects qc-spaces that do not satisfy the non-intersection constraints.

[17] By pairwise qc-space we mean the qc-space for a pair of fixed-axis parts.

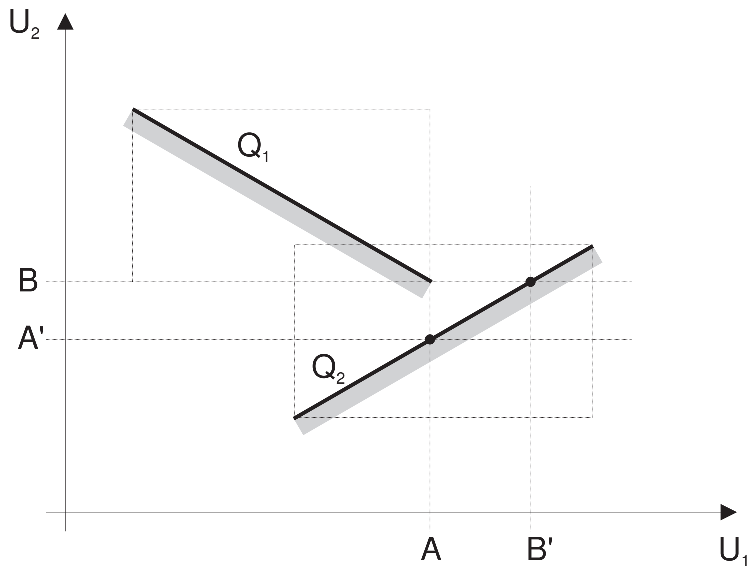


Figure 4-26: Two non-intersecting, finite qcs-curves with intersecting bounding boxes.

bodies rotates more than a full revolution and the other either rotates less than a full revolution or translates.[18] Figure 2-3 shows an example.

SketchIT handles a cylindrical qc-space by flattening it out into a plane and adding boundaries at 0 and 2π. When the trajectory through the plane reaches one of the boundaries, the simulator simply moves it to the other boundary (i.e.,when the angle reaches 0 the simulator changes the angle to 2π, or vice versa).

The difficulty is how to construct boundaries in qc-space that are located precisely at 0 and 2π; qc-space is, after all, a qualitative representation. The remedy is to construct the boundaries in the original c-space before abstracting to qc-space. Because the original c-space is numerical, the boundaries have precise locations there.

The yoke and rotor device from Section 1.1.3 provides us with an example of a cylindrical qc-space. Figure 4-27 shows the flattened out version of the qc-space SketchIT obtains by unrolling the cylinder and adding boundaries. Because curves A1 and B2 cross the boundaries, one piece of each these curves is just above the 0 boundary and another piece is just below the 2π boundary.

[18]If both bodies rotate through full revolutions the qc-space is a torus. SketchIT currently does not handle toroidal qc-spaces. However, the techniques required to handle a toroidal qc-space are a direct extension of those currently used to handle cylindrical one.

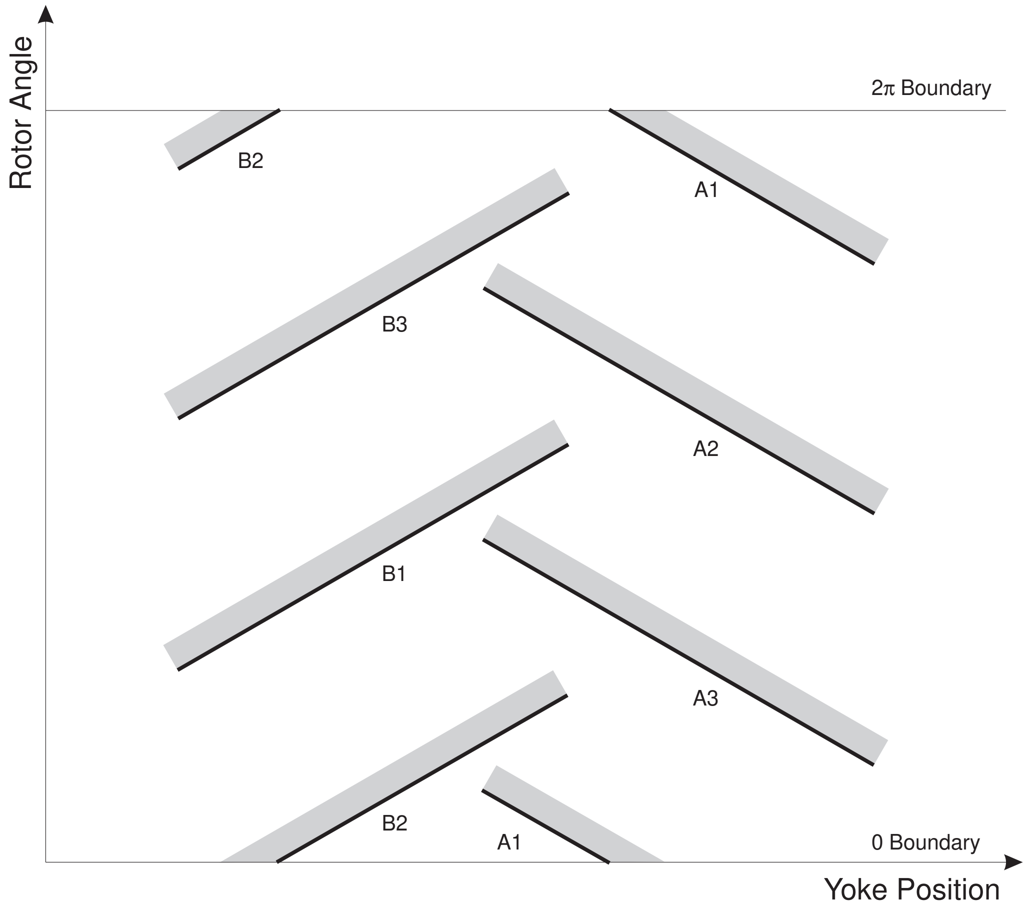


Figure 4-27: The qc-space for the yoke rotor flattened to produce a plane. The labels on the qcs-curves indicate the names of the interacting faces. For example, curve A1 is the interaction between face A on the yoke and face 1 on the rotor (see Figure 1-12). Because curves A1 and B2 cross the boundaries, one piece of each of these curves is just above the 0 boundary and another piece is just below the 2π boundary. Although the qcs-curves are drawn as straight lines, they can have any shape as long as they are monotonic.

4.8 Setting Up the Next Step

Section 4.5 described how the simulator determines when one or more simultaneous events changes the nature of the forces, requiring the simulator to recompute the motion of the parts of a device. This section describes how the simulator continues the simulation after these events occur.

To start the next step of simulation, the simulator must determine which pairs of engagement faces remain engaged after the terminal events of the previous step and which do not. The simulator must also determine which inputs are active during the next step. This section describes how the simulator performs the first of these two tasks. Section 4.9.1 discusses the second task.

4.8.1 Keeping Track of Engagements

At the start of the very first step of simulation, the simulator knows which faces are engaged and which are not (this is part of the initial conditions). To determine which faces are engaged at the beginning of the next time step, the simulator simply keeps track of changes in the engagements, i.e., it keeps track of which engaged faces become disengaged and which disengaged faces become engaged during the current step.

By computing the events that terminate the current step, the simulator already does most of the bookkeeping necessary to track the changes in engagements. For example, every new engagement is marked by an explicit event. Similarly, every kind of disengagement—except those caused by a process we call "pulling apart"—is marked by an explicit event. Hence, the only remaining task is to keep track of faces that disengage by "pulling apart."

By definition, "pulling apart" occurs when a pair of faces is engaged and the interface motion is retreating from both viewpoints, or stationary from one viewpoint and retreating from the other.[19] In this case, the faces instantly disengage, but there is no change in the nature of the forces, and hence the simulator does not label this as an event and terminate the current step. To keep track of which faces disengage by pulling apart, the simulator simply examines the interface motion of each pair of faces at the start of the current step.

By continuing to apply this continuity principle, the simulator keeps track of the engagements from one step to the next.

[19]There are two ways a pair of faces can disengage: one is when the trajectory in qc-space is (qualitatively) perpendicular to the qcs-curve for the faces, the other is when the trajectory in qc-space reaches the end of the qcs-curve. The former case is what we call pulling apart, the latter case is an event that the simulator detects.

4.8.2 Trial Steps

Sometimes when a pair of faces disengages by sliding past each other, either a spring relaxes or an input turns off so that the net force causes the faces to instantly re-engage. In this situation, the simulator will be in error if it assumes that the faces are disengaged during the next step. To avoid this error, the simulator analyses the forces that exist when the faces are just on the verge of separating to determine if the forces actually cause the faces to separate.

The simulator does this by computing a trial step of simulation. The simulator sets up the next step of simulation as if the faces remain engaged. The simulator then computes the forces on the bodies to determine if the faces really do disengage. If they do, the simulator throws away this step of simulation and computes a new one assuming the faces do disengage. Otherwise, the simulator keeps this step and continues on.

4.9 Controlling The Simulation

Section 4.4 described how the simulator starts a step of simulation from a known set of initial conditions. Section 4.5 described how the simulator determines when an event changes the nature of the forces and hence terminates the current step of simulation. Section 4.8 described how the simulator determines which pairs of engagement faces are engaged in the next step of simulation.

In this section we add the last two details that tie all of these pieces together, allowing us to compute a complete simulation: we describe how the simulator selects the appropriate external inputs to drive the simulation and how it selects the initial conditions to start it in the first place.

4.9.1 External Inputs

In a conventional (numerical) simulation, external inputs are commonly specified as a function of time. In our qualitative world, however, this is not possible because time is not a well defined quantity. Instead, we specify the external inputs as a function of the device state using a state transition diagram. Each node in the diagram is a list of the pairs of faces that are engaged and the springs that are relaxed. (The pairs of faces not listed at a node are by default disengaged, the springs not listed are by default not relaxed.) The arcs are the external inputs that drive the device. Figure 1-3b, for instance, describes how the circuit breaker should behave in the face of heating and cooling the hook and pressing the reset pushrod.

Because the external inputs are specified as a function of state, the simulator can compute the complete simulation in a piecewise fashion by simulating each arc in the diagram independently. For the circuit breaker, for example, the program simulates the "hook heats," "hook cools", and "reset" arcs as three separate simulations.

To simulate an arc, the program first computes initial positions for each body from the state specified by the node at the tail of the arc. The program then applies the external inputs of the arc and simulates the behavior until the motion either reaches steady state or stops. Later, the tester (Chapter 5) checks to see if the states at the end of the simulation of one arc match the initial conditions for the next arc, and so on. If so, the candidate qc-space is capable of producing the desired behavior and is accepted.

4.9.2 Computing Initial Positions

To start the simulation of an arc, the program must first compute initial positions for each body. The program derives the initial positions from the list of engaged faces and relaxed springs specified by the node at the head of the arc. The program does this using the same kind of constraint propagation it uses in Section 4.5.2 to compute the next events.

For each pair of engaged faces, the program constructs algebraic constraints that ensure the initial positions of the corresponding pair of bodies are inside the bounding box of the qcs-curve for the pair of faces. For each relaxed spring, the program constructs a constraint that ensures the body attached to the movable end of spring is located at the spring's neutral position.

By solving these constraints, the program obtains a range of values for the initial position of each body. The range for a particular body may be large enough that its initial position is not ordered with respect to the landmarks of the end points and i-points of the qcs-curves. If this happens, the simulator must branch in order to consider all possible orderings (see Section 4.5.3), and hence all possible initial positions consistent with the specified initial state.[20]

If the constraints on the initial positions are inconsistent (i.e., cannot be satisfied), the candidate qc-space is incapable of producing the desired behavior and is rejected.

4.9.3 Will The Simulation Terminate?

One concern in running a simulation is how long the simulation should continue. For conventional (numerical) simulations, one usually specifies a time interval. In our qualitative world, however, it is not possible to specify a time interval because time is not well defined in our qualitative representation. Instead, we simulate until the motion either reaches steady state or terminates. In this section, we demonstrate that the simulation will always reach one of these two conditions.

There are three ways in which the motion of a device can continue indefinitely in response to a single set of inputs (i.e., the set of inputs applied at a node in the

[20]Currently, the program enumerates the possible orderings and prompts the user to select one of the possibilities. This is convenient for debugging purposes. It is a simple extension to have the program branch to consider all of the enumerated choices rather than prompting the user.

state transition diagram): the device could exhibit cyclic behavior, the device could asymptotically approach some rest state, or the motion of the parts could reach a steady state.

First, we consider cyclic behavior. There are three kinds of cyclic behavior: free oscillations, such as the oscillation of a mass-spring oscillator; forced oscillations caused by an oscillating input; and forced oscillations caused by a cycle of engagements, such as the oscillation of a ratchet pawl that occurs when the ratchet wheel is continuously rotated.

Because SKETCHIT's world is inertia-free, there can be no free oscillations. In SKETCHIT's world external inputs do not oscillate (they either apply a constant direction force, apply a constant direction motion, or select a temperature). Hence, there can be no oscillations caused by oscillating inputs. The application of a *sequence* of external inputs with alternating magnitudes can produce forced oscillations. (For example, the *sequence* of inputs applied to the circuit breaker causes cyclic behavior.) However, the single set of inputs applied at a particular node in the state transition diagram cannot cause forced oscillations.

A device constructed such that its qc-space has periodic boundary conditions can produce cycles of engagements that lead to forced oscillations. For example, the yoke and rotor device's qc-space has a periodic boundary condition, and as a result continuous rotation of the yoke causes a continuous cycle of engagements. Once SKETCHIT detects this kind of cycle, it can terminate the simulation because no new behavior will occur. SKETCHIT can detect this kind of cycle by checking if the current state of the device is the same as a previous state. For two states to be the same, the same engagement pairs must be engaged, the same springs must be relaxed, and the same inputs must be active.[21] For the yoke and rotor, this definition of same state correctly identifies the behavior of the device as cyclic. An open question is if there are devices for which this definition produces false positives.

To summarize, in SKETCHIT's world, the only way a device can produce cyclic behavior in response to a single set of inputs is if the device's qc-space has periodic boundary conditions. Because SKETCHIT can detect and terminate this kind of cycle, the simulation of an arc will never continue indefinitely as a result of cyclic behavior.

Second, we consider the asymptotic approach to a rest state. A step of simulation ends because an event occurs, not because a fixed amount of time has elapsed. The simulator's qualitative representation abstracts away the notion of time duration. Hence, the time required to asymptotically approach a rest state is abstracted away and the simulation terminates after a finite number of steps. Thus, the simulation of a device that asymptotically approaches a rest state will not continue indefinitely.

If there are no cycles, the only way that motion can reach a steady state is if the motion is unbounded, that is, the trajectory in qc-space continues forever without

[21] Although our procedure for detecting cycles is a simple extension of the current program, we have not yet implemented it.

reaching a qcs-curve. Because the simulator exhaustively checks the trajectory for collision with each qcs-curve, it can determine when no collisions are possible. When the simulator detects this condition, it terminates the simulation.

Hence, none of the three ways in which motion continues indefinitely can cause the simulation of an arc to continue indefinitely. Of course, if the motion terminates (i.e., does not continue indefinitely), so does the simulation. Thus, the simulation of an arc will always terminate.

4.10 Examples: Envisionments

The last section described how the simulator computes a complete simulation by simulating each arc of the state transition diagram individually. This section shows some examples of the output the simulator produces as it simulates an arc. One purpose of these examples is to illustrate the amount of branching that occurs during a simulation. The examples are taken from the circuit breaker: the desired behavior is specified in Figure 1-3b and the particular candidate qc-space is shown in Figure 3-1.

The first example is the simulation of the "hook heats" arc. This behavior, called the trip action, occurs when there is a current overload. Figure 4-28 shows SKETCHIT's display of the simulation results. Because the simulator computes an envisionment, the simulation has a tree structure (node "time-3" is the root). Each node in the tree corresponds to a step of simulation. By clicking a node with one mouse button the user obtains the initial conditions for that step; by clicking with another mouse button, the user obtains a description of the events that caused that step to terminate. In the discussion below, we summarize the results the user would obtain by probing the nodes with the mouse.

At time-3 the hook restrains the lever (the lever-stop in engaged), the pushrod is against its stop (the pushrod-stop is engaged), and the hook heats. Note that we model the heating of the hook as an instantaneous change in the hook's neutral position: Before time-3, the hook's neutral position is the "hook=cold" curve in Figure 3-1, just after time-3, the hook's neutral position is the "hook=hot" curve. Although the hook's neutral position changes instantly, the hook's physical position changes gradually in response to the unbalanced force resulting from the change in neutral position.

As the hook begins to relax, it releases the lever (time-4-zinc-1). As the hook continues to relax, the lever heads toward a collision with the pushrod. There are three possible next events: the lever collides with the pushrod (time-7), the hook reaches its neutral position (time-6), or the lever collides with the pushrod at the same instant the hook reaches its neutral position (time-5). After the collision at time-7, there is only one possible next event: the hook reaches its neutral position (time-8). After the hook relaxes at time-6, there is only one possible next event: the lever collides with the pushrod (time-9).

As a second example, Figure 4-29 shows the simulation results for the reset action (i.e, the "reset" arc). The reset action turns a tripped circuit breaker back on after it has cooled.

The simulation begins at time-3, when the hook is relaxed at the "hook=cold" neutral position, the pushrod is against its stop (the pushrod-stop is engaged), the lever is against the pushrod (the push-pair is engaged), and the pushrod is being pressed. The pushrod pushes the lever which eventually strikes the hook at time-4 (the cam-follower is engaged). As the pushrod continues to push the lever, the lever pushes the hook out of the way until time-5-zinc-1 when the lever disengages the hook (the cam-follower disengages). At the start of step time-5 the simulator performs a single trial simulation step to determine if the lever and hook really do disengage (see Section 4.8.2).[22]

After the lever disengages the hook, the pushrod continues to push the lever, and the hook begins to relax. There are four events that will eventually happen: the pushrod will reach the end of its stroke, the lever will strike the hook (lever-stop engages), the pushrod will reach its stop (pushrod-stop engages), and the hook will reach its neutral position. The fist three of these events always occurs in the order listed. The fourth event can occur before or after any of the other three events. Each of the branches that emanate form time-5-zinc-1 corresponds to a different choice for when the fourth event occurs.

For example, in the branch that terminates at time-10, the sequence of events is: the pushrod reaches the end of its stroke at the same instant the hook relaxes (time-8), the lever hits the hook (the lever-stop engages, time-9), and the pushrod hits its stop (the pushrod-stop engages, time-10). In the branch that terminates at time-13 the sequence of events is: the hook relaxes (time-7), the pushrod reaches the end of its stroke (time-11), the lever hits the hook (the lever-stop engages, time-12), and the pushrod hits its stop (the pushrod-stop engages, time-13).

[22] The "zinc-1" in "time-5-zinc-1" indicates that a trial step occured.

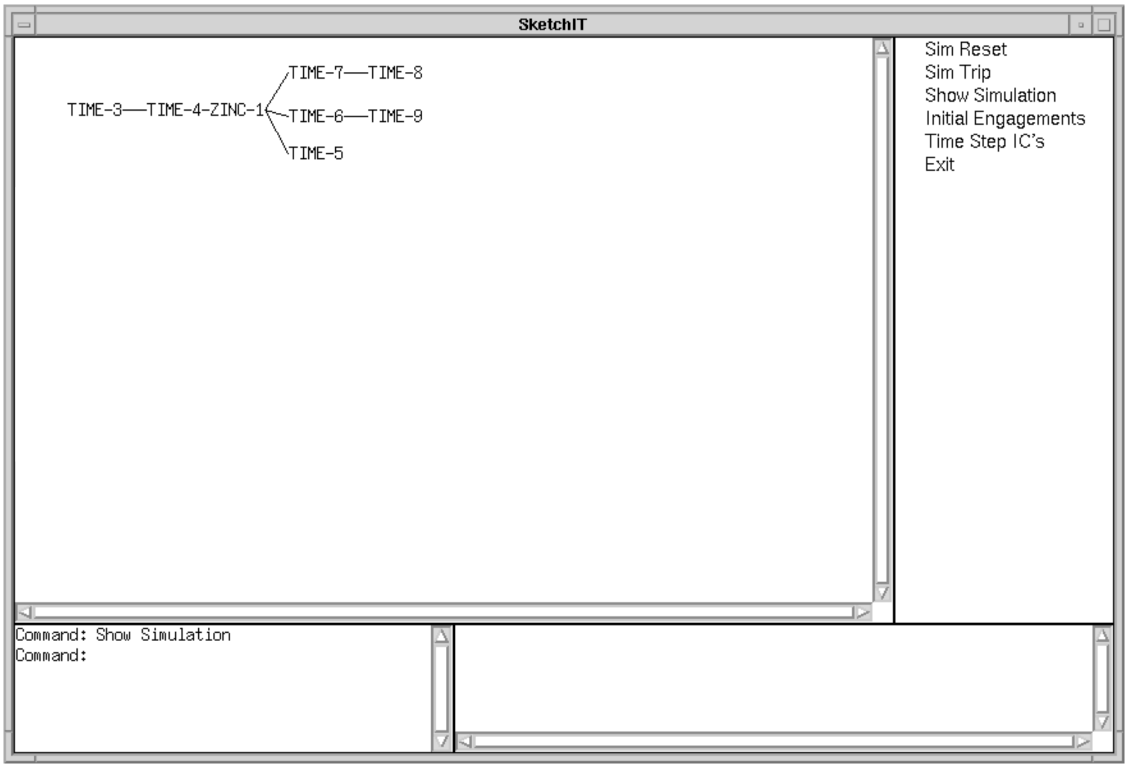


Figure 4-28: An envisionment of the circuit breaker's motion during the trip action.

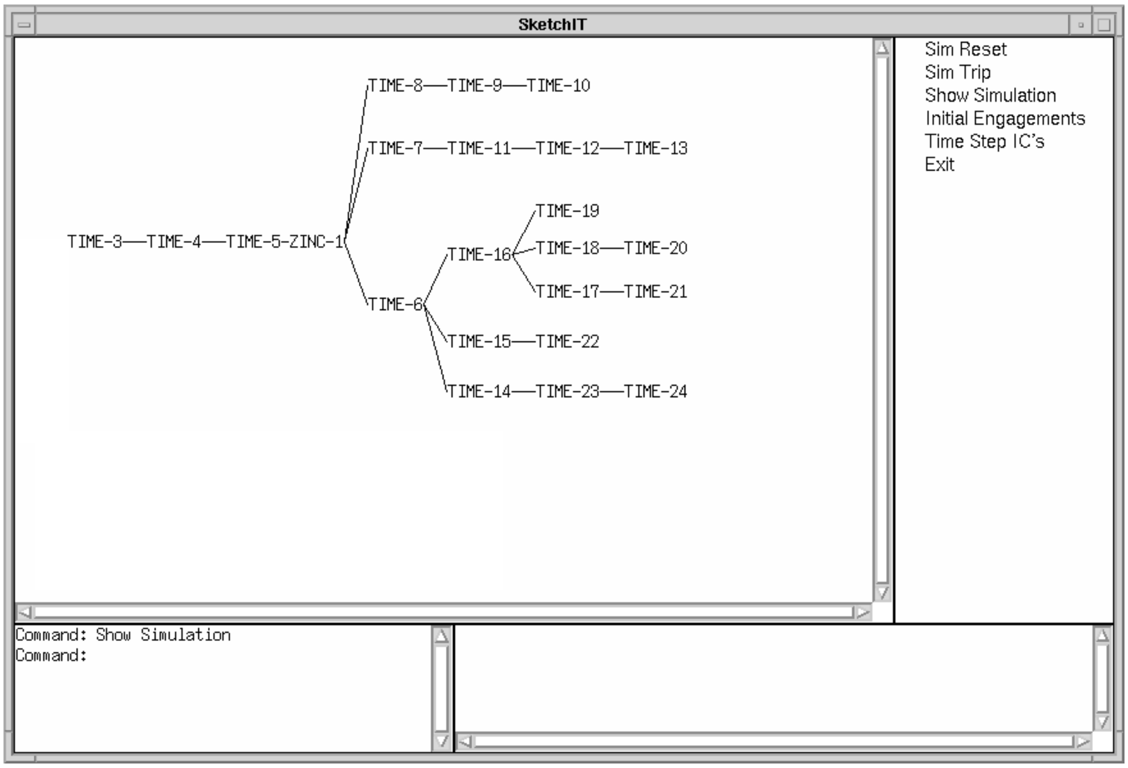
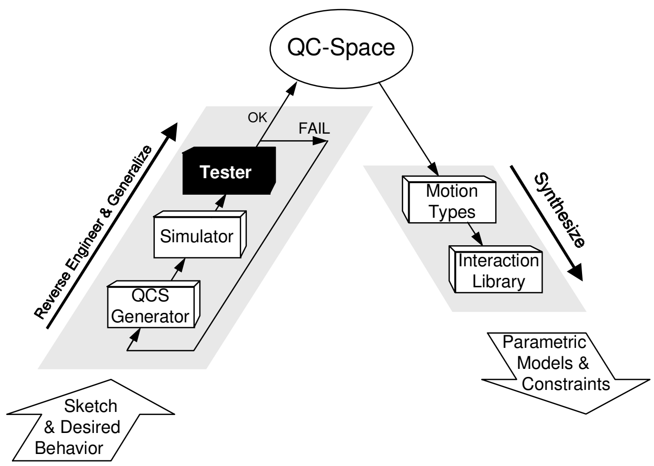


Figure 4-29: An envisionment of the circuit breaker's motion during reset.

Chapter 5

The Tester



5.1 Current Implementation

During the reverse engineering and generalization process, SketchIT's job is to determine what behavior each piece of the sketch ought to provide. It does this by searching for a qc-space description of the sketch that provides the desired overall behavior.

Chapter 3 described the qc-space generator, which produces a sequence of candidate qc-spaces for the sketch. Each of these candidates describes one set of possible behaviors for each part of the sketch.

A qc-space is an implicit description of behavior: qc-space describes the kinematics of the device, that is, it describes all possible positions the device's parts can occupy.

Chapter 4 described the simulator SketchIT uses to make explicit the behavior of a candidate qc-space. The simulator computes all possible motions the device's parts can exhibit in response to the external inputs specified by the state transition diagram. The simulator computes the simulation in a piecewise fashion: it simulates each arc of the state transition diagram individually.

This chapter describes the tester that SketchIT uses to compare the simulated behavior of a candidate qc-space to the desired overall behavior. Because the simulator simulates each arc individually, the tester's job is to determine if the simulation of one arc reaches the initial conditions of the next arc, and so on. If so, the candidate qc-space is capable of producing the desired behavior and is accepted.

Because the simulator computes all possible behaviors, the simulation of any particular arc may branch. Hence the simulation of a single arc may reach several possible terminal states. To determine if the simulation of one arc reaches the initial conditions of the next, the tester matches each of the terminal states of one arc against the specified initial state of the next (i.e., to next node in the state transition diagram). Two states match if the same pairs of faces are engaged and the same springs are relaxed in the both states. The tester requires that all of the terminal states of one arc match the initial state of the next. Otherwise it rejects the candidate qc-space because in this case the device is capable of producing at least one behavior that does not match the desired behavior.

5.2 A More Powerful Tester

The tester described above is the simplest implementation adequate for the examples we explored. This section describes how we would construct a more powerful tester. One of the features of this more powerful tester is that it will support a richer behavior specification language. This section begins by describing this richer specification language.

5.2.1 Extending the Behavior Specification Language

We specify the desired behavior of a device with a state transition diagram. In the current implementation, each node in the diagram is a list of the pairs of faces that are engaged and the springs that are relaxed. The pairs of faces not listed at a node are by default disengaged, the springs not listed are by default not relaxed. The arcs are the external inputs that drive the device.

The more powerful tester will allow additional choices for specifying the states of engagement faces and springs at a node. Engagement faces will be either engaged, not engaged, or unspecified. Similarly, springs will be either stretched, relaxed, compressed, or unspecified.

The designer will be able to annotate arcs to specify a desired state for springs and engagement faces during a transition. For example, the designer may specify that a particular pair of faces should remain not engaged during a particular transition.

Finally, the designer will be able to specify partially ordered states, that is, a set of states that can occur in any order.

5.2.2 Extending the Tester

The current tester assumes that, after one set of inputs is applied, the device reaches steady state before the next set is applied. This assumption is appropriate for the examples we explored, but is in general too restrictive.

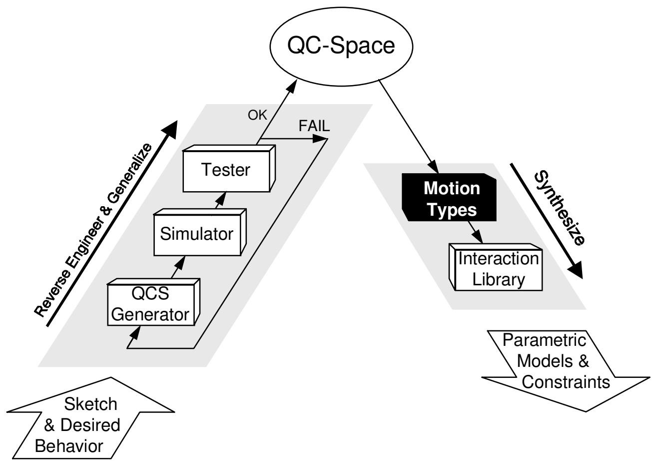
A better approach is for the tester to monitor each branch of the simulation to determine when the state in that branch reaches the next node in the state transition diagram. When this happens, the simulator will instruct the simulator to apply the next set of inputs to that branch.

If all of the branches pass through the desired set of states (i.e., the nodes in the state transition diagram) in the desired order, the qc-space will surely provide the desired behavior and will be accepted.

If some of the branches pass through the desired states in the desired order, while others do not, the qc-space is capable of providing the desired behavior only for specific choices of masses, springs, and actuators. Numerical simulation can be used to verify the design once these choices have been made. Hence, we can use qualitative simulation for conceptual design and progress to more precise numerical techniques (which are more computationally expensive) as more of the design details are selected. An alternative approach would be to construct additional constraints on the masses, springs, and actuators that ensure that only the desirable branches of the simulation are possible. These constraints would be added to the BEP-Models computed from the qc-space.

Chapter 6

The Motion Type Selector



6.1 Introduction

This chapter begins discussion of the synthesis process, whose task is to generate geometry that implements the working qc-spaces computed by the reverse engineering process. If a qc-space produces the desired behavior, then any geometry that implements that qc-space will produce the desired behavior.

The motion type selector, which we discuss in this chapter, starts the synthesis process by selecting a motion type, either rotation or translation, for each part in the device. In the next chapter we describe the interaction library which selects geometry for each engagement pair.

6.2 Abstracting Away Motion Type

SketchIT is free to select a new motion type for each part because qc-space abstracts away this property. More precisely, qc-space abstracts away the motion type of parts that translate and parts that rotate less than a full revolution. (We consider parts that rotate through full revolutions below.)

As evidence that a rotating part that turns less than a revolution can be replaced by a translating part (and vice versa) with no resulting change in behavior, consider that when the simulator computes motion, it makes no distinctions between these two kinds of parts. The simulator need not make distinctions because it computes motion using generalized forces which abstract away a primary difference between the two kinds of parts. Recall that for translating parts the generalized forces are the usual kind of forces, because these are what effect the motion of translating parts. For rotating parts, the generalized forces are torques and moments of the usual kind of forces, because these are what effect the motion of rotating parts. For both kinds of parts, the sum of the generalized forces determines the motion. Thus, as long as the simulator uses generalized forces, the rules for computing motion (i.e, behavior) do not depend on whether the part rotates or translates.

Qc-space cannot abstract away the motion type of parts that rotate more than a full revolution. As we discussed in Section 2.2.2, if a part rotates through full revolutions its coordinate axis in qc-space will wrap around to produce a cylindrical topology. However, translating parts never produce a cylindrical topology in qc-space. Thus, there is a qualitative difference between these two kinds of parts, and consequently one kind cannot be substituted for the other.

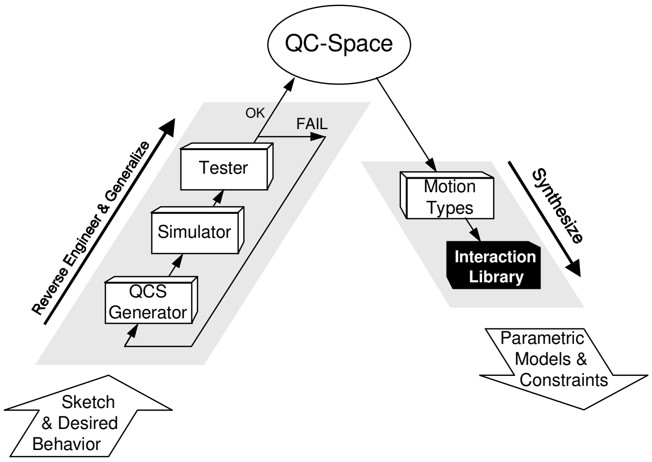
From a more physical perspective, if a part that rotates through full revolutions turns far enough in a single direction, it will return to the position from which it started, without ever having to reverse its direction of motion. The only way that a translating part can return to the place from which it started is if the part reverses its direction of motion. Hence, a translating part cannot replace a rotating part that turns through full revolutions.

In summary, as long a rotating parts rotates less than a full revolution, the use of generalized forces abstracts away all difference between it and a translating part.

By changing translating parts to rotating ones, and vice versa, SKETCHIT can generate a rich assortment of new designs, as we saw illustrated in Figure 1-6. The new designs obtained by changing motion types are completely consistent with the desired behavior, because they produce the sequence of engagements given in the state transition diagram.

Chapter 7

The Interaction Library



7.1 Introduction

The previous chapter described how SketchIT's synthesis process considers all possible motion types, either rotation or translation, for each component. Here we discuss how SketchIT implements a qc-space, i.e., how it translates the qc-space into geometry. A qc-space is characterized by the qualitative slope of each qcs-curve, and the partial orderings of the landmarks. To implement a qc-space, SketchIT must find geometry for each pair of engagement faces, such that the faces produce qcs-curves that have the correct slope and satisfy the partial orderings of the landmarks.

SketchIT selects an implementation for a pair of faces from a library of interactions. Each library entry contains a pair of parameterized faces and a set of constraints that ensure the faces implement a monotonic cs-curve with a particular qualitative slope.[1] To facilitate achieving the partial orderings of the landmarks,

[1] A qcs-curve represents a family of monotonic cs-curves. For a specific choice of parameter values, a library entry produces a single cs-curve. This cs-curve is one member of the family of cs-curves

each library entry contains algebraic expressions for the end points of the cs-curve produced by that pair of faces. By selecting a library entry for each qcs-curve, and assembling the parametric geometry and constraints for each selection, SKETCHIT constructs a BEP-Model, a parametric model augmented with constraints that ensure the geometry provides the desired behavior.

Figure 7-1 shows all possible types of finite qcs-curves. Because there are eight types, and the coordinates q_1 and q_2 can be either rotation or translation, we need a library capable of implementing 32 different types of interactions.

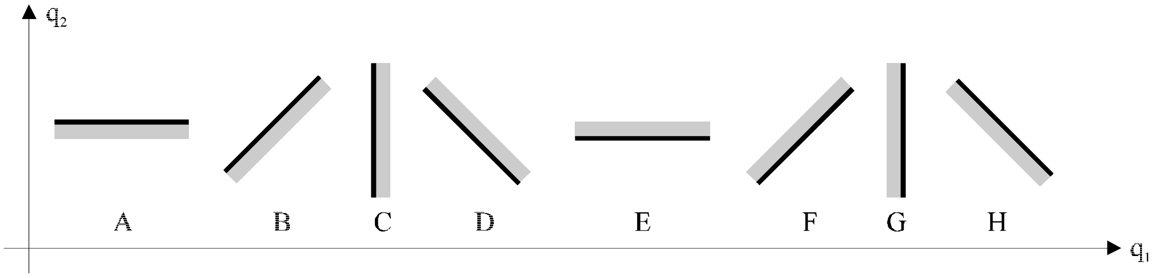


Figure 7-1: For drawing convenience, qcs-curves are shown as straight line segments; they can have any shape as long as they are monotonic.

The interaction between a rotating or translating face and a fixed face produces either a horizontal or vertical, infinite qcs-curve. These infinite qcs-curves are infinite versions of curves A, C, E, and G inFigure 7-1.[2] Because there are four types of infinite qcs-curves, and one of the interacting bodies can either rotate or translate while the other body is fixed, our library must include implementations for 8 additional types of interactions. Hence, the complete library must include a total of 40 interactions: 32 for finite qcs-curves and 8 for infinite ones.

However, by appropriate use of coordinate transformations, we can implement the 40 different kinds of interactions with just 10 different kinds of implementations. Table 7.1 show the 10 distinct cases that remain after using coordinate transformations.

In the next section, we describe the library in more detail. Later, we describe how SKETCHIT assembles the library entries into a BEP-Model.

7.2 The library

In this section, we describe the interaction library, providing a derivation of several library entries. The complete library is located in Appendix A.

defined by the corresponding qcs-curve.

[2]The neutral positions of springs and the motion limits of actuators also produce infinite qcs-curves, but they do not describe interacting faces, and hence require no geometric implementation.

curve type	motion types	cases
diagonal	both translation	1
diagonal	both rotation	2
diagonal	translation & rotation	1
horizontal or vertical	both translation	1
horizontal or vertical	both rotation	1
horizontal or vertical	translation & rotation	2
horizontal or vertical	translation & fixed	1
horizontal or vertical	rotation & fixed	1

Table 7.1: The number of different implementations (cases) required to implement all possible qcs-curves. "Motion type" refers to the motion type the bodies containing the faces. The last two table entries correspond to infinite qcs-curves.

We use as our main example library entries that implement qcs-curve H in Figure 7-1, for the case in which q_1 is rotation and q_2 is translation (i.e., rotation in the negative direction causes translation in the positive direction). We show two different implementations for this curve.

7.2.1 Cam and Offset Follower

As our first implementation for curve H, we use the two flat faces whose parameterization is shown in Figure 7-2. Our goal is to derive a set of constraints on the parameters such the faces implement a monotonic qcs-curve with the same slope as qcs-curve H.

As Figure 7-2 shows, there are 7 parameters that characterize the pair of faces (we do not count θ and x because they measure positions, i.e., they are position parameters not geometric parameters). To obtain constraints ensuring that the faces implement qcs-curve H, we must identify the regions in the 7-dimensional parameter space for which the faces produce a monotonic curve of correct slope. Finding all such regions, and expressing the result as a set of algebraic constraints is a difficult, if not intractable, task. Instead, we look for individual regions for which the constraints are simple. The more regions we can identify, the more ways the program will have to implement a design.

To guide us in this search for the simple regions, we take inspiration from the kinds of geometry that designers commonly use to achieve a particular kind of interaction. For example, a common way to implement curve H using the geometry of Figure 7-2 is if the geometry acts like a cam and offset follower: a design for which the translating face does not pass through the pivot of the rotating face. With this as our inspiration, our goal is to find a set of constraints which describe the class of solutions corresponding to a cam with offset follower.

Our first constraint is that the two faces exist, that is, they have non-zero length.

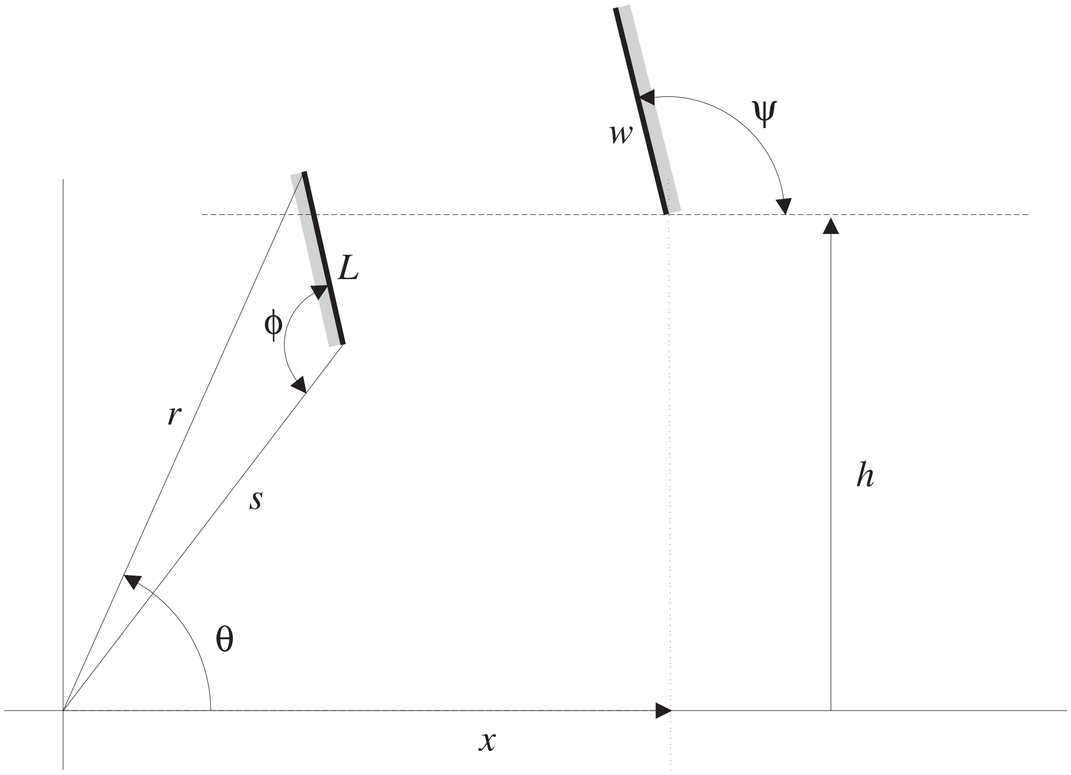


Figure 7-2: The parameterization of a pair of faces used to implement qcs-curve H. The rotating face rotates about the origin, its position measured positive counterclockwise with angle θ. The translating face translates horizontally, its position measured positive to the right with x. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

Defining w as the length of the translating face and L as the length of the rotating face, we trivially obtain:

$$w > 0 \tag{7.1}$$

$$L > 0 \tag{7.2}$$

To ensure that the translating face does not pass through the pivot (a hallmark of cam-and-offset-follower behavior), we define the follower offset, h, subject to the constraint:

$$h > 0 \tag{7.3}$$

We define h such that when one looks down the positive x-axis, the translating face is to the left.

To ensure that the rotor (i.e., the rotating body) can actually engage the slider (i.e., the translating body), we must ensure that the rotor is long enough to reach the slider. To this end we construct the distance from the pivot to each end of the rotating face. The longer of the two distances we label r, the other we label s (because of the constraint expressed by Equation 7.7, one distance is always larger than the other). r and s are related to the length of the rotating face by:

$$r = (s^2 + L^2 - 2sL\cos(\phi))^{1/2} \tag{7.4}$$

For the rotor to be long enough to engage the slider, r must satisfy:

$$r > h \tag{7.5}$$

If s is sufficiently large, when the rotor is vertical and pointing upward ($\theta \approx \pi/2$), the slider face will be able to pass under the rotor face. As a result, there will be two separate ranges of angle for which the rotor and slider will be able to engage, one range to the right of vertical and another to the left of vertical. In this case, the interaction between the rotor and slider will be two disjoint cs-curves, and thus this geometry will not be a valid implementation of curve H. To ensure that this kind of disjointed interaction does not occur, we must constrain length s.[3] An overly conservative constraint, which we use for its simplicity, is:

$$s < h \tag{7.6}$$

A less restrictive, but equally effective constraint is $s < h + w\cos(\psi - \pi/2)$. This constraint ensures that when the rotor is vertical (i.e., $\theta = \pi/2$) the top of the slider face is higher than the bottom of the rotor face. For historical reasons, the program uses the more conservative constraint.

Defining rotation positive counterclockwise, we can refer to the line labeled r in Figure 7-2 as the leading edge of the rotor, and the line labeled s as the trailing edge.

If the leading edge is shorter that the trailing edge, the cs-curve will not have the desired slope.[4] Consider a configuration in which the bottom end of the slider face touches the middle of the rotor face and the rotor turns counterclockwise as shown in Figure 7-3. If the leading edge is shorter than the trailing edge, the radius from the pivot to the contact point will get longer. Because this radius gets longer, the rotor will push the slider to the right. In this case, the cs-curve will have (at least locally) a slope like qcs-curve B in Figure 7-1, rather than like qcs-curve H. Hence, we must constrain the leading edge to be longer than the trailing edge. We express this constraint in terms of the angle ϕ between the trailing edge and the rotor face:

---

[3] We can use the disjointed solution if we know that during the normal operation of the device, the device only be in configurations from one range of engagement.

[4] We assume that $0 \leq \phi \leq \pi$.

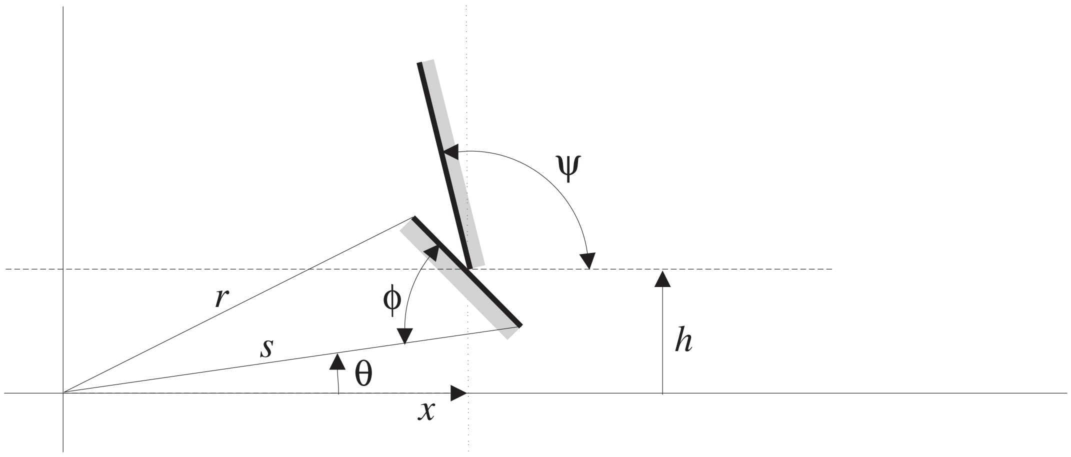


Figure 7-3: The leading edge of the rotor is shorter than the trailing edge. Turning the rotor counterclockwise pushes the slider to the right.

$$\phi > \pi/2 \tag{7.7}$$

Consider the configuration in which the tip of the rotor is touching the bottom of the slider face as shown in Figure 7-4. If the rotor face is horizontal in this configuration (it is not horizontal in the figure), then the pair of faces will act as a stop: the slider face will be able to slide freely along the rotor face, but the rotor will be prevented from rotating clockwise. The corresponding qcs-curve will locally be vertical, not diagonal like qcs-curve H. Hence, we must constrain the rotor face so that it is not horizontal when in this configuration. If we define c as the angle between the rotor face and vertical (see Figure 7-4), the appropriate constraint is:

$$c < \pi/2$$

Our task now is to express c in terms of the parameters of the faces. Figure 7-4 shows the parameters as well as some intermediate variables we use in the derivation. We start by considering triangle e-f-g shown in the figure. Because a and b are two angles of this triangle, and c is the complement of the third angle:

$$c = a + b$$

Using the law of cosines we obtain for a:

$$a = \arccos\left(\frac{L^2+r^2-s^2}{2Lr}\right)$$

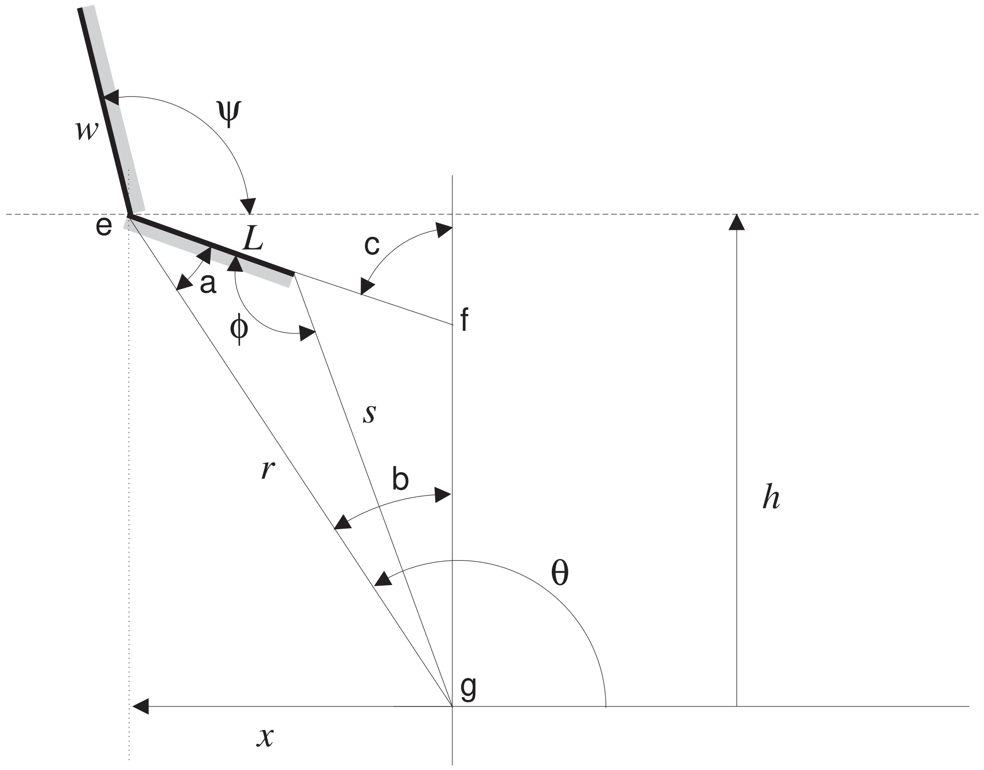


Figure 7-4: When in this configuration, the rotor face must not be horizontal.

By inspection of the figure we obtain for b:

$b = \arccos(h/r)$

Combining the previous four expressions, we obtain the desired constraint:

$$\arccos(\frac{L^2 + r^2 - s^2}{2Lr}) + \arccos(h/r) < \pi/2 \tag{7.8}$$

Our expression for a assumes that $0 \leq \phi \leq \pi$. Equation 7.7 enforces the lower bound, but we require an explicit constraint for the upper bound:

$$\phi \leq \pi \tag{7.9}$$

For the qcs-curve to be monotonic, the tip of the rotor must never move tangent to the slider face (see Figure 2-13 for an example of a non-monotonic interaction). This in turn requires that the angle between the slider face and the leading edge of the rotor must always be greater than $\pi/2$. If we label this angle z, as shown in Figure 7-5, the constraint is:

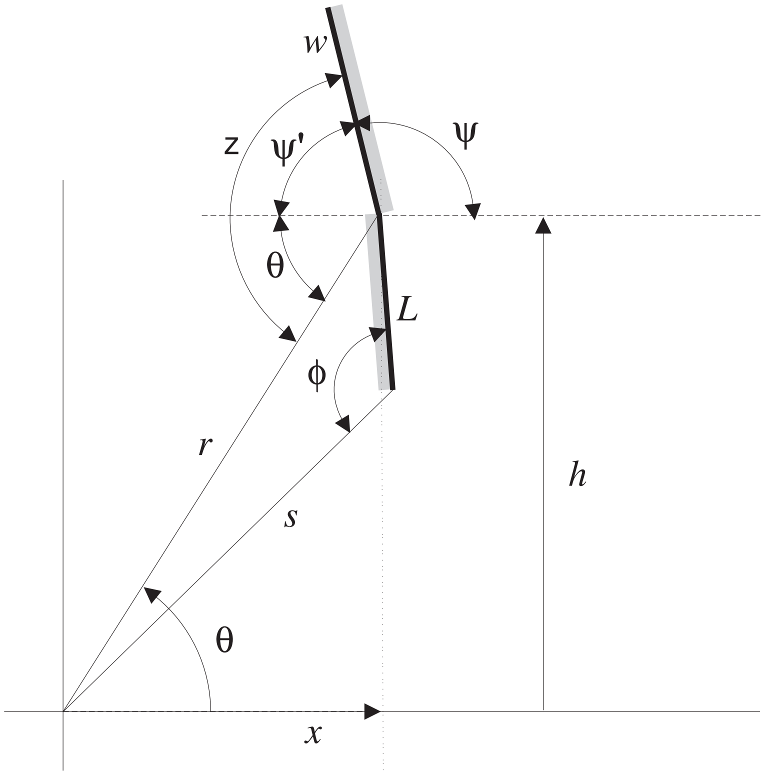


Figure 7-5: The rotor face must not move tangent to the slider face.

$z > \pi/2$

The worst case is when the rotor angle is at its smallest, which occurs when the tip of the rotor touches the lower end of the slider face as shown in Figure 7-5. If we define ψ' as the complement of the angle of the slider face, then by inspection of the figure:

$z = \psi' + \theta$

Because ψ' is the complement of ψ we can write:

$\psi' = \pi - \psi$

For the tip of the rotor to touch the bottom of the slider face, θ's value must be:

$\theta = \arcsin(h/r)$

Combining the previous four expressions we obtain the final form of the constraint:

$$\psi < \arcsin(h/r) + \pi/2 \tag{7.10}$$

If the slider face angle ψ is equal to 0, only one end of the slider face will touch the rotor. If ψ is less than 0, the rotor can touch the back side of the slider face, but this should not happen because contact is allowed only on the outside surface of a part. To ensure that the rotor cannot touch the back side of the rotor face, and that the contact can actually be face contact (rather than contact at just one end of the face) we enforce the constraint:

$$\psi > 0 \tag{7.11}$$

Equation 7.1 through Equation 7.11 are the complete set of constraints sufficient to ensure that the geometry in Figure 7-2 implements qcs-curve H. Now to complete this library entry, we must derive expressions for the end points of the qcs-curve. One end point corresponds to the configuration in Figure 7-5. This end point is the upper left end point of curve H. Its coordinates are:

$$\theta_1 = \arcsin(h/r) \tag{7.12}$$

$$x_1 = r\cos(\theta_1) \tag{7.13}$$

The second end point corresponds to the configuration in Figure 7-4. Its coordinates are (note that $x_2 = -x_1$):

$$\theta_2 = \pi - \arcsin(h/r) \tag{7.14}$$

$$x_2 = -r\cos(\theta_1) \tag{7.15}$$

By appropriate use of coordinate transformations, we can use this pair of faces to implement several other qcs-curves.[5] For example, if we measure the position of the slider positive to the left rather than to the right, we have an implementation for curve F. If we flip the geometry over as in Figure 7-6, still measuring the rotor angle positive counterclockwise, we have an implementation for curve B. If we flip the geometry over and measure the slider position positive to the left rather than to the right, we have an implementation for curve D. Thus, with the same basic geometry and constraints, we can implement all four kinds of diagonal curves for the case in which q_1 is rotation and q_2 is translation. By the obvious coordinate transformation, we can also use this geometry to implement all four kinds of diagonal curves for the

[5]The coordinate transformations change the expressions for the end points of the cs-curve (e.g., Equation 7.12) but they do not affect the constraints that ensure the curve is monotonic with proper slope (e.g., Equation 7.1).

case in which q_1 is translation and q_2 is rotation.

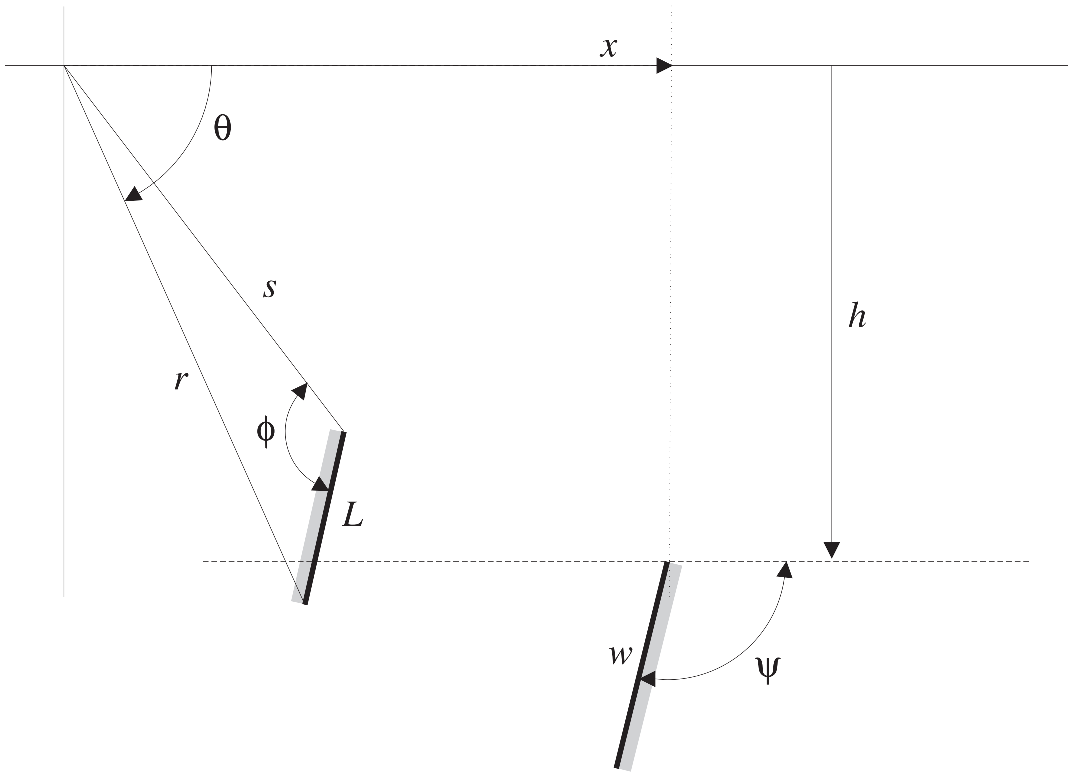


Figure 7-6: The parameterization of a pair of faces used to implement qcs-curve B. The rotating face rotates about the origin, its position measured positive *counterclockwise* with angle θ. The translating face translates horizontally, its position measured positive to the right with x. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

7.2.2 Cam and Centered Follower

As our second implementation for qcs-curve H, we use the two flat faces whose parameterization is shown in Figure 7-7. Here again, our goal is to derive a set of constraints on the parameters such the faces implement a monotonic curve with the same slope as curve H. This pair of faces is the same as the ones we used for the cam and offset follower (Figure 7-2), but here we explore a different region of the 7-dimensional parameter space; we explore the region corresponding to a cam and centered follower. [6]

[6]The parameterization in Figure 7-7 is entirely equivalent to that in Figure 7-2.

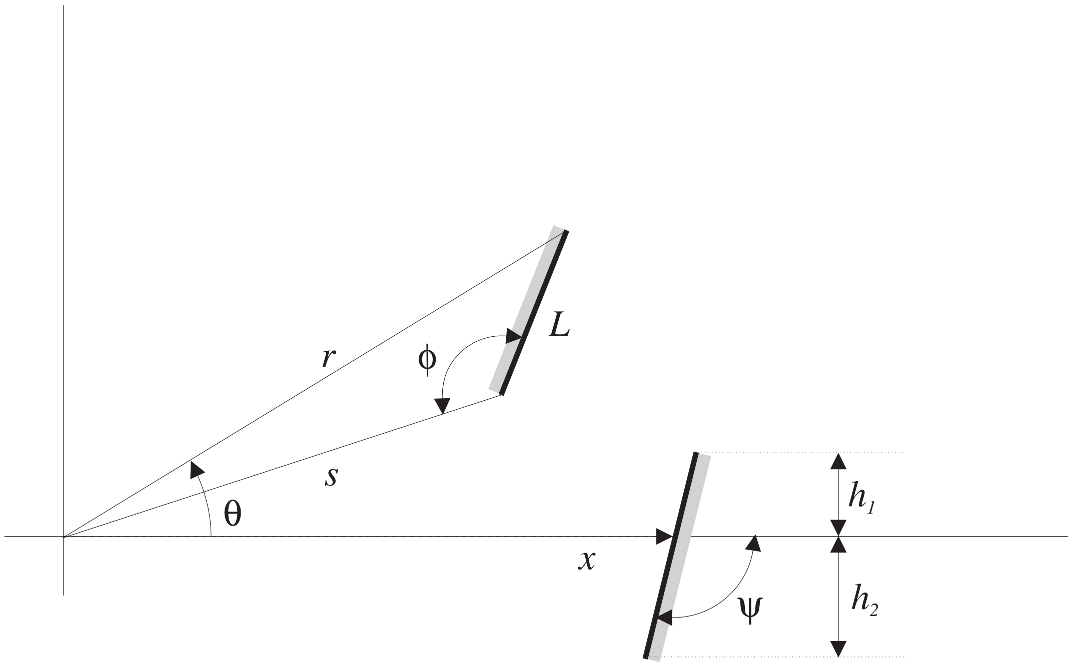


Figure 7-7: The parameterization of a pair of faces used to implement qcs-curve H. The rotating face rotates about the origin, its position measured positive counterclockwise with angle θ. The translating face translates horizontally, its position measured positive to the right with x. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

Our first constraint is that the rotor face exists, that is, it has non-zero length. Defining L as the length of the rotor face, we trivially obtain:

$$L > 0 \tag{7.16}$$

To ensure that the translating face passes through the pivot (a hallmark of centered-cam-and-follower behavior), we define the offsets between the ends of the slider face and the pivot, h_1 and h_2, subject to the constraints:

$$h_1 > 0 \tag{7.17}$$

$$h_2 > 0 \tag{7.18}$$

If ψ, the angle of the slider face, equals $\pi/2$, the interaction will be non-monotonic like the example in Figure 2-13. If $\psi = \pi$, the slider face will act as a stop, limiting the clockwise rotation of the rotor. Hence, we obtain limits on the allowable range of ψ:

$$\psi > \pi/2 \tag{7.19}$$

$$\psi < \pi \tag{7.20}$$

We prefer solutions in which the rotor face can make face contact (rather than just point contact) with the slider. Defining ϕ, the angle between the trailing edge of the rotor and the rotor face, we express this condition as:

$$\phi > \pi/2 \tag{7.21}$$

$$\phi < \pi \tag{7.22}$$

These two constraints are conservative. We can obtain the strict lower bound for ϕ by ensuring that, when the trailing edge of the rotor touches the slider, the rotor face never becomes perpendicular to the slider face. If ϕ was smaller than this strict lower bound, the back side of the rotor face could touch the slider, but this should not happen because contact is allowed only on the outside surface of a part. We can obtain the strict upper bound for ϕ by ensuring that when the rotor face touches the top end of the slider face, the rotor face is not horizontal. If the rotor face was horizontal in this configuration, the slider would act as a stop, preventing clockwise rotation of the rotor.

As with the previous library entry, we define the leading edge of the rotor to be r and the trailing edge to be s. Because of Equations 7.21 and 7.22, the leading edge is always longer than the trailing edge. r and s are related to the length of the rotating face by:

$$r = (s^2 + L^2 - 2sL\cos(\phi))^{1/2} \tag{7.23}$$

Next, we constrain the range of engagement so that the interaction is monotonic. If both h_1 and h_2 are greater than r, engagement is possible for all angles of the rotor. In this case, the cs-curve will be a closed curve in the cs-plane, rather than a monotonic curve. To prevent this, we must ensure that the faces disengage for both sufficiently large and sufficiently small angles of the rotor. To ensure that the faces disengage at large angles we require that:

$$s > h_1 \tag{7.24}$$

At the other extreme, we must ensure that (during clockwise rotation) the faces disengage before the rotor angle becomes small enough that the leading edge (r) of the rotor becomes perpendicular to the slider face. If the leading edge did become perpendicular, the rotor would act as a stop, preventing the slider from moving to the left. To derive the constraint that prevents this, we define two new quantities, a

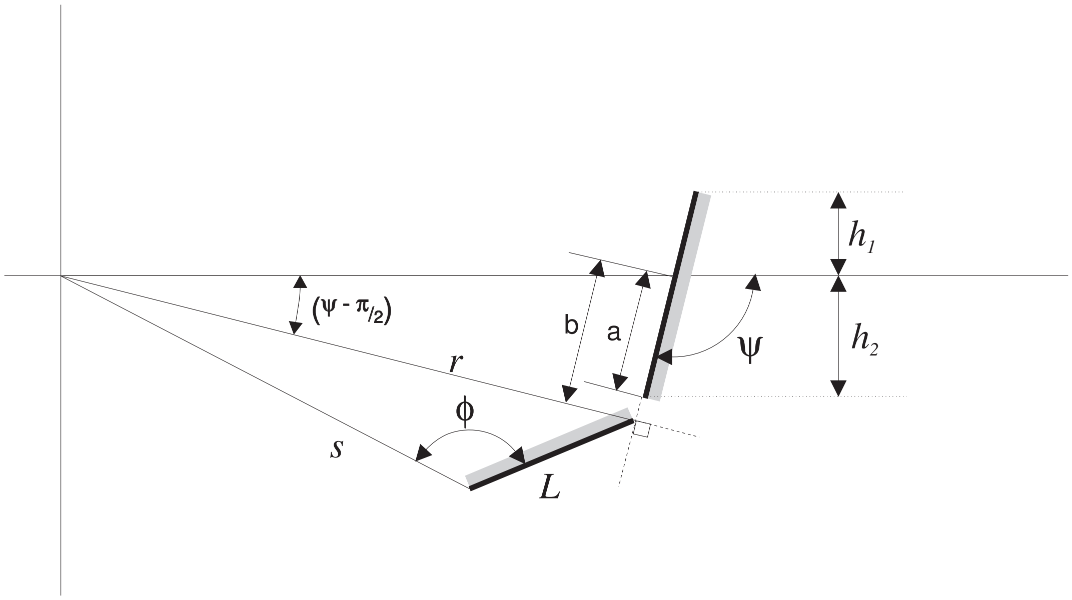


Figure 7-8: The leading edge of the rotor is perpendicular to the extended slider face.

and b, as shown in Figure 7-8. a is the length of the slider face below the x-axis:

$$a = h_2/\cos(\psi - \pi/2)$$

When the leading edge of the rotor touches and is perpendicular to the extended slider face, b is the distance measured along the extended slider face from the x-axis to contact point. When the rotor is in this configuration, the angle between the leading edge and the x-axis is $\psi - \pi/2$, and b's value is therefore:

$$b = r\tan(\psi - \pi/2)$$

Now, we can express the condition that the faces disengage before the leading edge becomes perpendicular to the slider face as:

$$a < b$$

Upon simplification, the previous three expressions yield the desired constraint:

$$0 > r/\tan(\psi) + h_2/\sin(\psi) \tag{7.25}$$

Equation 7.16 through Equation 7.25 are the primary constraints that ensure the geometry in Figure 7-7 implements qcs-curve H. Section 7.3 describes some additional

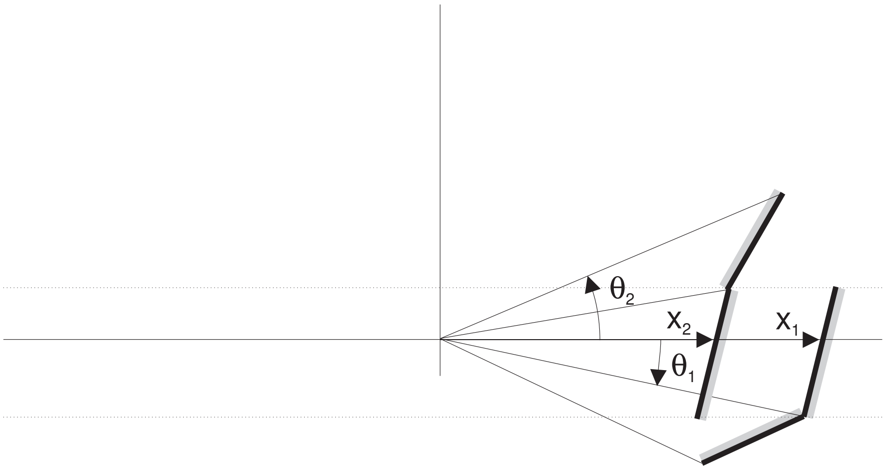


Figure 7-9: Two superimposed configurations of the rotor and slider. These configurations are the end points of the qcs-curve for the intended interaction between the rotor and slider.

constraints that must also be satisfied.

To complete this library entry, we must compute the coordinates of the two end points of the cs-curve for the interaction.[7] Figure 7-9 shows the configurations corresponding to the end points. One end point is the configuration in which the leading edge of the rotor face touches the lower end of the slider face. In qc-space, this end point is the upper left end of curve H. Its coordinates are:[8]

$$\theta_1 = -\arcsin(h_2/r) \tag{7.26}$$

$$x_1 = r\cos(\theta_1) - h_2/\tan(\psi) \tag{7.27}$$

The other end point is the configuration in which the trailing edge of the rotor face touches the upper end of the slider face:

$$\theta_2 = \arcsin(h_1/s) + \arccos\left(\frac{s^2 + r^2 - L^2}{2sr}\right) \tag{7.28}$$

$$x_2 = s\cos(\arcsin(h_1/s)) + h_1/\tan(\psi) \tag{7.29}$$

As with the cam and offset follower, by appropriate use of coordinate transformations, we can use this same basic geometry and constraints to implement all four kinds of diagonal curves for the case in which one body rotates and the other translates.

[7] This geometry actually produces two cs-curves. Here we discuss the end points of the desired curve. In Section 7.3 we discuss the other curve.

[8] Note that $\tan(\psi - \pi/2) = -1/\tan(\psi)$.

7.3 Parasitic Interactions

For the cam and offset follower we were able to construct constraints that ensure that there is only one range of engagement. Because there is a single range of engagement, the interaction is a single continuous curve in c-space. For the cam and centered follower, however, there are two ranges of engagement. The first range of engagement, shown in Figure 7-9, is the desired one, discussed in the previous section. We use this range to implement qcs-curves of type H. The second range of engagement occurs when the rotor points to the left and the slider is on the left side of the pivot as shown in Figure 7-10. This range produces an undesired, or parasitic, cs-curve which is disjoint from the desired one.

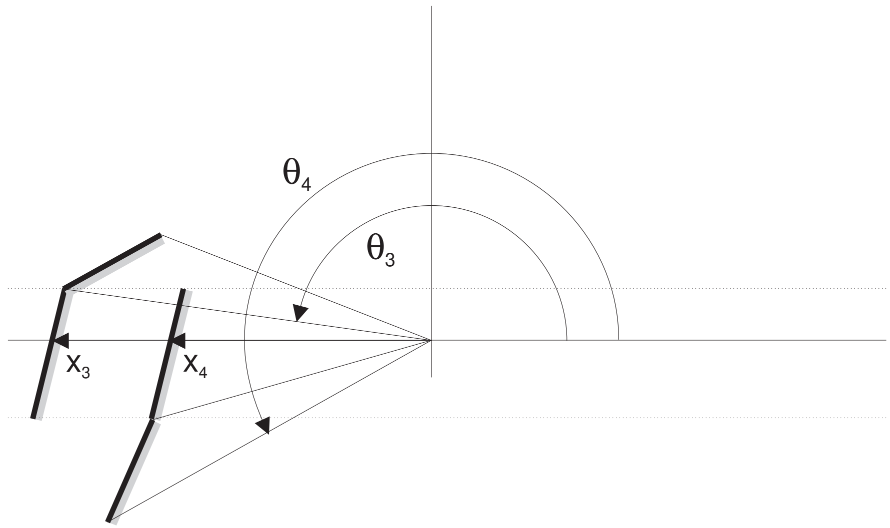


Figure 7-10: Two superimposed configurations of the rotor and slider. These configurations are the end points of the qcs-curve for the parasitic interaction between the rotor and slider.

Because the cam and centered follower produce a parasitic cs-curve in addition to the desired one, we can use this geometry to implement curves of type H, only if we can ensure that during the normal operation of the device, the parasitic engagement does not occur. We have explored, but have not implemented, two methods for dealing with this. The first method is to wait until more design details have been selected, then to use numerical simulation to verify that the parasitic engagement does not occur. If it does occur, we either reject the design or look for different parameter values, masses, springs, or actuators for which the engagement does not occur. For example, we might attempt to eliminate the parasitic engagement by selecting parameter values that make the rotor longer than the travel of the slider, thereby placing the parasitic engagement out of reach of the slider.

The second method is to derive additional constraints that ensure the parasitic engagement is outside the bounds of the normal operation of the device. To do this, we first examine the qualitative simulation to obtain the landmark values for the extremes of the motion of the rotor and slider. Measuring angles in the range $-\pi$ to π, we call the maximum and minimum positions of the rotor θ_{max} and θ_{min} respectively. Similarly, we call the maximum and minimum positions of the slider x_{max} and x_{min} respectively.

Next we compute expressions for the end points of the parasitic curve. Figure 7-10 shows the configurations for the two end points. One end point is the configuration in which the leading edge of the rotor touches the upper end of the slider face. Its coordinates are:

$$\theta_3 = \pi - \arcsin(h_1/r) \tag{7.30}$$

$$x_3 = r\cos(\theta_3) + h_1/\tan(\psi) \tag{7.31}$$

The other end point is the configuration in which the trailing edge of the rotor face touches the bottom end of the slider face. Measuring angles in the range $-\pi$ to π, the coordinates are:

$$\theta_4 = -\pi + \arcsin(h_2/s) + \arccos(\frac{s^2 + r^2 - L^2}{2sr}) \tag{7.32}$$

$$x_4 = s\cos(-\pi + \arcsin(h_2/s)) - h_2/\tan(\psi) \tag{7.33}$$

Finally, we construct expressions that ensure that the range of motion is outside the bounding box defined by the end points of the parasitic curve. Measuring θ in the range $-\pi$ to π, the constraints are:

$$\left\{ \begin{array}{c} \theta_{max} < \theta_3 \\ \theta_{min} > \theta_4 \end{array} \right\} \tag{7.34}$$

or

$$\{x_{mim} > x_4\} \tag{7.35}$$

There are two ways to satisfy this constraint: either the rotor does not turn far enough (Equation 7.34), or the slider stroke is not long enough for the parasitic engagement to occur (Equation 7.35). For completeness, we could add another disjunct, $x_{max} < x_3$, however, we know that this will not be satisfied: For the desired interaction to occur, x must spend some time in the range between x_1 and x_2 (see Figure 7-9). Because this range is to the right of x_3, the additional disjunct will not be satisfied.

As Figure 7-10 shows, during the parasitic engagement, the slider touches the back side of the rotor face. However, once the rotor face is embedded in a solid component, it is not possible to touch its back side. Hence, we have a second reason why we must ensure that the parasitic engagement is out of the range of normal operation of the device.

7.4 Where to Attach the Faces

We describe the pair of faces in a library entry with respect to a local coordinate system included in that entry. To attach the faces to actual components, we must first transform the coordinates of the faces to match those of the components. Fortunately, we can precompute the necessary coordinate transformations and include them in the library. Hence, this task is performed as part of constructing the library.

We illustrate this task by deriving expressions to transform the coordinates of the faces in Figure 7-2 to match those of an arbitrary pair of components. Our goal is a set of templates (such as Figure 7-15) that show where to attach the faces.

We start by placing the library frame (the coordinate frame of the faces) in the global coordinate frame as shown in Figure 7-11. The origin of the library frame is located at the point $(\mathbf{X_L}, \mathbf{Y_L})$ in the global frame and is inclined by angle Ω_L from the horizontal.

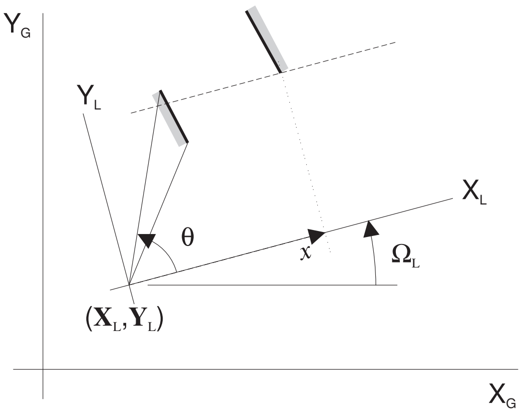


Figure 7-11: The location of the library coordinate frame with respect to the global coordinate frame.

(We use bold serif fonts, like $\mathbf{X_L}$, to refer to the origin of a coordinate frame. We use sans serif fonts, like $\mathsf{X_R}$, to refer to an axis of a coordinate frame.)

Next, we place the rotor frame (the coordinate frame of the component to which we attach the rotating face) in the global coordinate frame as shown in Figure 7-12. The origin of the rotor frame (i.e., the rotor's pivot) is located at the point $(\mathbf{X_R}, \mathbf{Y_R})$ in the global frame. θ_R is the rotation angle of the rotor.

Similarly, we place the slider frame (the coordinate frame of the component to which we attach the translating face) in the global coordinate frame as shown in

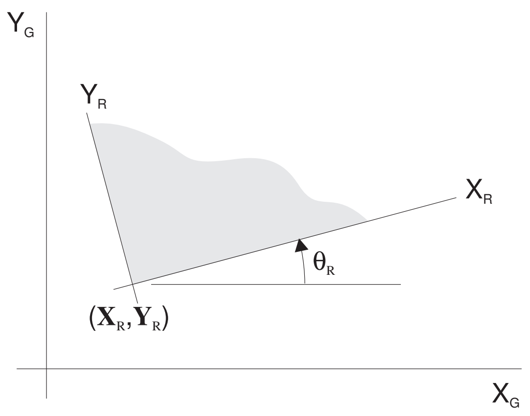


Figure 7-12: The location of the rotor coordinate frame with respect to the global coordinate frame.

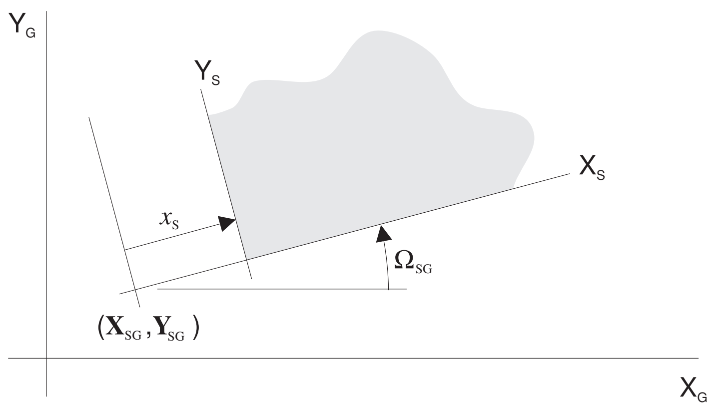


Figure 7-13: The location of the slider coordinate frame with respect to the global coordinate frame.

Figure 7-13. The slider frame translates along a fixed guide inclined an angle Ω_{SG} from horizontal. The *guide's* origin is located at the point $(\mathbf{X_{SG}}, \mathbf{Y_{SG}})$ in the global frame. x_S is the displacement of the slider along the guide.

The rotating face rotates about the origin of the library frame, and the rotor rotates about the origin of the rotor frame. Because the face is rigidly attached to the rotor, the face and the rotor must rotate about the same point. Hence, the two origins must actually be the same point:

$$\mathbf{X_L} = \mathbf{X_R}$$
$$\mathbf{Y_L} = \mathbf{Y_R}$$

Similarly, the translating face and the slider must translate in the same direction:

$$\Omega_L = \Omega_{SG}$$

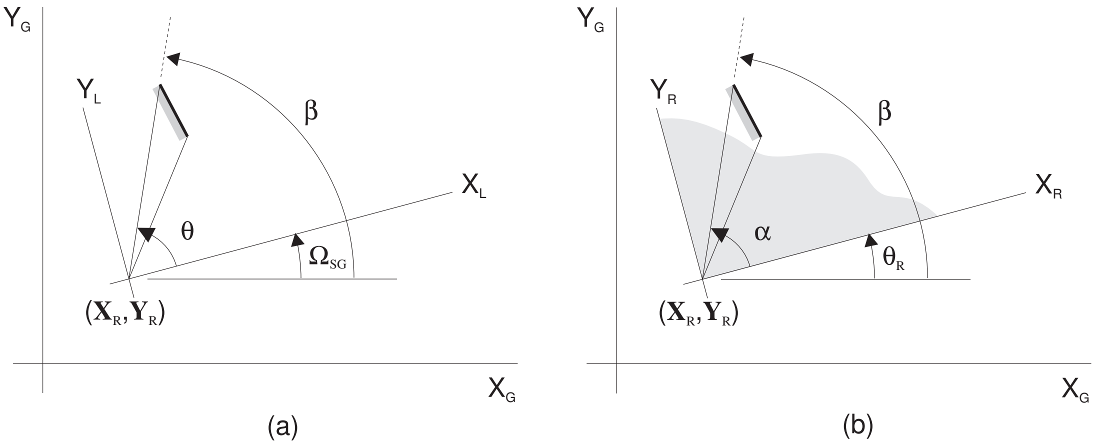


Figure 7-14: (a) The location of the rotating face in the library frame. (b) The location of the rotating face in the rotor frame.

Figure 7-14a shows a new version of the library frame in which we have included our new knowledge about the location of the origin and the inclination from horizontal. The figure also shows the absolute rotation angle, β, of the rotating face. We can express this angle in terms of the other angles in the figure:

$$\beta = \Omega_{SG} + \theta$$

Figure 7-14b shows another view of the rotor frame in which we have rigidly attached the rotating face to the rotor body. The face is attached to the rotor at some unknown angular offset, α, which is a free design parameter. The figure also shows the absolute rotation angle, β, of the rotating face. We can derive a new expression for β in terms of the angles in the figure:

$$\beta = \theta_R + \alpha$$

If we combine these two expressions for β, we obtain the desired expression relating θ_R to θ:

$$\theta_R = \theta + \Omega_{SG} - \alpha$$

For our convenience, we eliminate α in favor of a new parameter θ_{off}. The new parameter is related to the old one by:

$$\theta_{off} = \Omega_{SG} - \alpha$$

Using this new parameter, our expression relating θ_R to θ takes the simple form:

$$\theta_R = \theta + \theta_{off} \tag{7.36}$$

Similarly, we eliminate α from Figure 7-14b to obtain Figure 7-15. We use this new figure as our template for attaching the rotating face to a rotating body.

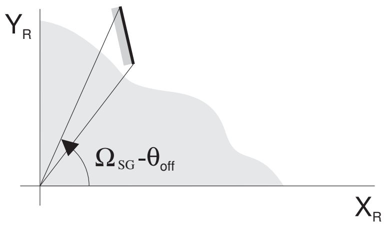


Figure 7-15: Template indicating where to attach the rotating face to the rotor.

Now we turn our attention to the translating face. Figure 7-16 shows the slider guide, slider, and library frames in the global frame. Our goal is to determine the location of point **P**, the bottom end of the translating face, with respect to the slider frame.

We begin by expressing **P** in the library frame:

$$\mathbf{P}^L = \begin{pmatrix} x \\ h \end{pmatrix}$$

where the superscript "L" indicates that **P** is expressed in the library frame.

Next, we express **P** in the frame of the *slider guide*:

$$\mathbf{P}^{SG} = \mathbf{P}^{L} + \begin{pmatrix} \Delta\mathsf{X} \\ \Delta\mathsf{Y} \end{pmatrix}$$

where the point ($\Delta\mathsf{X}$, $\Delta\mathsf{Y}$) is the location of the library frame with respect to slider guide frame. (Here again, the superscript "SG" indicates the coordinates are expressed in the slider guide frame.) We use a standard rotation matrix (rotating by $-\Omega_{SG}$) to express this location in terms of the absolute locations (i.e., the locations with respect to the global frame) of the slider guide frame and the library frame:

$$\begin{pmatrix} \Delta\mathsf{X} \\ \Delta\mathsf{Y} \end{pmatrix} = \begin{bmatrix} \cos(\Omega_{SG}) & \sin(\Omega_{SG}) \\ -\sin(\Omega_{SG}) & \cos(\Omega_{SG}) \end{bmatrix} \begin{pmatrix} \mathbf{X}_R - \mathbf{X}_{SG} \\ \mathbf{Y}_R - \mathbf{Y}_{SG} \end{pmatrix}$$

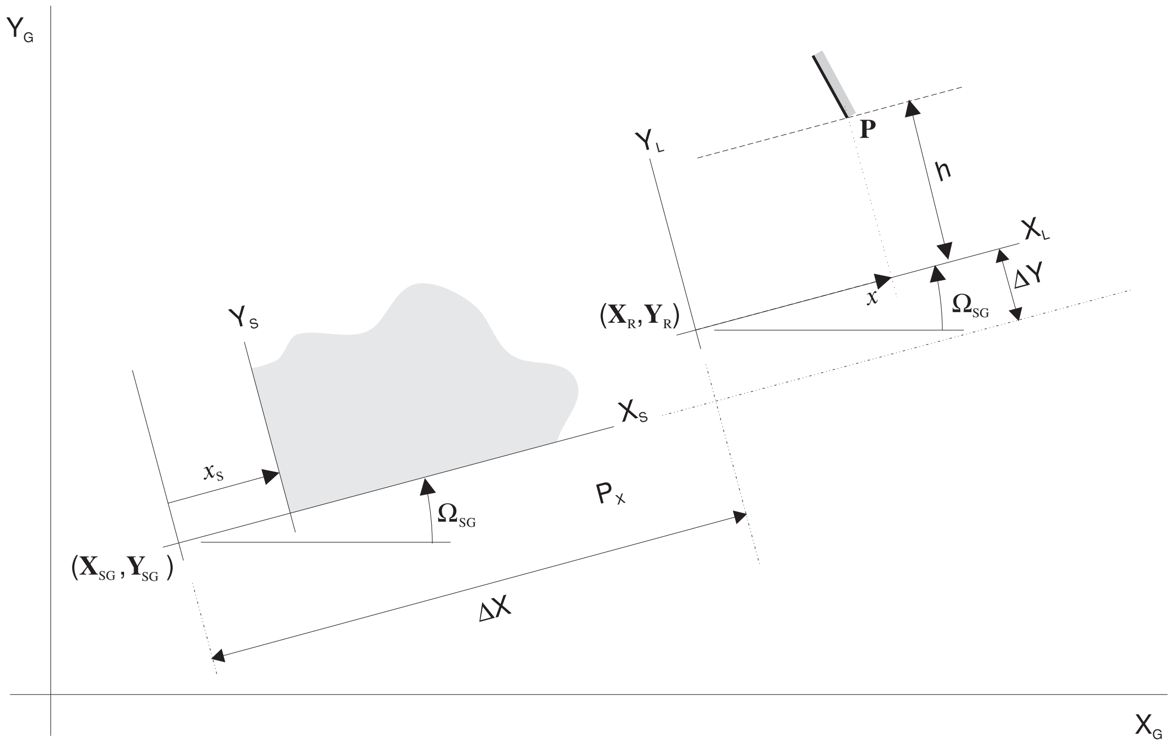


Figure 7-16: The slider frame and the library frame in the global frame.

Because, the slider frame is displaced from the slider guide frame by a distance x_S, we can use our expression for $\mathbf{P}^{SG}$ to obtain an expression for $\mathbf{P}$ in the slider frame:

$$\mathbf{P}^{S} = \mathbf{P}^{SG} + \begin{pmatrix} -x_S \\ 0 \end{pmatrix}$$

Combining all of the previous expressions we obtain:

$$\mathbf{P}^S = \begin{pmatrix} x \\ h \end{pmatrix} + \begin{bmatrix} \cos(\Omega_{SG}) & \sin(\Omega_{SG}) \\ -\sin(\Omega_{SG}) & \cos(\Omega_{SG}) \end{bmatrix} \begin{pmatrix} \mathbf{X_R} - \mathbf{X_{SG}} \\ \mathbf{Y_R} - \mathbf{Y_{SG}} \end{pmatrix} + \begin{pmatrix} -x_S \\ 0 \end{pmatrix}$$

This gives us the location of $\mathbf{P}^S$ in terms of several design parameters (Ω_{SG}, $\mathbf{X_R}$, $\mathbf{Y_R}$, $\mathbf{X_{SG}}$, and $\mathbf{Y_{SG}}$) and two position parameters: x and x_S. Because the face is rigidly attached to the slider, the position of $\mathbf{P}$ in the slider frame must be a constant. Hence, we must be able to eliminate the two position parameters (which are variables) from our expression for $\mathbf{P}^S$.

To this end, we note that x and x_S measure the positions of the face and the slider (respectively) with respect to references that are fixed in the global frame. Also, because the face is rigidly attached to the slider, if the face undergoes a displacement, the slider must undergo the identical displacement. These two facts combined require that x_S and x differ by at most a constant. Calling that constant x_{off} we obtain:

$$x_S = x + x_{off} \tag{7.37}$$

Substituting this expression into our previous expression for $\mathbf{P}^S$ we obtain the desired result:

$$\mathbf{P}^S = \begin{bmatrix} \cos(\Omega_{SG}) & \sin(\Omega_{SG}) \\ -\sin(\Omega_{SG}) & \cos(\Omega_{SG}) \end{bmatrix} \begin{pmatrix} \mathbf{X_R} - \mathbf{X_{SG}} \\ \mathbf{Y_R} - \mathbf{Y_{SG}} \end{pmatrix} + \begin{pmatrix} -x_{off} \\ h \end{pmatrix} \tag{7.38}$$

With this expression, we now know where to attach the face to the slider. We simply attach the face so that its lower end is located at the point $\mathbf{P}^S$, as shown in Figure 7-17.

As a second example, we consider the faces in Figure 7-2 for the case in which we measure x (the position of the translating face) positive to the left rather than to the right. As we discussed previously, defining x positive to the left turns this geometry into an implementation for qcs-curves of type F in Figure 7-1.

In this case the position of the rotor and the position of the rotating face are still related by Equation 7.36, and we still use Figure 7-15 as our template for attaching the face to the rotor. Similarly, the position of the slider and the position of the translating face are still related by Equation 7.37. However, we must derive a new template for attaching the translating face to the slider.

In the previous case, when we measured x positive to the right, we placed the library frame and the slider frame into the global frame such that the axes of the library and slider frames pointed in the same direction. (That is, $\mathsf{X_L}$ pointed in the same direction as $\mathsf{X_S}$, and $\mathsf{Y_L}$ pointed in the same direction $\mathsf{Y_S}$ as shown in Figure 7-16.) However, when we measure x positive to the left rather than to the right, we place the two frames in the global frame such that their axes point in opposite directions.

Doing this, we derive a new expression for the location of **P** in the slider frame:

$$\mathbf{P}^S = \begin{bmatrix} \cos(\Omega_{SG}) & \sin(\Omega_{SG}) \\ -\sin(\Omega_{SG}) & \cos(\Omega_{SG}) \end{bmatrix} \begin{pmatrix} \mathbf{X_R} - \mathbf{X_{SG}} \\ \mathbf{Y_R} - \mathbf{Y_{SG}} \end{pmatrix} + \begin{pmatrix} -x_{off} \\ -h \end{pmatrix} \tag{7.39}$$

The difference between this expression and Equation 7.38 is the sign in front of "h." We also derive a new template, shown in Figure 7-18 indicating where to attach the face to the slider. In our previous template, the face was located in the first quadrant of the coordinate system. Here, because the axes of the slider frame are rotated 180 degrees from the previous case, the face is in the third quadrant.

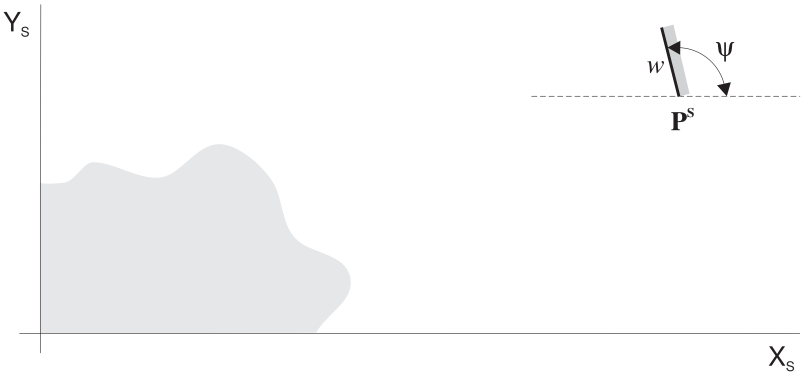


Figure 7-17: Template indicating where the translating face is attached to the slider: The lower end of the translating face is attached to the slider at the point $\mathbf{P}^S$.

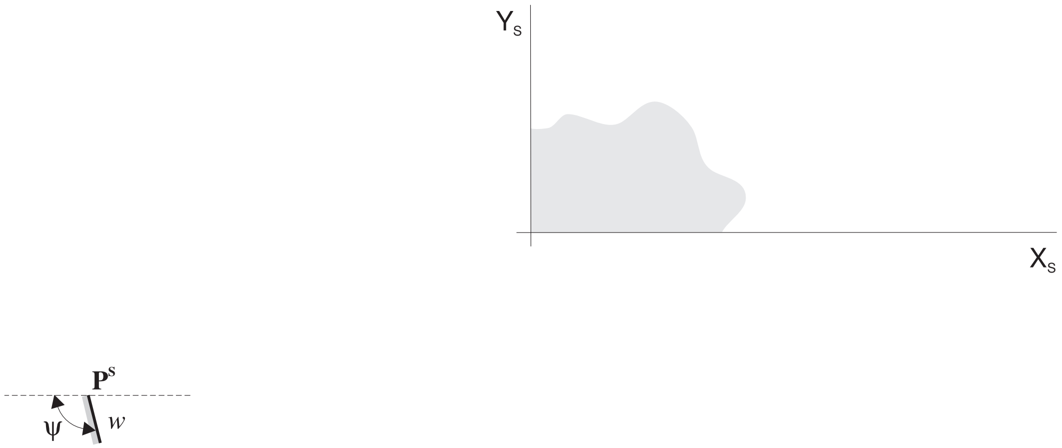


Figure 7-18: A template for attaching the translating face to the slider when the position, x, of the translating face is measured positive to the left relative to the library frame.

7.5 Assembling the BEP-Model

In this section, we describe how SKETCHIT uses library entries like those in Section 7.2 and templates like those in Section 7.4 to construct a BEP-Model. We illustrate this process by an example, showing how SKETCHIT constructs BEP-Models for the circuit breaker qc-space in Figure 7-19. Because qualitative simulation has verified that this qc-space produces the desired behavior, the BEP-Models SKETCHIT derives from this qc-space will define families of designs that all produce the desired behavior.

To construct a BEP-Model, SKETCHIT must construct an implementation for all of the qcs-curves in Figure 7-19.[9] For each qcs-curve SKETCHIT: selects an appropriate library entry; creates new instances of the entry's parameters; assigns values to the landmarks for the end points of the qcs-curve; instantiates the constraints from the library entry; instantiates the parametric geometry contained in the library entry; and instantiates the parametric geometry in the templates that indicate how the faces are attached to components.

In our example, we consider how SKETCHIT constructs the portion of the BEP-Model that relates to the cam-follower interaction (top of Figure 7-19). Figure 7-20 shows the constraints for this piece of the BEP-Model. Because the cam-follower qcs-curve is of type F, the geometry in Figure 7-2 is a valid implementation, as long as the position of the slider is measured positive to the left (i.e., the direction of positive displacement is the same as the negative x-axis of the library frame). This is in fact, the library entry that SKETCHIT selected for this BEP-Model.

To use this library entry, or any other entry, SKETCHIT must create new instances of the parameters it contains. SKETCHIT does this by appending a unique number to the end of each parameter name. For example, `H_4` is a new instance of the parameter h.

The first four constraints in Figure 7-20 assign values to the landmarks of the end points of the cam-follower qcs-curve using Equations 7.12 through 7.15.[10] As Equation 7.36 and Equation 7.37 require, SKETCHIT includes in the value of each landmark, a term for the offset between the coordinates of the components and the coordinates of the faces (i.e., `off_1` and `off_2`). The values for landmarks $\mathtt{LM}_8$ and $\mathtt{LM}_{10}$ include extra negative signs (compared with Equations 7.13 and 7.15) because

[9] No implementation is necessary for qcs-curves corresponding to spring neutral positions or motion limits of actuators because these curves do not represent interacting faces.

[10] To be rigorous we must constrain the landmarks of rotating bodies to be between 0 and 2π. This is true for bodies that are intended to rotate less than a full revolution as well as for bodies that are intended to rotate more that a full revolution. In the former case, constraining the landmarks to be between 0 and 2π ensures that the implementation will actually rotate less than a full revolution. In the latter case, the device's qc-space is actually a flattened out representation of a cylinder or torus (see Section 4.7). In this flattened out representation all landmarks are explicitly constrained to be between the boundaries at 0 and 2π. For the circuit breaker example, we did not explicitly constrain the landmarks of the lever to be between 0 and 2π. However, for the yoke and rotor example, we did explicitly constrain the landmarks of the rotor.

of a coordinate transformation: here we measure the position of the slider positive to the left in Figure 7-2 rather than to the right.

The remaining constraints in Figure 7-20 are instances of Equations 7.1 through 7.11. These are the constraints that ensure the cs-curve for the cam-follower interaction is monotonic with the desired slope.

Figure 7-20 shows only the constraints that the cam-follower qcs-curve contributes to the BEP-Model. The cam-follower also contributes parametric geometry to the BEP-Model. This contribution consists of the parametric geometry defining the faces shown in Figure 7-2 and the templates indicating how the faces are attached to the components shown in Figure 7-15 and Figure 7-18.

This completes the process of implementing the cam-follower qcs-curve. SKETCHIT repeats this process for the push-pair, lever-stop, and pushrod-stop qcs-curves.

Next, SKETCHIT assigns values to the landmarks for the i-points. Consider the i-point defined by the intersection of the horizontal "hook=cold" line and the cam-follower curve in the top of Figure 7-19. The location of this intersection determines the value of landmark $\mathtt{LM}_E$. SKETCHIT reports this constraint with the statement "`(LM_E = (INTERSECT (HOOK = LM_16) CAM-FOLLOWER))`," indicating that the value of landmark $\mathtt{LM}_E$ is defined by the intersection between the horizontal line located at landmark $\mathtt{LM}_{16}$ and the cam-follower qcs-curve. Hence, to compute the value of $\mathtt{LM}_E$ the constraint solver (the program that evaluates the BEP-Model) must compute the intersection between the horizontal line and the qcs-curve.

There are 5 i-points in Figure 7-19. Each one produces a constraint like the one defining the value of $\mathtt{LM}_E$. Figure 7-21 shows the five constraints.

Finally, to complete the BEP-Model, SKETCHIT instantiates constraints on the relative locations of the landmarks. As we described in Section 4.6, the primary landmarks are completely ordered with respect to themselves. The i-points for a particular qcs-curve are ordered with respect to both themselves and the primary landmarks, but are not necessarily ordered with respect to the i-points of the other qcs-curves. Figure 7-22 shows the constraints SKETCHIT instantiates to enforce the landmark orderings for the qc-space in Figure 7-19.[11]

Figure 7-23 shows the complete set of constraints for the BEP-Model constructed in this example. In addition to using the cam-and-offset-follower library entry for the cam-follower, SKETCHIT has selected the cam-and-centered-follower library entry for the push-pair (Figure 7-7), the translating rotor-stop library entry for the lever-stop (Appendix A), and the fixed slider-stop library entry for the pushrod-stop (Appendix A). Figure 7-24 shows a hand generated solution to this BEP-Model. This solution contains connective geometry in addition to the faces defined by the BEP-Model.

[11] Some of these constraints are redundant, however, this causes no difficultly. For example, to enforce the ordering of the i-points of the cam-follower qcs-curve, SKETCHIT constructs the constraints `(LM_E < LM_11)` and `(LM_9 < LM_E)`. Then, when enforcing the ordering of the primary landmarks, SKETCHIT constructs the constraint `(LM_9 < LM_11)`. However, this constraint is subsumed by the other two, and hence is unnecessary.

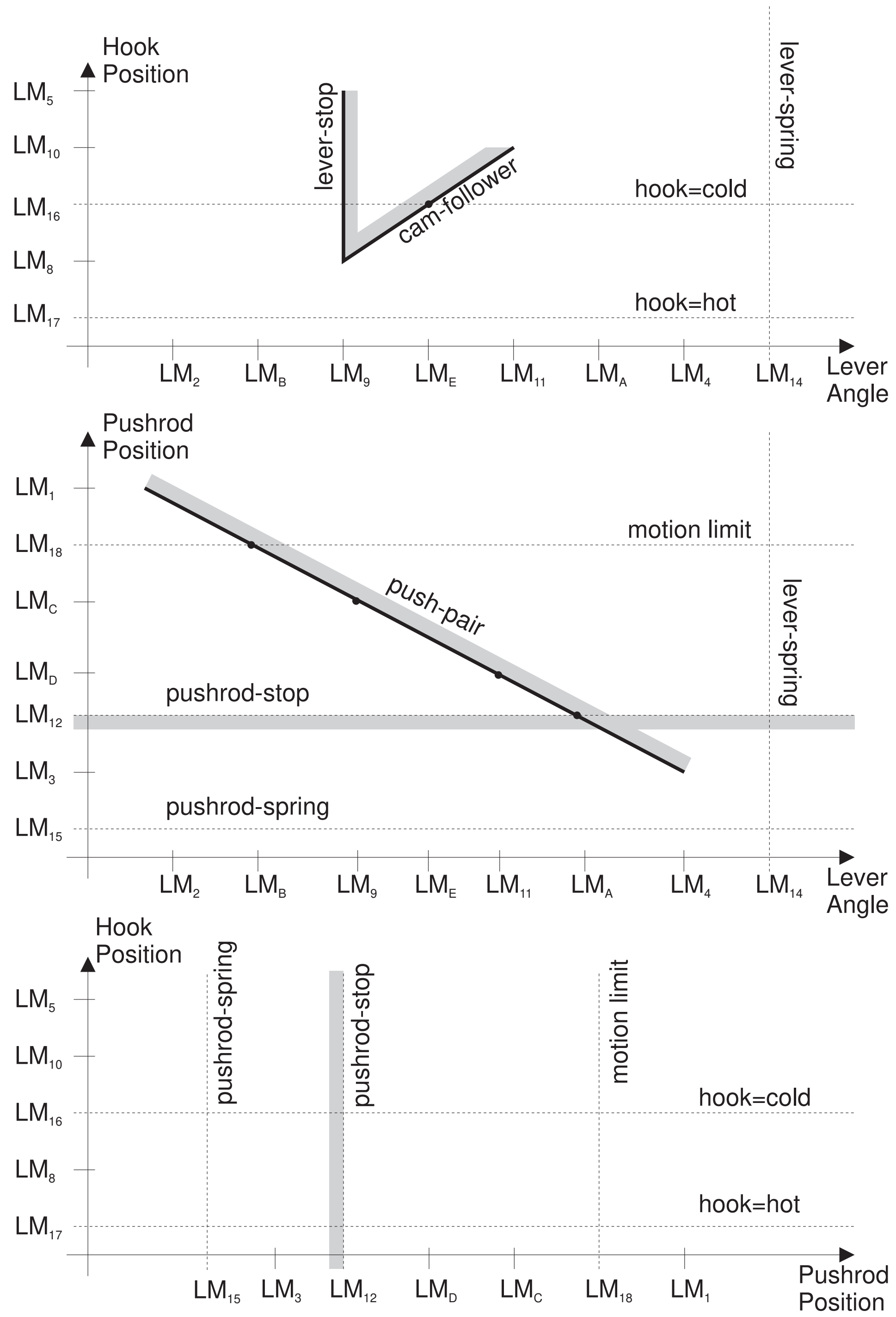


Figure 7-19: A circuit breaker qc-space that provides the desired behavior.

```
LM_9 = ASIN (H_4 / R_6) + off_1
LM_8 = - (R_6 * COS (ASIN (H_4 / R_6))) + off_2
LM_11 = 180 - ASIN (H_4 / R_6) + off_1
LM_10 = - (- R_6 * COS (ASIN (H_4 / R_6))) + off_2
W_8 > 0
L_7 > 0
H_4 > 0
R_6 = SQRT (S_5 ^2 + L_7 ^2 - 2* S_5 * L_7 * COS (PHI_9))
R_6 > H_4
S_5 < H_4
PHI_9 > 90
ACOS (H_4 / R_6)
 + ACOS((L_7 ^2 + R_6 ^2 - S_5 ^2) / (2 * L_7 * R_6)) < 90
PHI_9 <= 180
PSI_10 < ASIN (H_4 / R_6) + 90
PSI_10 > 0)
```

Figure 7-20: The constraints for the portion of the BEP-Model describing the cam-follower interaction.

```
LM_E = (INTERSECT (HOOK = LM_16) CAM-FOLLOWER)
LM_B = (INTERSECT (PUSHROD = LM_18) PUSH-PAIR)
LM_C = (INTERSECT (LEVER = LM_9) PUSH-PAIR)
LM_D = (INTERSECT (LEVER = LM_11) PUSH-PAIR)
LM_A = (INTERSECT (PUSHROD = LM_12) PUSH-PAIR)
```

Figure 7-21: The i-points for the circuit breaker qc-space.

```
LM_18 < LM_1        LM_C < LM_18        LM_D < LM_C
LM_12 < LM_D        LM_3 < LM_12        LM_15 < LM_3
LM_12 < LM_18       LM_4 < LM_14        LM_A < LM_4
LM_11 < LM_A        LM_E < LM_11        LM_9 < LM_E
LM_B < LM_9         LM_2 < LM_B         LM_9 < LM_11
LM_2 < LM_9         LM_11 < LM_4        LM_10 < LM_5
LM_16 < LM_10       LM_8 < LM_16        LM_17 < LM_8
```

Figure 7-22: The landmark orderings for the circuit breaker qc-space.

```
;; cam-follower
LM_9 = ASIN (H_4 / R_6) + off_1
LM_8 = - (R_6 * COS (ASIN (H_4 / R_6))) + off_2
LM_11 = 180 - ASIN (H_4 / R_6) + off_1
LM_10 = - (- R_6 * COS (ASIN (H_4 / R_6))) + off_2
W_8 > 0
L_7 > 0
H_4 > 0
R_6 = SQRT (S_5 ^2 + L_7 ^2 - 2* S_5 * L_7 * COS (PHI_9))
R_6 > H_4
S_5 < H_4
PHI_9 > 90
ACOS (H_4 / R_6)
 + ACOS((L_7 ^2 + R_6 ^2 - S_5 ^2) / (2 * L_7 * R_6)) < 90
PHI_9 <= 180
PSI_10 < ASIN (H_4 / R_6) + 90
PSI_10 > 0

;; push-pair
LM_2 = - (ASIN (H1_30 / S_32)
       + ACOS((S_32 ^2 + R_33 ^2 - L_34 ^2) / (2* S_32 * R_33))
       + off_3
LM_1 = - (S_32 *COS(ASIN(H1_30 / S_32)) + H1_30 / TAN(PSI_36)) + off_4
LM_4 = ASIN (H2_31 / R_33) + off_3
LM_3 = - (R_33 * COS(ASIN(H2_31 / R_33)) - H2_31 / TAN(PSI_36)) + off_4
L_34 > 0
H1_30 > 0
H2_31 > 0
PSI_36 > 90
PSI_36 < 180
PHI_35 < 180
PHI_35 > 90
R_33 = SQRT (S_32 ^2 + L_34 ^2 - 2* S_32 * L_34 * COS (PHI_35))
S_32 > H1_30
0 > R_33 / TAN (PSI_36) + H2_31 / SIN (PSI_36)

;; lever-stop
LM_9 = off_5
LM_8 = R_21 - LS_19 + off_6
```

Figure 7-23: The complete set of constraints of a BEP-Model for the circuit breaker. (continued)

```
LM_5 = R_21 + LR_18 + off_6
LS_19 > 0
LR_18 > 0
R_21 > 0

;; pushrod-stop
LM_12 = off_7
E_3 > 0
PHI_2 > 0
PHI_2 < 180
L1_1 > 0

LM_E = (INTERSECT (HOOK = LM_16) CAM-FOLLOWER)
LM_B = (INTERSECT (PUSHROD = LM_18) PUSH-PAIR)
LM_C = (INTERSECT (LEVER = LM_9) PUSH-PAIR)
LM_D = (INTERSECT (LEVER = LM_11) PUSH-PAIR)
LM_A = (INTERSECT (PUSHROD = LM_12) PUSH-PAIR)


LM_18 < LM_1          LM_C < LM_18          LM_D < LM_C
LM_12 < LM_D          LM_3 < LM_12          LM_15 < LM_3
LM_12 < LM_18         LM_4 < LM_14          LM_A < LM_4
LM_11 < LM_A          LM_E < LM_11          LM_9 < LM_E
LM_B < LM_9           LM_2 < LM_B           LM_9 < LM_11
LM_2 < LM_9           LM_11 < LM_4          LM_10 < LM_5
LM_16 < LM_10         LM_8 < LM_16          LM_17 < LM_8
```

Figure 7-23 continued: The complete set of constraints of a BEP-Model for the circuit breaker.

7.6 Exporting to DesignView

We use DesignView to evaluate BEP-Models. Currently, we transfer the BEP-Models to DesignView manually, but this can be done automatically using DesignView's macro language.

Our BEP-Models provide DesignView with enough information to solve all of the constraints except those defining the locations of the i-points, such as the constraints shown in Figure 7-21. To solve these constraints, DesignView must compute the intersection between a diagonal cs-curve and a horizontal or vertical line. This requires that DesignView have access to either a symbolic or numerical representation for the shape of the cs-curve. We provide DesignView only with the location of the end points of a cs-curve, not the shape of the curve itself.[12]

[12]For the yoke and rotor example we did provide DesignView with an algebraic expression for the

Because DesignView is a numerical solver, it is adequate to express the shape of a cs-curve numerically; for this we can use the techniques developed in [26]. Thus, to complete the implementation, we can connect DesignView to a program that can numerically compute the shape of a diagonal cs-curve as a function of the geometric parameters of the pair of faces. This will provide the last needed link and allow DesignView to enforce all necessary constraints.

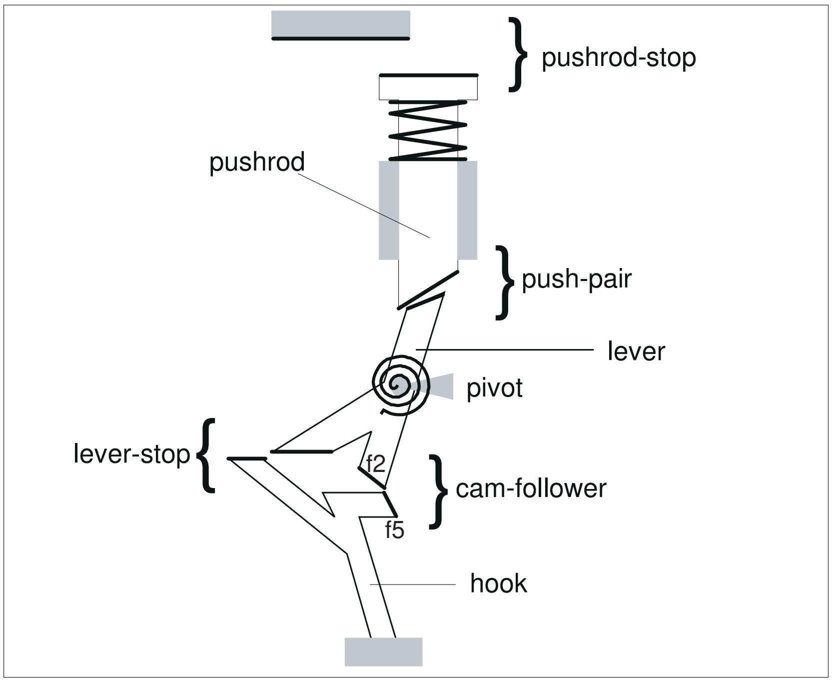


Figure 7-24: A hand generated solution to the BEP-Model.

shapes of the cs-curves, thus allowing DesignView to enforce all of the constraints of the BEP-Model. We are continuing to develop algebraic expressions for the curves produced by the library entries.

7.7 Periodic Boundary Conditions

To facilitate simulation, SketchIT flattens out cylindrical qc-spaces into planes with boundaries at 0 and 2π. During synthesis SketchIT must in effect reconstruct the cylindrical qc-space from the flattened representation.

The reconstruction process focuses on the qcs-curves that cross the boundaries. In the flattened out representation, the boundary-crossing qcs-curves are split into two pieces: one piece just above the 0 boundary and another just below the 2π boundary. To facilitate synthesis, SketchIT must join these pieces back together.

SketchIT joins the two pieces of a split curve by subtracting 2π from the ordinates of the end points and i-points of the piece of the curve near the 2π boundary (here we assume that the ordinate is the dimension that wraps around to form a cylinder). For example, Figure 7-25 shows the flattened qc-space for the yoke and rotor from Section 1.1.3 and Figure 7-26 shows the result of joining the boundary-crossing curves back together.

Finally, to construct a BEP-Model, SketchIT selects appropriate library entries for the re-joined boundary-crossing qcs-curves. Because the remaining qcs-curves lie entirely between the boundaries of the flattened out representation, SketchIT can directly select library entries for them without any additional effort.

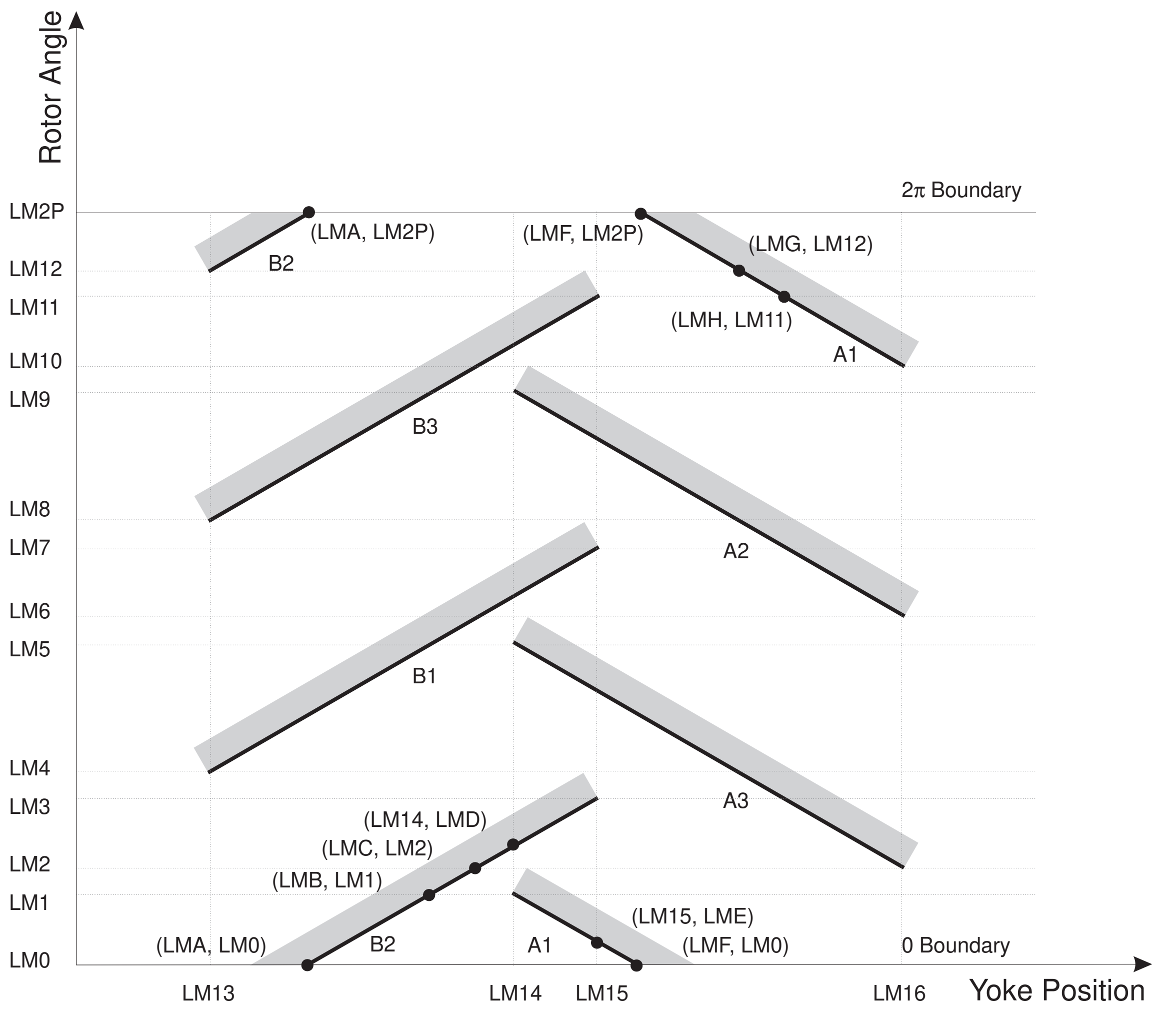


Figure 7-25: The qc-space for the yoke and rotor flattened to produce a plane. The labels on the qcs-curves indicate the names of the interacting faces. For example, curve A1 is the interaction between face A on the yoke and face 1 on the rotor (see Figure 1-12). Because curves A1 and B2 cross the boundaries, one piece of each of these curves is just above the 0 boundary and another piece is just below the 2π boundary. The small black circles with accompanying coordinate tuples are i-points (not all i-points are labeled). Although the qcs-curves are drawn as straight lines, they can have any shape as long as they are monotonic.

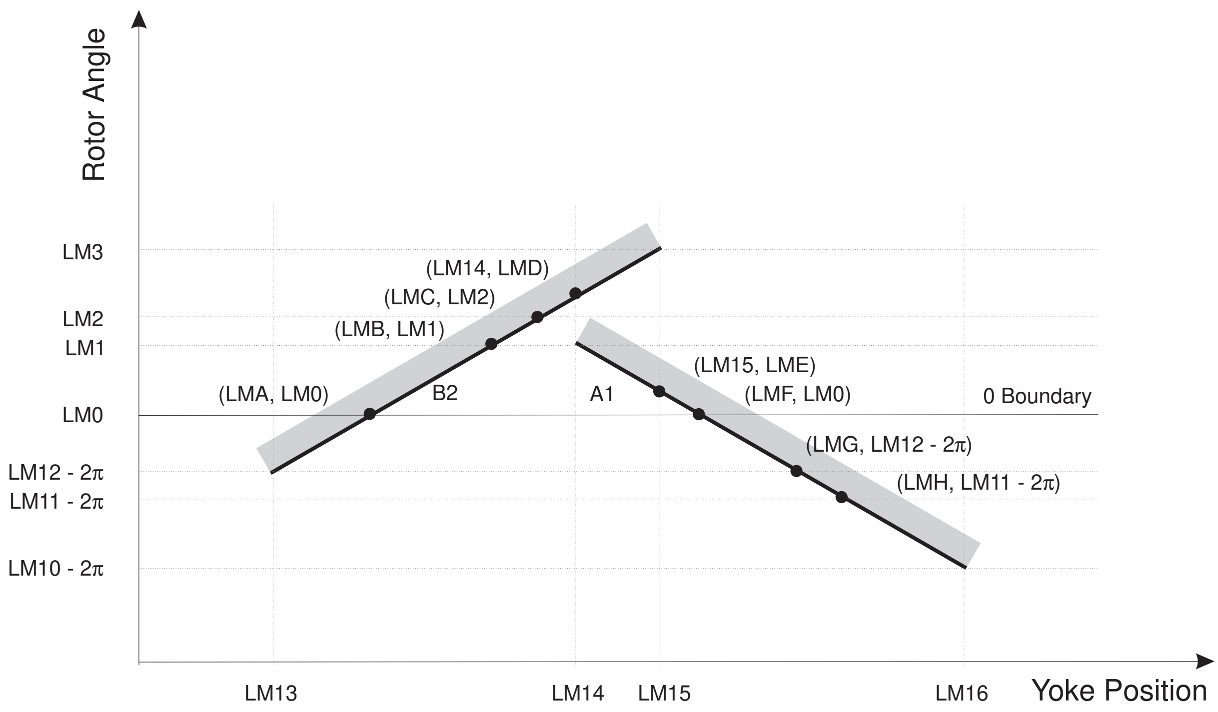


Figure 7-26: A reconstruction of the boundary-crossing qcs-curves from the yoke and rotor qc-space. The pieces of the boundary-crossing curves near the 2π boundary in Figure 7-25 have been moved below the 0 boundary by subtracting 2π from the ordinates of their end points and i-points. The small black circles with accompanying coordinate tuples are i-points. Although the qcs-curves are drawn as straight lines, they can have any shape as long as they are monotonic.

Chapter 8

Related Work

Our task is to use a description of desired behavior to turn a sketch of a mechanical device into multiple families of working geometries. No previous work has addressed this particular task. The work that comes closest to ours is the work in design automation (specifically the approaches based on bond graphs and kinematic building blocks), the work in shape design, and the work in sketch understanding. The first three sections of this chapter describe these three areas of research.

The remaining sections describe supporting work—such as the work in qualitative physics and simulation, and work that indirectly touches on some of the issues involved in our task—such as the work in component connection models and geometric features.

8.1 Design Automation

8.1.1 Bond Graph Approaches

Our techniques can be viewed as a natural complement to the bond graph techniques of the sort developed by Ulrich [53]. (See [30] for a comprehensive discussion of bond graphs.) Our techniques are useful for computing geometry that provides a specified behavior, but because of the inertia-free assumption employed by our simulator, our techniques are effectively blind to energy flow. Bond graph techniques, on the other hand, explicitly represent energy flow but are incapable of representing geometry.

Prabhu and Taylor [41] and Welch and Dixon [54] extend the bond graph design approach to allow specification of positions and orientations of components, but they still do not design the shapes of interacting parts.

Their techniques synthesize a design using an abstract representation of behavior (bond graphs), then use library lookup to map to implementation. We use a similar paradigm, however, because our library contains interacting faces, while theirs includes complete components, we can design interacting geometry, while they cannot. Like our techniques, their techniques produce design variants.

8.1.2 Kinematic Building Blocks

Our techniques focus on the geometry of devices that have time varying engagements (i.e., variable kinematic topology). Therefore, our techniques are complementary to the well know design techniques for fixed topology mechanisms, such as the gear train and linkage design techniques in Erdman and Sandor [10].

Although these well known techniques are not applicable to our problem, they can be used to construct mechanism design tools for fixed topology devices as Rosen et al. [42] demonstrate. They describe a knowledge based tool capable of designing dwell mechanisms based on cams, gears, and linkages.

There has been a lot of recent interest in automating the design of fixed topology devices. A common task is the synthesis of a device that transforms a specified input motion into a specified output motion: Kota and Chiou [31] use a matrix to represent the desired motion transformation, then use matrix decomposition to decompose this into basic building blocks. Subramanian and Wang [52] use iteratively deepening search to compose a sequence of "abstract mechanisms" that achieves the desired motion transformation. The bond graph based design techniques described above are also applicable to this task. All of these approaches are capable of producing design variants, but again, these techniques are not suitable for designing variable topology devices.

8.2 Shape Design

Although our work crosses many research boundaries, it is most closely related to the work in shape design.

Joskowicz and Addanki [25] describe an automated tool for shape design. They start with two interacting shapes (2D profiles) and the desired behavior described as a c-space. They then modify the part shapes by adding and deleting line segments and arcs until the shapes produce the desired c-space. While they take the desired c-space as input, we compute a qc-space that will provide the desired behavior. Their techniques produce a single design instance, rather than a family of designs as SKETCHIT does.

Joskowicz [24] uses a set of local and global operators to simplify and abstract the c-space for a device in order to reduce irrelevant details. For example, one of the operators first divides the c-space boundaries (cs-curves) into monotonic pieces and then replaces those pieces with straight line segments. Another operator eliminates parts of the c-space that cannot be reached because of a particular choice of external inputs to the device. The closest application of these operators to our task is comparing the behavior of two devices: if their simplified and abstracted c-spaces are the same, the two devices provide the same qualitative behavior. Our task is still recognizably different: we want to generate designs that all produce the same qualitative behavior rather than recognizing designs that provide the same behavior.

Caine [2] and Joskowicz and Sacks [27] describe interactive design tools that allow the designer to modify shape by modifying c-space or vice versa. While we use library lookup to map from qc-space to geometry, they use numerical techniques, which can be computationally expensive, to map changes in c-space to changes in geometry. Their techniques produce a single design instance, while our techniques produce multiple families of designs. Finally, their techniques are interactive while ours are automatic.

There is some recent work in exploring the mapping between shape and behavior. Joskowicz et al. [28] use kinematic tolerance space (an extension of c-space) to examine how variations in the shapes of parts affect their kinematic behavior. Their task is to determine how a specified variation in shape affects behavior, ours is to determine what constraints on shape are sufficient to ensure the desired behavior.

Faltings [14] examines how much a single geometric parameter can change, when all others are held constant, without changing the place vocabulary (topology of c-space). Their task is to determine how much a given parameter can change without altering the current behavior, while ours is to determine the constraints on all the parameters sufficient to obtain a desired behavior.

Our task is most similar to that of Faltings and Sun [16]. They describe an interactive design system that modifies a user selected geometric parameter until there is a change in the place vocabulary, and hence a change in behavior. Their system then uses qualitative simulation to determine if the new behavior of the modified geometry matches a specified desired behavior. (They have a language for specifying desired behavior just as we do.) They use the techniques in [14] to determine how much a parameter must change in order to change the place vocabulary.

They modify c-space by modifying geometry, we modify qc-space directly. Because there are many changes in geometry that map to the same change in c-space, their search space is larger than ours. Also, our tool is automatic while theirs is interactive, and we can generate design variants while they cannot.

Gupta and Jakiela [19] describe a novel technique by which a known component "carves out" the shape of an unknown mating component. They require that one of the interacting shapes is known, but we do not. Also, they require a complete description of the desired motion of each component, while we do not.

8.3 Sketch Understanding

There is little previous work in sketch understanding.[1] Narayanan et al. [37] use a diagram of a device to reason about its behavior, but they use a pre-parsed description of the behaviors of each component while we reason directly from the geometry of the interacting faces.

Faltings [13] suggests that a sketch is not a single qualitative model but instead represents a family of precise models. He demonstrates that, taking a sketch as

[1] See [51] for a discussion of common sketching techniques used in engineering.

a qualitative metric digram (i.e., a line drawing with approximate dimensions) it is possible to compute what he calls kinematic topology. Kinematic topology [15] characterizes the topology of the free space regions of the device's c-space. Kinematic topology is an abstraction of place vocabulary [12] and as such, it often contains ambiguities. These ambiguities suggest behaviors that might possibly be obtained by modifying the geometry of the device. However, he does not provide methods for determining which modifications will yield these other behaviors.

8.4 Qualitative Physics

Our work builds upon the a large and growing body of research in the field of qualitative physics. Weld [55] provides a comprehensive overview of the field. Here, to provide background for our qualitative simulation techniques, we discuss a representative sampling of the work in qualitative physics.

de Kleer [6] describes a program that produces causal explanations of the small signal behavior of electric circuits. The program computes behavior by using constraint propagation techniques to propagate the circuit's inputs through the circuit. These techniques are related to the work of Stallman and Sussman [50]. We use a simplified version of these propagation techniques in Section 4.4 to compute the motion of the bodies in a device.

de Kleer [7] describes a qualitative simulator based on confluences, i.e., qualitative differential equations. The behavior of each component in de Kleer's world is characterized by a set of qualitative states (operating regions); the behavior in each qualitative state is described by a set of confluences. The simulator computes all possible consistent qualitative states of the components in a device, and all possible transitions from one set of consistent qualitative states to another. Hence, the simulator computes an envisionment just as ours does.

de Kleer's simulator must be provided with an enumeration of the possible qualitative states (operating regions) of a component. In our domain, however, it is not possible to enumerate the possible qualitative states of a component because a component's behavior depends on the shapes of the other components it interacts with.

Williams [56] describes a simulator that can reason about both the small signal and the large signal behavior of an electric circuit. The simulator computes the small signal model that applies in a particular operating mode, then predicts which parameters change, possibly causing a transition to another operating mode. The simulator then uses constraint analysis to determine which of the possible transitions actually happens first. (Kuipers [33] formalizes these techniques.) This is very similar to the way our simulator computes the next event in Section 4.5: it computes the possible events in each individual qcs-plane, then uses constraint propagation to determine which of these events can happen first.

Forbus [18] views the world from a process centered perspective rather than a device centered perspective. These techniques are suited to modeling processes like boiling which do not involve a fixed collection of "stuff:" as the boiling process evolves, water turns to steam and leaves the system. However, the behavior of the kinds of mechanical devices we are interest in cannot be conveniently decomposed into a set of processes.

8.5 Simulation

SketchIT uses dynamic simulation to compute the behavior of a candidate c-space. Conventional dynamic simulators (e.g., [39], [20], and [49]) predict motion by numerically integrating the equations of motion of the device. (See [4], [36], or [29] for an introductory text on dynamics.) Because we use a qualitative representation (qc-space), we cannot use numerical integration, and hence must use a qualitative simulator.

Faltings [12] describes a qualitative simulator for fixed-axis devices. The simulator is based on a representation called place vocabulary, which is a qualitative version of c-space. Place vocabulary decomposes the free space regions of a cs-plane into "places," regions in which the contact between the pair of components is uniform. There are three kinds of places: two-dimensional regions with no contacts, segments of cs-curves[2] (one contact exists), and intersections between cs-curves (two contact points exists). Place vocabulary also encodes the allowed transitions between the places. Because this representation is qualitative, ambiguities may arise as to which place transitions will actually occur in response to the applied inputs. In this case, the simulator computes all possible transitions.

Faltings' simulator is intended to be a kinematic simulator, and hence has a limited ability to reason about forces.[3] Our simulator, on the other hand, is a dynamic simulator, i.e., a simulator that computes the motion resulting from applied forces.

Forbus et al. [17] extend Faltings' techniques to produce a dynamic simulator. Their simulator models inertia, while we assume that motion is inertia-free. They represent forces as qualitative vectors, we do not. Our representation for forces (see Section 4.2) is designed to eliminate the ambiguity that occurs when summing qualitative force vectors, thereby reducing branching of the simulation.

8.6 Computing Numerical C-Space

We obtain the first candidate qc-space by abstracting the numerical c-space of the sketch. Because, the abstraction process requires only a partial description of the numerical c-space of the sketch, we developed simplified, special purpose techniques

[2] They use the term constraint curve rather than cs-curve.

[3] Kinematics is the study of motion without reference to the forces which cause that motion. Kinematic simulation is commonly used to analyze the motion of devices that have a motion source applied to each degree of freedom.

that compute just the required information. There are many general purpose techniques for computing the complete numerical c-space of a device. Lozano-Pérez [34], for example, describes an algorithm for computing the configuration space of two polygons that translate in the plane without rotation. Brost [1] and Caine [2] describe algorithms for the case in which the polygons rotate as well as translate. More suited to our needs is the work of Joskowicz and Sacks [26]. They compute the full c-space for a fixed-axis device by computing the cs-plane for each pair of interacting parts in the device.

8.7 Component-Connection Models

A common abstraction used in qualitative reasoning about physical systems is the component-connection model: Each component has a set of ports; each port is associated with a parameter such as voltage or fluid flow rate. A set of constraints characterizes the relationships between the parameters of a particular component's ports. Components are connected at their ports; when two ports are connected, they share a common parameter. All of the interesting behavior occurs inside the components; the connections simply propagate parameter values.

By contrast, for the devices we are interested in, all of the interesting behavior occurs at the interaction between two components (i.e., the interaction between a pair of faces). Although the component-connection techniques do not apply directly to the problems in SketchIT's domain, they do address some relevant issues. Therefore, this section describes several examples of work in the area.

Doyle's [9] task is to hypothesize a structure that achieves a set of observable events; ours is to find geometry that achieves a desired behavior. He constructs hypotheses by connecting together primitive mechanisms (components). Each primitive mechanisms has a quantity type associated with its cause (input) and effect (output). Two primitive mechanisms can be connected together only if they have compatible quantity types. This serves as a primary source of constraint for limiting the generation of hypotheses. In our work, the sketch is the primary source of constraint for limiting search: we consider only devices that are similar to the initial sketch.

Falkenhainer and Forbus [11] describe a program that uses a library of model fragments and a description of the structure of a device to construct a model suitable for answering a user query about the device. The goal is to find the simplest model adequate to answer the query. Each of the model fragments describes one possible behavior of a component (i.e., one possible set of constraints relating the parameters at the ports.) The program's task is to determine which of the possible behaviors actually occurs in the context of the overall device.

Nayak [38] describes a similar system. His task is to construct a model that provides a causal explanation of a device's behavior. The expected behavior of the device provides constraint for limiting the search for an adequate model. The expected behavior is described as a desired causal path, for example, "how does the temperature of the thermistor determine the angular deflection of the pointer?"

Mashburn and Anderson [35] extend the methods of Falkenhainer and Forbus to produce a system that guarantees that the model is complete, i.e., that there are enough equations to solve for the desired quantity.

Davis [5] describes a system that performs circuit diagnosis. His task is to determine which components, if malfunctioning, could account for a discrepancy between the observed behavior of a device and the correct behavior. Said differently, the task is to find a model that predicts the observed behavior rather than the correct (intended) behavior.

The primary task of each of these systems is to determine what role each part of the device plays in achieving the overall behavior. During the reverse engineering and generalization process, SketchIT has a similar task of determining what role each part of the device plays in achieving the *desired* overall behavior (see Figure 1-16). Hence, the techniques used in these systems could be used to extend SketchIT's reverse engineering/re-synthesis paradigm to the component-connection domain.

8.8 Reverse Engineering

Before SketchIT can synthesize new designs for a device, it must reverse engineer the original sketch. Shrobe [46] has also examined the task of reverse engineering, but his work is in the domain of linkages. He numerically simulates the kinematics of the linkages using Kramer's TLA [32], then parses the simulation to identify a set of common behaviors such as dwell and frequency doubling. These techniques might be used to extend SketchIT's reverse engineering/re-synthesis paradigm to the linkage domain.

8.9 Geometric Features

In SketchIT's domain, the behavior of a device is determined by how the component shapes interact with each other. However there are other domains in which the geometric features of individual components determine important design properties. Here we consider some of those domains.

For example, Hirschtick [21] describes a knowledge based tool that assists in the design of aluminum extrusions. The tool is rule-based and works in the domain of extrusions (cross-sections) built from lines segments and arcs. The rules trigger off of patterns of geometric features. One rule, for example, states that a thick wall and a thin wall that meet at a corner should have a fillet. The program begins by identifying the important features of the extrusion: walls, corners, hollow cavities, fillets, etc., then applies the rule-base to produce manufacturing advice.

Dixon et al. [8] describe a similar manufacturing advisor. However, they use a feature based geometric design tool to construct a geometric model of the device, rather than recognizing features in a conventional CAD model of the device. Their domain is aluminum extrusions, castings, and injection molded plastic.

Wolter and Chandrasekaran [57] describe a feature-oriented design system capable of representing a wide range of functions. They represent geometry with a hierarchy of structures called geomes. A geome is a collection of geometric elements with constraints on how those elements are combined. They label a geome with the function it is commonly used for. For example, a geome consisting of a cylindrical rod inside a round hole of the same diameter would be labeled a pivot. The designer constructs a design by assembling geomes that provide the desired functions.

They also provide abstract geomes, geomes that have no implementation, allowing the designer to describe the intended function of a device rather than the structure. The revolute-constraint geome, for example, is a pivot with no implementation. The designer can later refine the abstract geomes to more specific geomes that do have geometric implementations.

Their functional language can represent static features (e.g., a slot) or constant contact (e.g., a pin in a hole or a rack and pinion). They cannot handle functions that require intermittent contact such as a the lever and hook in the circuit breaker.

8.10 Representing Function

During reverse engineering, SKETCHIT determines what role each part of the device plays in achieving the desired overall behavior. Other researchers have examined representations that allow a designer to explicitly represent the intended function of the parts of a device. They reason from symbolic assertions about behavior while we reason directly from the geometry of the device.

Hodges [22], for example, describes a representation for capturing the behavior and function of the parts of a device. This representation is used to determine when a device may be useful for a task for which it was not originally intended. Iwasaki et al. [23] describe a language for representing the function of the parts of a device. Their goal is to explicitly record how the device is supposed to work in addition to recording the device's structure.

8.11 Algebraic Constraints

SKETCHIT produces output in the form of a BEP-Model, a parametric model with constraints that ensure the desired behavior. Serrano and Gossard [44] describe a system called MATHPAK that is suitable for solving the constraints of a BEP-Model.[4] Similarly, Serrano [45] describes a system for efficiently solving systems of algebraic

[4] DesignView, the system we actually use to solve the constraints of a BEP-Model, is based on MATHPAK.

constraints like those contained in a BEP-Model.

Another approach to solving the constraints of a BEP-Model is to use a numerical optimizer. This would find a solution to the constraints that also maximizes some objective function. Orelup et al. [40] describe one example of an optimizer that could be used.

Chapter 9

Discussion

9.1 Contributions

This work produced four main results:

- We produced a computer program that can turn a sketch of a mechanical device into working geometry by using a description of desired behavior as a guide.
- We built a computer program that can speak the engineer's natural language, sketches.
- This computer program can also produce design variants.
- We created qualitative configuration space as a particularly powerful representation for these tasks.

On the way to achieving these main results, we solved several important subproblems, including:

- We developed a library of parameterized faces with constraints that ensure the faces implement monotonic interactions.
- We developed efficient qualitative simulation techniques.

9.1.1 Turning a Sketch into Working Geometry.

We demonstrated that a computer program, using a description of the desired behavior, can turn a sketch of a mechanical device into working geometry. We did this by developing computational techniques to perform this task, and testing them with an implemented computer program called SketchIT.

To our knowledge, no previous work has addressed the task of automatically turning sketches into working geometry.

To perform this task, SKETCHIT uses a paradigm of reverse engineering followed by synthesis. During reverse engineering, SKETCHIT identifies what role each part of the sketch plays in achieving the desired overall behavior. During synthesis, SKETCHIT generates geometry that implements the identified role of each part.

SKETCHIT works in the domain of fixed-axis devices whose behavior is well approximated with quasi-statics. We chose this domain because it is a large and useful class of mechanical devices, and because it is sufficiently tractable to facilitate our progress. Although our work focused on a particular class of mechanisms, the reverse–engineering–and–synthesis paradigm is applicable to a much larger class of design problems. For example, the paradigm could be applied to designs that rely on friction and inertia to achieve their behavior (see Section 9.2.1).

9.1.2 Speaking the Engineer's Language

When engineers communicate with each other, they frequently use sketches because they are a convenient way to both record and communicate design information. By creating a computer program that can interpret a sketch and transform it into working geometry, we have moved one step closer to CAD tools that speak the language that engineers speak to one another.

Traditional design tools are often difficult to use because they require the designer to use languages optimized for the program's convenience rather than the engineer's (e.g., finite element analysis tools require the engineer to describe a device as a mesh). As a result, these tools often go unused until the later stages of design where they are used for analysis and documentation. A CAD tool that speaks the engineer's natural language would likely be used throughout the entire design process, in much the same way one would use a human assistant.

9.1.3 Design Variants

SKETCHIT produces two kinds of design variants: It produces the first kind by replacing rotating parts with translating ones and vice versa. It produces the second kind by selecting alternative implementations for the pairs of interacting faces in the device. We have demonstrated that the designs SKETCHIT produces include novel designs.

SKETCHIT represents each of its designs with a BEP-Model, a parametric model augmented with constraints that ensure the device produces the desired behavior. The constraints in each BEP-Model represent the regions in parameter space (i.e., the space spanned by the geometric parameters) that provide the desired behavior. Thus they define a family of design solutions that the designer can explore in order to satisfy other design requirements, such as requirements on size or cost. We demonstrated that the BEP-Model's family of designs includes a wide variety of design solutions.

Because the BEP-Model's constraints prevent changes to the geometry that are inconsistent with the desired behavior, the BEP-Model makes it easy for the designer to explore the design space.

During conceptual design a high premium is placed on examining as many design alternatives as possible. By producing many alternative designs, SketchIT has the potential to greatly improve the conceptual design process.

9.1.4 QC-Space: As a Representation

SketchIT's primary tasks are to turn a sketch into working geometry and to produce alternative implementations. SketchIT uses the problem solving paradigm in Figure 9-1 to perform these tasks. This paradigm is essentially abstraction followed by re-synthesis. SketchIT first performs abstraction, instead of going directly from the original design to new designs, because an abstract representation simplifies the reasoning process: A sketch contains an enormous amount of geometric detail, much of which is irrelevant to the behavior. The abstraction process strips away the irrelevant detail to expose what is essential.

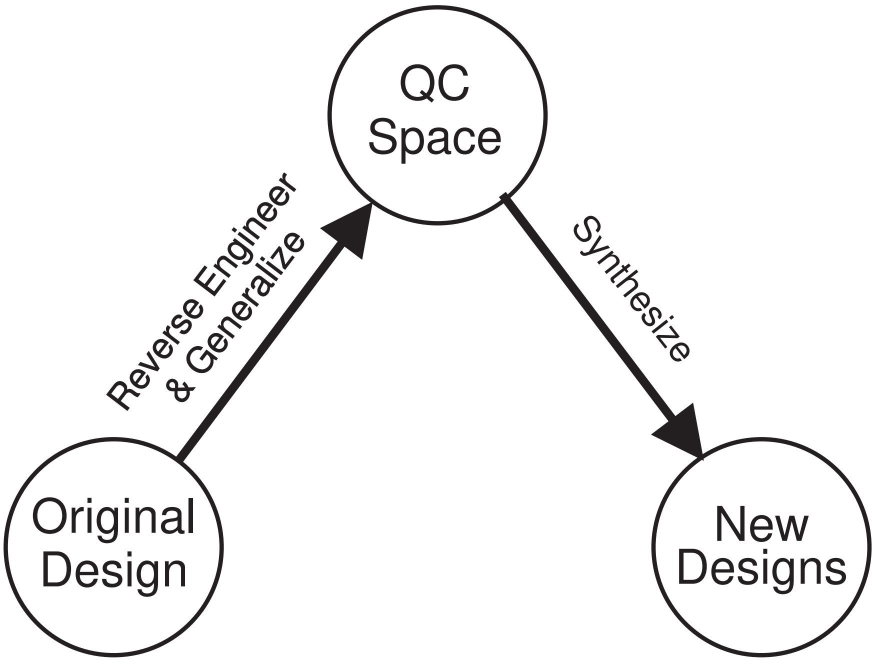


Figure 9-1: SketchIT's problem solving paradigm.

The question is, what is the right abstraction? For mechanical devices, the behavior is achieved through interactions between component shapes. Hence, it is not the shapes themselves that are essential, but rather the interactions between them. We suggest therefore, that the right abstraction is one that captures interactions but strips away their implementations (i.e., strips away individual component shapes). Qc-space provides precisely this kind of abstraction.

Because qc-space explicitly represents behavior, it is an efficient representation for mechanical design. Given the intimate connection between shape and behavior, design of mechanical artifacts is typically conceived of as the modification of shape to achieve behavior. But if changes in shape are attempts to change behavior, and if the mapping between shape and behavior is quite complex [2], then, we suggest, why not manipulate a representation of behavior? Our qc-space is just such a representation. We suggest that it is complete and yet offers a far smaller search space. It is complete because any change in shape will produce a c-space that maps to one of the qc-spaces that our program will consider. Qc-space is a far smaller search space because a single change to qc-space maps to many changes in shape.

All of this would be for naught if the mapping from qc-space back to implementation were difficult. However, this turns out to be a relatively simple problem: Implementations for interactions can be cataloged; our interaction library is an example. By constructing catalogues with multiple implementations for each kind of interaction, it is possible to exploit this mapping as an opportunity to produce design variants, as indeed we do.

Our system is an existence proof of the value of qc-space as a representation. We used qc-space to construct an implemented computer program, and that program successfully turned sketches into working geometry and produced design variants.

9.1.5 The Library

Using libraries in design tools is not a new idea; the novelty here is in cataloging mechanical interactions. Conventional design libraries contain either complete components or small assemblies of components (e.g., [52]) that are concatenated to produce complete designs. Our library, on the other hand, contains implementations for mechanical interactions. Because interactions are at a finer level of granularity than components or assemblies, our library provides much greater expressive power than does a conventional library.

9.1.6 Efficient Simulation

Our simulator employs several innovative techniques that reduce ambiguity in force balances. By reducing ambiguity, we reduce the amount of search (branching) required to simulate a device, and hence reduce the amount of computation required.

We represent forces by their projections onto the degrees of freedom of the bodies to which they are applied. We can make this simplification without introducing inaccuracies because the projections are the only components of a force that affect the motion of a body. The advantage is that by simplifying forces to their projections, we greatly reduce the ambiguity in force balances.

Another way we reduce ambiguity is by reasoning about the type of constraint—motion constrained or compliant—that an engagement applies. One of our basic principles is that motion constrained engagements overpower compliant ones. For example, the resultant of a positive motion constrained engagement force and a negative compliant engagement force is always a positive force. Without our basic principle, this force balance would be ambiguous.

9.2 Future Work

9.2.1 Extensions

SketchIT is a prototype software system and as such requires some additional work before it is complete. The simulator, described in Chapter 4, is the part of the system that requires the bulk of the continued effort. The required extensions needed to complete the simulator are described throughout that chapter.

Although the interaction library, described in Chapter 7 and Appendix A, contains at least one implementation for each of the different kinds of qcs-curves, it is by no means complete. Our library contains implementations that use only flat faces[1] and hence we must consider other types of implementations, such as curved faces. In addition, our use of flat faces is not exhaustive: for most pairs of faces we do not enumerate all the regions in the parameter space (i.e., the space spanned by the geometric parameters of the pair of faces) for which the faces implement a given kind of qcs-curve.

We generated our initial library by hand. We would like to develop computational approaches to complete the library. One approach would be to construct a general expression for the cs-curve of a pair of faces and then to apply symbolic algebra techniques to identify the constraints necessary for this general curve to be monotonic with the desired qualitative slope.

In its current implementation, SketchIT uses search to reverse engineer the sketch: the program systematically generates candidate qc-spaces for the sketch and tests them to see if they provide the desired behavior. We believe that we can reverse engineer the sketch much more efficiently using debugging rules that examine why a particular qc-space fails to produce the desired behavior (see Section 3.3.2). By diagnosing these kinds of failures, we could produce new candidate qc-spaces by judicious repair of the current one.

We currently use state transition diagrams to specify the desired behavior of a device in terms of a desired sequence of engagements. We would like to develop a behavior specification language that is more like the verbal language that engineers use to describe the behavior of devices. For example, we would like to specify the desired behavior using common engineering terms like "ratchet," "clutch", or "trip mech-

[1] Circular faces are used when rotors act as stops.

anism." We believe that the state transition diagram language is a good substrate upon which to implement this better language. For example, many engineering terms, such as "ratchet," have a direct translation into a desired sequence of engagements, and hence are simply a macro on top of the state transition diagram language.

SKETCHIT's domain is fixed-axis devices for which engagements are frictionless and inertia is negligible. While this is a useful class of devices (see Sacks and Joskowicz [43]), we would like to relax some of these restrictions in order to enlarge the class of devices SKETCHIT can handle. For example, we would like to relax the restriction to frictionless engagements.

Relaxing this restriction would require modifications to both the simulator and the synthesis modules that generate the BEP-Models. A simple model of friction is that it either dominates the force balance or is negligible. Hence, when computing force balances, the simulator would have an additional property to branch on: whether or not the friction from an engagement dominates the other forces applied to a body. Once SKETCHIT determines which engagements should provide high friction and which should not, it must add appropriate constraints to the BEP-Model.

These constraints can be expressed in terms of the friction cone, which defines the range of forces (i.e., the range of orientations of a force) that a surface can resist without slipping. The size of the friction cone is a function of the coefficient of friction and the geometric parameters of the interacting faces. To ensure that an engagement provides high friction, SKETCHIT must add constraints to the BEP-Model that ensure the engagement forces are inside the friction cone. Conversely, to ensure that the friction is negligible, the program must add constraints that ensure the engagement forces are outside the friction cone.

Another restriction we would like to relax is the restriction to fixed-axis devices. We have identified a commonly occurring class of devices in which a pair of parts has three degrees of freedom (instead of two degrees of freedom as in a fixed-axis device) but the configuration space is still tractable. These devices have switchable degrees of freedom: for each different mode of operation one of the degrees of freedom is switched off so that at any given time, at most two degrees of freedom are active. SKETCHIT could represent these kinds of devices with a set of qcs-planes, one for each mode of operation. The simulator would select the appropriate qcs-plane for each step of simulation.

Figure 9-2 shows an example consisting of two rotors, labeled A and B. Rotor A has a spring-loaded plunger protruding from it, B has a spring pushing it toward a stop. Devices similar to this are used, for example, to index the film advance in cameras. In the first mode of operation, the plunger's degree of freedom is turned off. As rotor A turns clockwise it pushes rotor B as shown in Figure 9-3. Eventually the rotors disengage and B's spring pushes B against its stop. In the second mode of operation, rotor B's degree of freedom is turned off and the plunger's degree of freedom is turned back on: Rotor A turns counterclockwise causing the plunger to be depressed by engaging rotor B as shown in Figure 9-4. In normal use, the device would alternate between these two modes.

Figure 9-3 shows the qc-space for the first mode. The qc-space is a plane defined by the degrees of freedom of the rotors. Because the plunger's degree of freedom is turned off in this mode, its degree of freedom does not appear in the qc-space. Figure 9-4 shows the qc-space for the second mode. This qc-space is also a plane, but this time it is defined by the degrees of freedom of rotor A and the plunger. At any point in the simulation, one or the other of these qcs-planes will be active. The simulator's new task is to select the applicable qcs-plane for each step of the simulation.

After making these extensions to our system, the next task is to determine how well these techniques scale to design problems more complex than the three working examples reported here.

We are now beginning to explore how our techniques can be applied to other problem domains. For example, we believe that the BEP-Model will be useful for kinematic tolerance analysis (see [3] for an overview of tolerancing). Here the task is to determine if a given set of variations in the shapes and locations of the parts of a device will compromise the desired behavior. A possible approach to this task is to determine if the variations are contained in the family of working designs defined by the BEP-Model. For example, a simplistic implementation would use Monte Carlo simulation to determine if a large number of designs randomly selected from the specified set all satisfy the constraints of the BEP-Model.

We have also begun to explore design rationale capture. We believe that the constraints of the BEP-Model will be a useful form of design documentation, serving as a link between the geometry and the desired behavior. The constraints might, for example, be used to prevent subsequent redesign efforts from modifying the geometry in a way that compromises hard won design features in the original design.

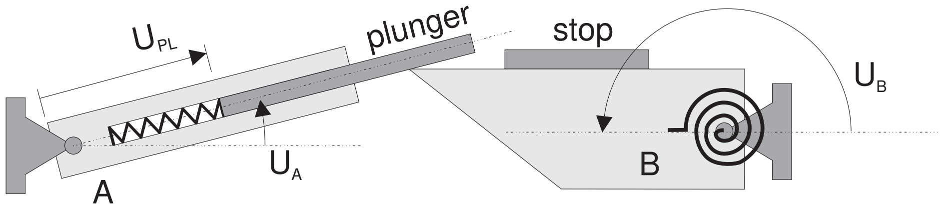


Figure 9-2: A device with switchable degrees of freedom. The device employs two rotors, labeled A and B. Rotor A has a spring-loaded plunger protruding from it, B has a spring pushing it toward a stop.

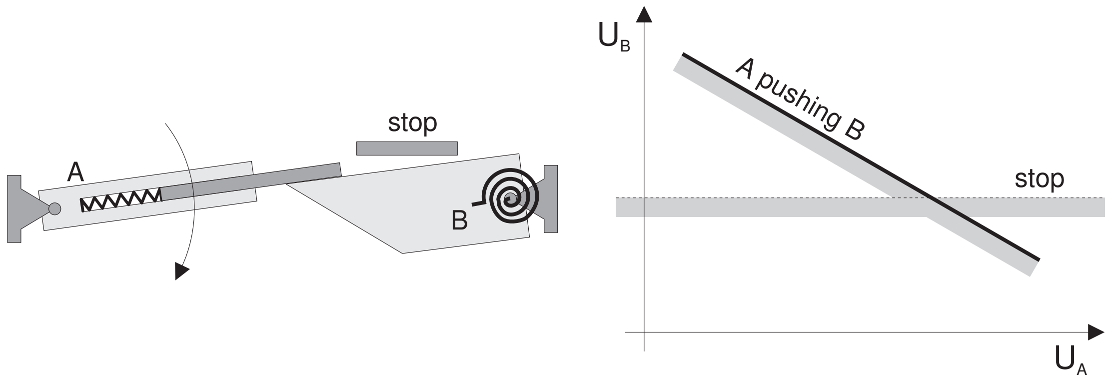


Figure 9-3: The first mode of operation and the corresponding qc-space. The plunger's degree of freedom is switched off.

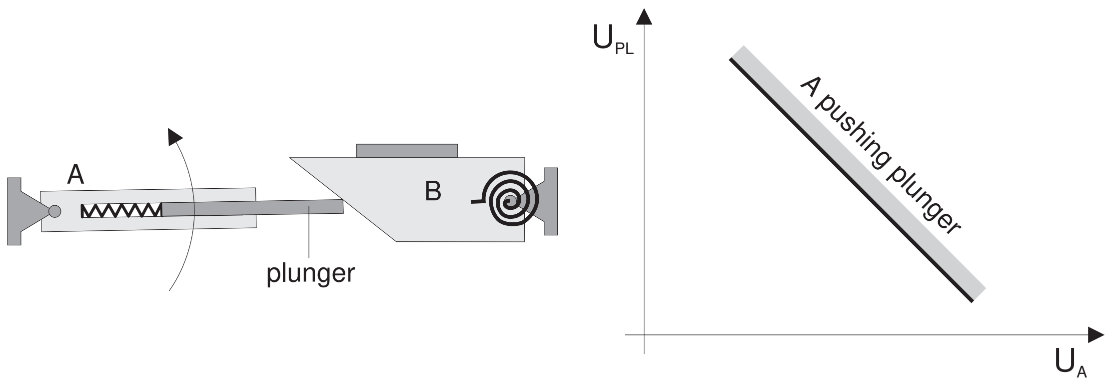


Figure 9-4: The second mode of operation and the corresponding qc-space. Rotor B's degree of freedom is switched off.

9.2.2 The Vision

The ultimate goal of this project was to develop techniques that allow a computer program to read, understand, and use sketches of mechanical devices. The end result is a computer program that can transform a sketch into working geometry and produce alternative implementations. From a broader perspective, this project is a small step toward a much larger vision of getting the knowledge of the engineer into the computer.

For the most part, computers are currently expediters of bookkeeping. Their primary use in design is for documentation and analysis. We look to a future in which computers take a more active role in the design process, performing the same kinds of tasks that a human design assistant might. For this future to become a reality, computer programs must be able to understand the language that designers use to communicate design information, and the programs must contain knowledge about the physical artifacts being designed. For SKETCHIT to perform its tasks, for example, it must be able to read a sketch—the language that engineers use to describe mechanical devices—as well as use knowledge about mechanical devices: knowledge of the laws of motion and knowledge in the form of a library of interactions.

This future will be the third revolution in the use of computers in design. The first revolution was the electronic drafting board, the second was the computer as a physical simulator, and the third will be getting the knowledge of the engineer into the computer.

Appendix A

Interaction Library

This appendix contains our complete library of interactions. Chapter 7 describes how we derive and use the library.

The library contains implementations for every possible type of qcs-curve. Figure A-1 shows all the possible types of finite qcs-curves. Because there are eight types, and the coordinates q_1 and q_2 can be either rotation or translation, the library must be capable of implementing 32 different types of interactions. The current library has one implementation for each of these 32 types.[1]

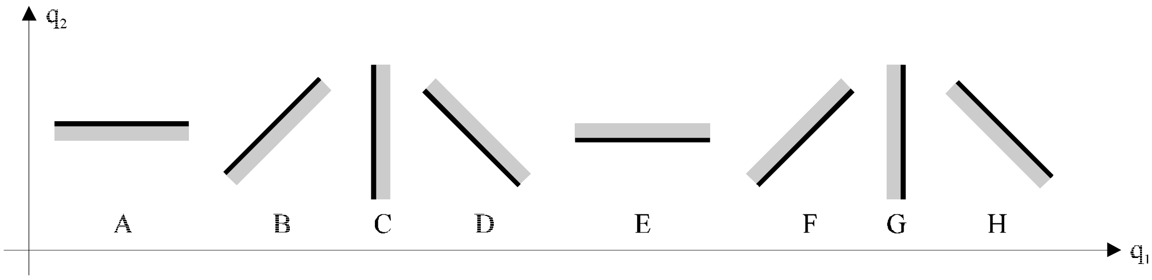


Figure A-1: The eight types of finite qcs-curves. For drawing convenience, the diagonal qcs-curves are shown as straight line segments; they can have any shape as long as they are monotonic.

The interaction between a rotating or translating face and a fixed face produces either a horizontal or vertical, infinite qcs-curve. These curves are infinite versions of curves A, C, E, and G in Figure A-1. Because there are four types of infinite qcs-curves, and one of the interacting bodies can either rotate or translate while the other body is fixed, the library must include implementations for 8 additional types

[1] There are two implementations for diagonal curves for which one body translates and the other rotates.

of interactions. Hence, the complete library includes a total of 40 interactions: 32 for finite qcs-curves and 8 for infinite ones.

By appropriate use of coordinate transformations, we can implement the 40 different kinds of interactions with just 10 different kinds of implementations. Table 7.1 shows the 10 distinct cases that remain after using coordinate transformations. Here we provide implementations for just the 10 distinct cases; Chapter 7 describes how to apply coordinate transformations to obtain the others.

A.1 Pushing: Rotation & Translation

Implementations for:

- interaction: pushing, rotation & translation
- curve length: finite
- curve type: diagonal
- number of implementations: 2

A.1.1 Cam and Offset Follower

The geometry in Figure A-2 can be used to implement a pushing interaction between a rotating face and a translating face. This particular implementation is called a cam and offset follower. The hallmark of this implementation is that the translating face cannot pass through the pivot of the rotating face (Section A.1.2 shows a different implementation in which the translating face can pass through the pivot). This geometry can be used to implement any diagonal qcs-curve for which one body rotates and the other translates. Here we show how it is used to implement qcs-curves with qualitative slope H. Section 7.2.1 shows how to implement the other three diagonal slopes by appropriate use of coordinate transformations.

CONSTRAINTS

For the cs-curve to be monotonic and have qualitative slope H, the geometry must satisfy the following constraints:

$$w > 0 \tag{A.1}$$

$$L > 0 \tag{A.2}$$

$$h > 0 \tag{A.3}$$

$$r = (s^2 + L^2 - 2sL\cos(\phi))^{1/2} \tag{A.4}$$

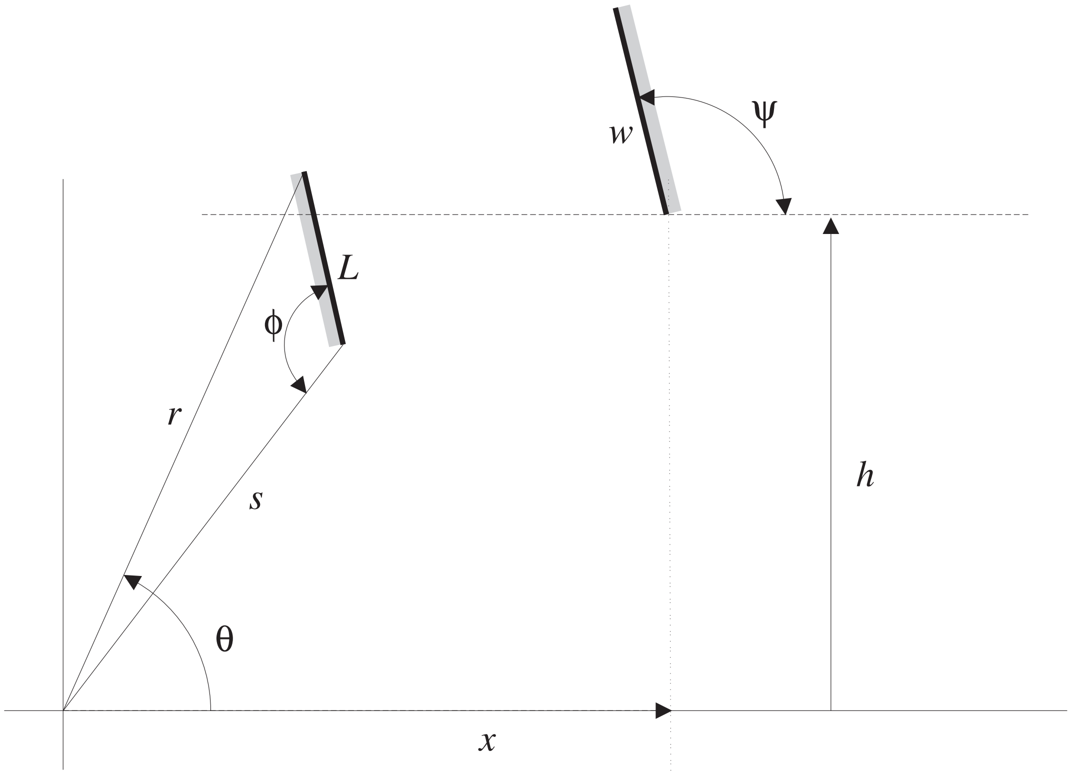


Figure A-2: The parameterization of a pair of faces used to implement pushing interactions between rotating and translating faces. The rotating face rotates about the origin, its position measured positive counterclockwise with angle θ. The translating face translates horizontally, its position measured positive to the right with x. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

$$r > h \tag{A.5}$$

$$s < h \tag{A.6}$$

$$\phi > \pi/2 \tag{A.7}$$

$$\arccos(\frac{L^2 + r^2 - s^2}{2Lr}) + \arccos(h/r) < \pi/2 \tag{A.8}$$

$$\phi \leq \pi \tag{A.9}$$

$$\psi < \arcsin(h/r) + \pi/2 \tag{A.10}$$

$$\psi > 0 \tag{A.11}$$

END POINTS

Figure A-3 shows the qcs-curve implemented by the geometry in Figure A-2. The coordinates of the end points of the curve are:

$$\theta_1 = \arcsin(h/r) \tag{A.12}$$

$$x_1 = r\cos(\theta_1) \tag{A.13}$$

$$\theta_2 = \pi - \arcsin(h/r) \tag{A.14}$$

$$x_2 = -r\cos(\theta_1) \tag{A.15}$$

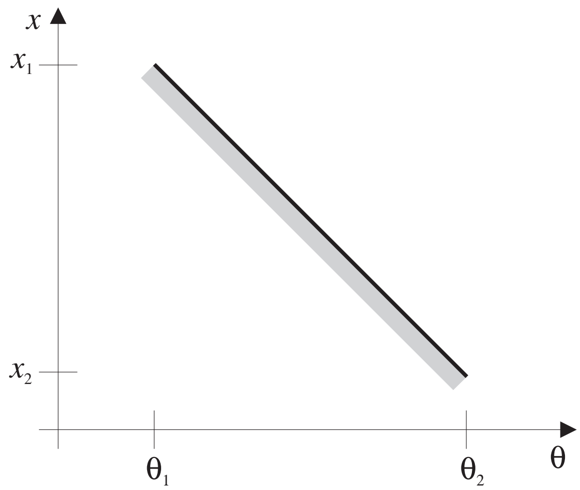


Figure A-3: The qcs-curve implemented by the geometries in Figures A-2 and A-4. For drawing convenience, the qcs-curve is shown as a straight line segment; it can have any shape as long as it is monotonic.

A.1.2 Cam and Centered Follower

The geometry shown in Figure A-4 can be used to implement a pushing interaction between a rotating face and a translating face. This particular implementation is called a cam and centered follower. The hallmark of this implementation is that the translating face can pass through the pivot of the rotating face. This geometry can be used to implement any diagonal qcs-curve for which one body rotates and the other translates. Here we show how it is used to implement qcs-curves with qualitative slope H. Section 7.2.1 shows the kinds of coordinate transformations needed to implement the other three diagonal slopes.

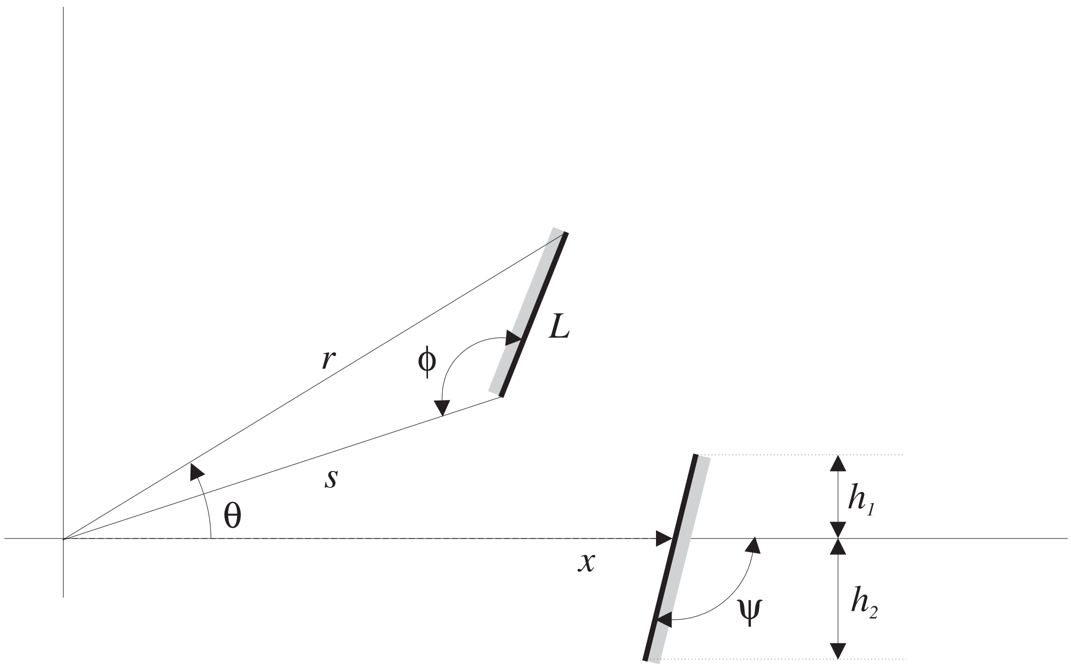


Figure A-4: The parameterization of a pair of faces used to implement pushing interactions between rotating and translating faces. The rotating face rotates about the origin, its position measured positive counterclockwise with angle θ. The translating face translates horizontally, its position measured positive to the right with x. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

CONSTRAINTS

For the cs-curve to be monotonic and have qualitative slope H, the geometry must satisfy the following constraints:

$$L > 0 \tag{A.16}$$

$$h_1 > 0 \tag{A.17}$$

$$h_2 > 0 \tag{A.18}$$

$$\psi > \pi/2 \tag{A.19}$$

$$\psi < \pi \tag{A.20}$$

$$\phi > \pi/2 \tag{A.21}$$

$$\phi < \pi \tag{A.22}$$

$$r = (s^2 + L^2 - 2sL\cos(\phi))^{1/2} \tag{A.23}$$

$$s > h_1 \tag{A.24}$$

$$0 > r/\tan(\psi) + h_2/\sin(\psi) \tag{A.25}$$

END POINTS

Figure A-3 shows the qcs-curve implemented by the geometry in Figure A-4. The coordinates of the end points of the curve are:

$$\theta_1 = -\arcsin(h_2/r) \tag{A.26}$$

$$x_1 = r\cos(\theta_1) - h_2/\tan(\psi) \tag{A.27}$$

$$\theta_2 = \arcsin(h_1/s) + \arccos(\frac{s^2 + r^2 - L^2}{2sr}) \tag{A.28}$$

$$x_2 = s\cos(\arcsin(h_1/s)) + h_1/\tan(\psi) \tag{A.29}$$

A.2 Pushing: Rotation & Rotation I

Implementations for:

- interaction: pushing, rotation & rotation
- curve length: finite
- curve type: diagonal types D and H

The geometry in Figure A-5 can be used to implement diagonal qcs-curves with negative slope for which both bodies rotate (i.e., types D and H). Here we show how it is used to implement qcs-curves with qualitative slope H. To implement curves with slope D, the geometry must be flipped about the horizontal line connecting the pivots (see for example, Figure 7-6). When the geometry is flipped over, the rotation angles are still measured positive counterclockwise.

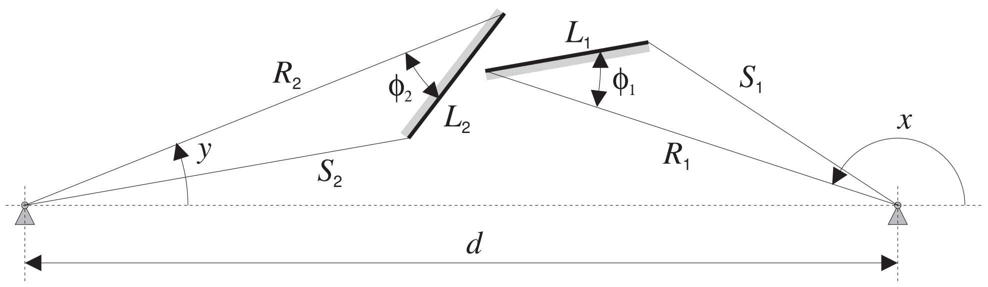


Figure A-5: The parameterization of a pair of faces used to implement pushing interactions between rotating faces for the case in which the qcs-curve has negative slope. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

CONSTRAINTS

For the cs-curve to be monotonic and have qualitative slope H, the geometry must satisfy the following constraints:

$$\phi_1, \phi_2, R_1, R_2, L_1, L_2 > 0 \tag{A.30}$$

$$S_1 = (R_1^2 + L_1^2 - 2R_1 L_1 \cos(\phi_1))^{1/2} \tag{A.31}$$

$$S_2 = (R_2^2 + L_2^2 - 2R_2 L_2 \cos(\phi_2))^{1/2} \tag{A.32}$$

$$d - S_2 > R_1 \tag{A.33}$$

$$R_1 > d - R_2 \tag{A.34}$$

$$d - S_1 > R_2 \tag{A.35}$$

$$R_2 > d - R_1 \tag{A.36}$$

$$L_1 < R_1 \cos(\phi_1) \tag{A.37}$$

$$L_2 < R_2 \cos(\phi_2) \tag{A.38}$$

$$e_1 = \arccos\left[\frac{d^2 + R_1^2 - R_2^2}{2dR_1}\right] \tag{A.39}$$

$$e_2 = \arccos\left[\frac{d^2 + R_2^2 - R_1^2}{2dR_2}\right] \tag{A.40}$$

$$e_1 < \pi/2 - e_2 - \phi_2 \tag{A.41}$$

$$e_2 < \pi/2 - e_1 - \phi_1 \tag{A.42}$$

END POINTS

Figure A-6 shows the qcs-curve implemented by the geometry in Figure A-5. The coordinates of the end points of the curve are:

$$x_1 = \pi - e_1 \tag{A.43}$$

$$y_1 = e_2 \tag{A.44}$$

$$x_2 = \pi + e_1 \tag{A.45}$$

$$y_2 = -e_2 \tag{A.46}$$

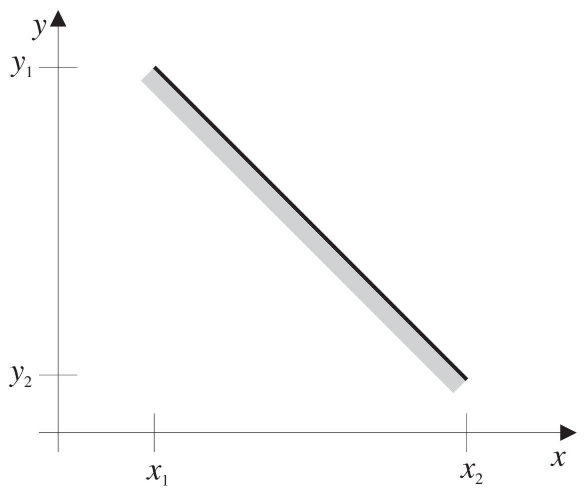


Figure A-6: The qcs-curve implemented by the geometries in Figures A-5 and A-9. For drawing convenience, the qcs-curve is shown as a straight line segment; it can have any shape as long as it is monotonic.

A.3 Pushing: Rotation & Rotation II

Implementations for:

- interaction: pushing, rotation & rotation
- curve length: finite
- curve type: diagonal types B and F

The geometry in Figure A-7 can be used to implement diagonal qcs-curves with positive slope for which both bodies rotate (i.e., types B and F). Here we show how it is used to implement qcs-curves with qualitative slope B. To implement curves with slope F, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside).

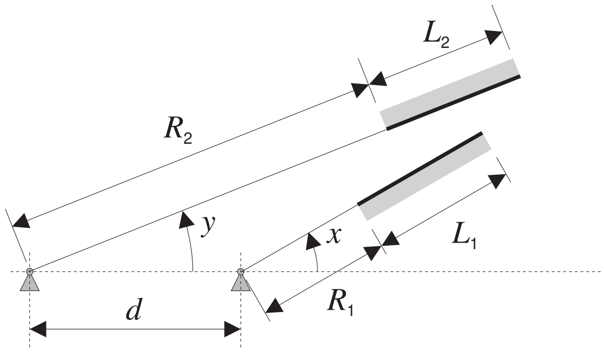


Figure A-7: The parameterization of a pair of faces used to implement pushing interactions between rotating faces for the case in which the qcs-curve has positive slope. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

CONSTRAINTS

For the cs-curve to be monotonic and have qualitative slope B, the geometry must satisfy the following constraints:

$$R_1, R_2, L_1, L_2, d > 0 \tag{A.47}$$

$$R_2 > d + R_1 \tag{A.48}$$

$$R_2 < d + R_1 + L_1 \tag{A.49}$$

$$R_2 + L_2 > d + R_1 + L_1 \tag{A.50}$$

END POINTS

Figure A-8 shows the qcs-curve implemented by the geometry in Figure A-7. The coordinates of the end points of the curve are:

$$x_1 = -\arcsin\left(\frac{R_2 \sin(-y_1)}{R_1 + L_1}\right) \tag{A.51}$$

$$y_1 = -\arccos\left(\frac{R_2^2 + d^2 - (R_1 + L_1)^2}{2R_2 d}\right) \tag{A.52}$$

$$x_2 = -x_1 \tag{A.53}$$

$$y_2 = -y_2 \tag{A.54}$$

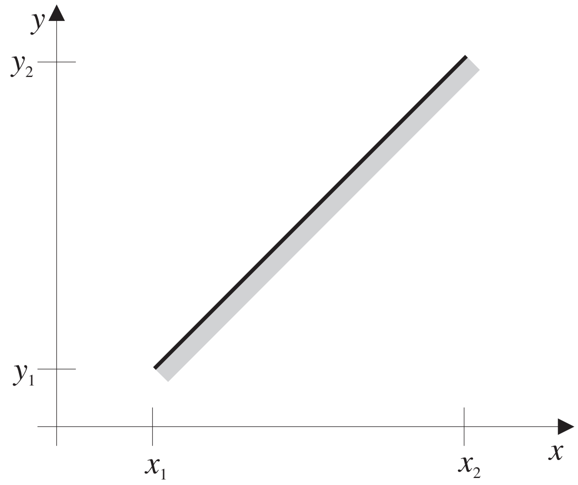


Figure A-8: The qcs-curve implemented by the geometry in Figure A-7. For drawing convenience, the qcs-curve is shown as a straight line segment; it can have any shape as long as it is monotonic.

A.4 Pushing: Translation & Translation

Implementations for:

- interaction: pushing, translation & translation
- curve length: finite

- curve type: diagonal

The geometry in Figure A-9 can be used to implement pushing interactions between translating faces. It can implement any diagonal qcs-curve for which both bodies translate. Here we show how it is used to implement qcs-curves with qualitative slope H. The other three kinds of diagonal curves are implemented by flipping the directions of positive displacement.

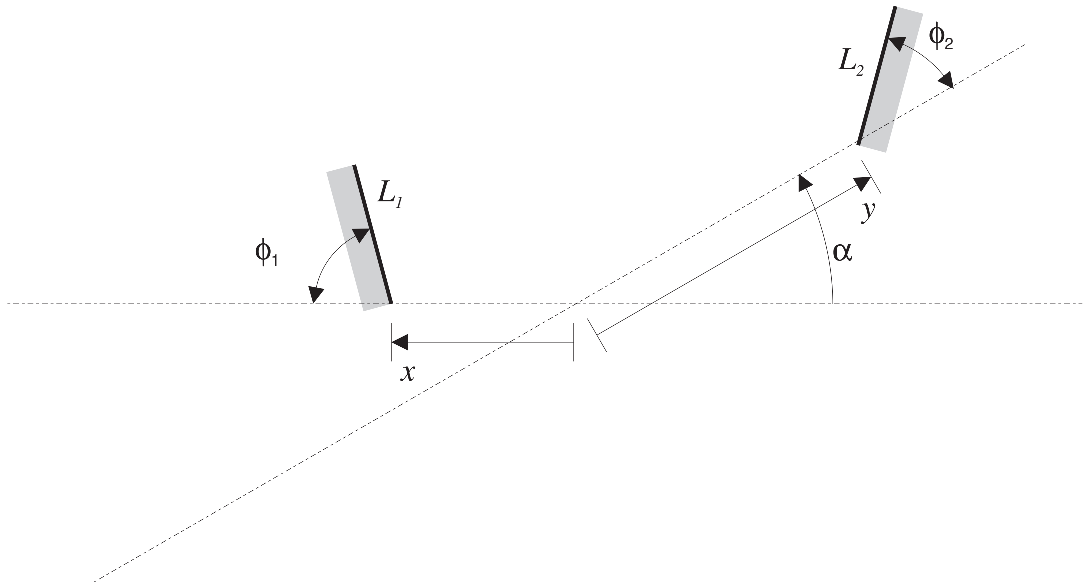


Figure A-9: The parameterization of a pair of faces used to implement pushing interactions between translating faces. Shading indicates the back sides of the faces, i.e., the sides that are inside the solids containing the faces.

CONSTRAINTS

For the cs-curve to be monotonic and have qualitative slope H, the geometry must satisfy the following constraints:

$$L_1, L_2 > 0 \tag{A.55}$$

$$0 < \alpha < \pi \tag{A.56}$$

$$0 < \phi_1 < \pi \tag{A.57}$$

$$0 < \phi_2 < \pi \tag{A.58}$$

$$0 < \alpha + \phi_1 < \pi \tag{A.59}$$

$$0 < \alpha + \phi_2 < \pi \tag{A.60}$$

Figure A-6 shows the qcs-curve implemented by the geometry in Figure A-9. The coordinates of the end points of the curve are:

$$x_1 = -L_1 \cos(\phi_1) - L_1 \frac{\sin(\phi_1)}{\tan(\alpha)} \tag{A.61}$$

$$y_1 = L_1 \frac{\sin(\phi_1)}{\sin(\alpha)} \tag{A.62}$$

$$x_2 = L_2 \frac{\sin(\phi_2)}{\sin(\alpha)} \tag{A.63}$$

$$y_2 = -L_2 \cos(\phi_2) - L_2 \frac{\sin(\phi_2)}{\tan(\alpha)} \tag{A.64}$$

A.5 Fixed Rotor-Stop

Implementations for:

- interaction: rotation stopped by stationary face
- curve length: infinite
- curve type: horizontal or vertical

The geometry in Figure A-10 can be used to implement a fixed rotor-stop. Here we show how it is used to implement infinite, vertical curves of type G. To implement infinite, vertical curves of type C, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Horizontal curves are implemented by defining the degree of freedom of the rotor to be the ordinate of the cs-plane rather than the abscissa.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is G, the geometry must satisfy the following constraints:

$$R, L > 0 \tag{A.65}$$

$$\phi > -\pi/2 \tag{A.66}$$

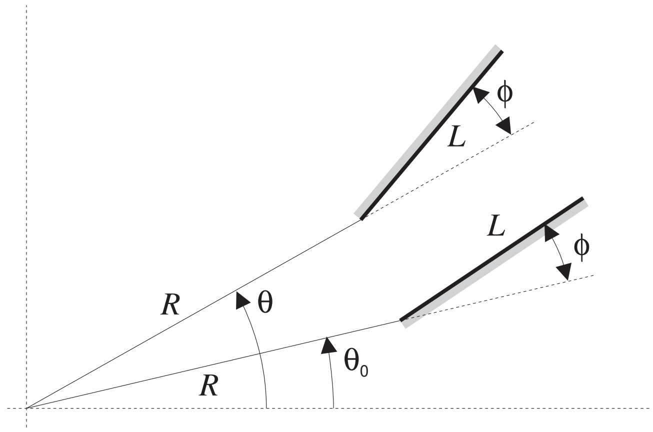


Figure A-10: Geometry used to implement a fixed rotor-stop. The position of the rotor is measured positive counterclockwise with angle θ. The fixed face is located at angle θ_0.

$$\phi < \pi/2 \tag{A.67}$$

END POINTS

Figure A-11 shows the qcs-curve implemented by the geometry in Figure A-10.

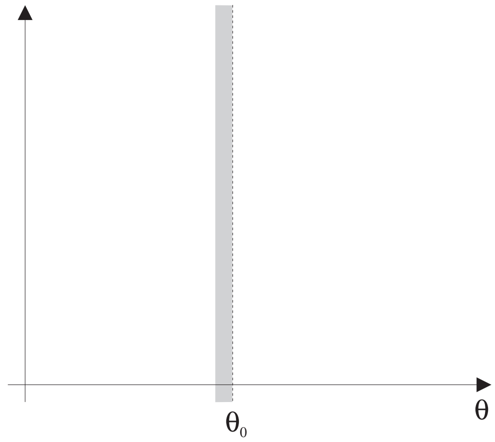


Figure A-11: The qcs-curve implemented by the geometry in Figure A-10.

A.6 Rotating Rotor-Stop

Implementations for:

- interaction: rotation stopped by rotating face
- curve length: finite
- curve type: horizontal or vertical

The geometry shown in Figure A-12 can be used to implement a rotating rotor-stop. Here we show how it is used to implement vertical curves of type C. To implement vertical curves of type G, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Horizontal curves are implemented by defining degree of freedom y to be the abscissa of the cs-plane and x to be the ordinate.

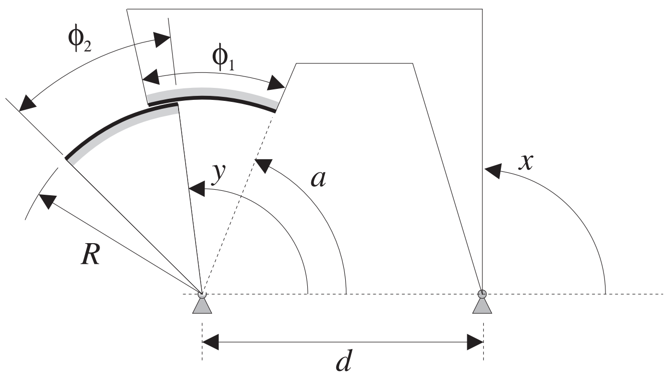


Figure A-12: Geometry used to implement a rotating rotor-stop. The positions of the rotors are measured positive counterclockwise.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is G, the geometry must satisfy the following constraints:

$$R, d, \phi_1, \phi_2 > 0 \tag{A.68}$$

$$a > 0 \tag{A.69}$$

$$a + \phi_1 < \pi \tag{A.70}$$

$$\phi_2 < 2\pi - \phi_1 \tag{A.71}$$

END POINTS

Figure A-13 shows the qcs-curve implemented by the geometry in Figure A-12.

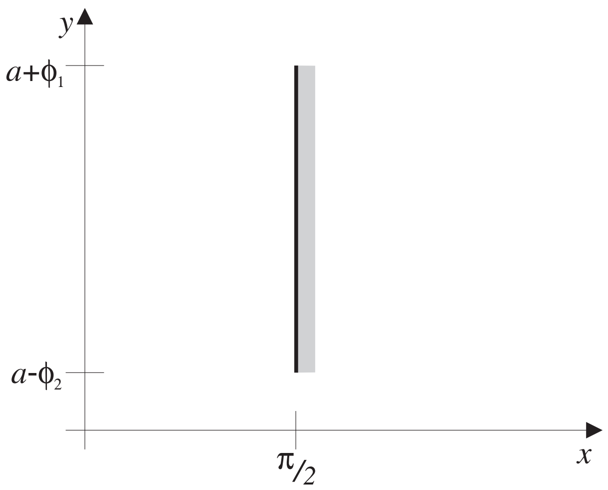


Figure A-13: The qcs-curve implemented by the geometry in Figure A-12.

A.7 Translating Rotor-Stop

Implementations for:

- interaction: rotation stopped by translating face
- curve length: finite
- curve type: horizontal or vertical

The geometry in Figure A-14 can be used to implement a translating rotor-stop. Here we show how it is used to implement horizontal curves of type A. To implement horizontal curves of type E, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Vertical curves are implemented by defining degree of freedom θ to be the abscissa of the cs-plane and x to be the ordinate.

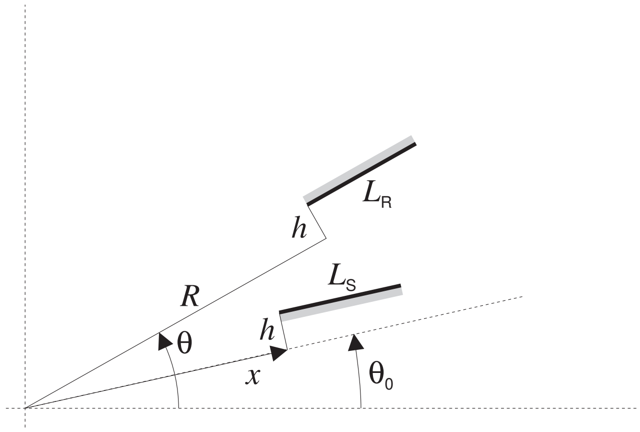


Figure A-14: Geometry used to implement a translating rotor-stop. The position of the rotor is measured positive counterclockwise with angle θ. The translating face slides along a guide located at angle θ_0.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is A, the geometry must satisfy the following constraints:

$$L_R, L_S, R > 0 \tag{A.72}$$

END POINTS Figure A-15 shows the qcs-curve implemented by the geometry in Figure A-14.

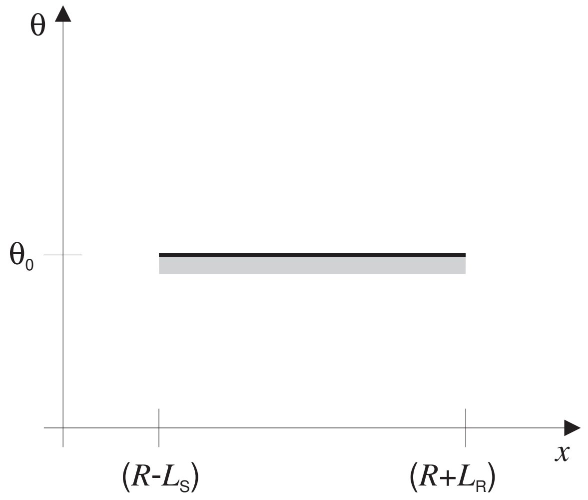


Figure A-15: The qcs-curve implemented by the geometry in Figure A-14.

A.8 Fixed Slider-Stop

Implementations for:

- interaction: translation stopped by fixed face
- curve length: infinite
- curve type: horizontal or vertical

The geometry shown in Figure A-16 can be used to implement a fixed slider-stop. Here we show how to implement vertical curves of type C. To implement vertical curves of type G, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Horizontal curves are implemented by defining the slider's degree of freedom to be the ordinate of the cs-plane rather than the abscissa.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is C, the geometry must satisfy the following constraints:

$$L_1, e > 0 \tag{A.73}$$

$$\phi > 0 \tag{A.74}$$

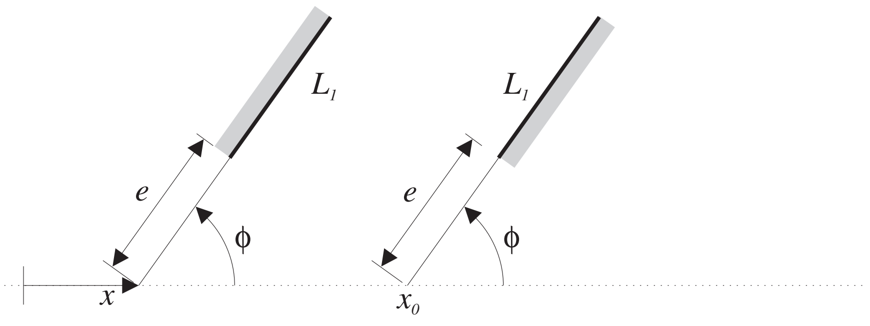


Figure A-16: Geometry used to implement a fixed slider-stop. The position of the slider is measured positive to the right with x. The fixed face is located at x_0.

$$\phi < \pi \tag{A.75}$$

END POINTS

Figure A-17 shows the qcs-curve implemented by the geometry in Figure A-16.

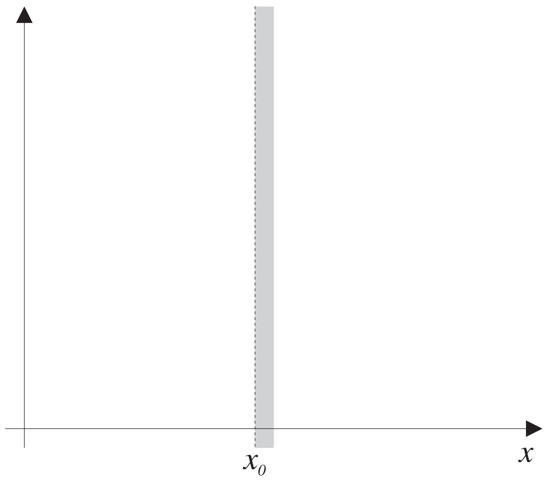


Figure A-17: The qcs-curve implemented by the geometry in Figure A-16.

A.9 Rotating Slider-Stop

Implementations for:

- interaction: translation stopped by rotating face
- curve length: finite
- curve type: horizontal or vertical

The geometry in Figure A-18 can be used to implement a rotating slider-stop. Here we show how it is used to implement vertical curves of type G. To implement vertical curves of type C, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Horizontal curves are implemented by defining degree of freedom y to be the abscissa of the cs-plane and x to be the ordinate.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is G, the geometry must satisfy the following constraints:

$$\phi, R > 0 \tag{A.76}$$

$$a < b \tag{A.77}$$

$$b < c \tag{A.78}$$

$$b - a < \pi/2 \tag{A.79}$$

$$c - b < \pi/2 \tag{A.80}$$

$$\phi < 2\pi - (c - a) \tag{A.81}$$

END POINTS

Figure A-19 shows the qcs-curve implemented by the geometry in Figure A-18.

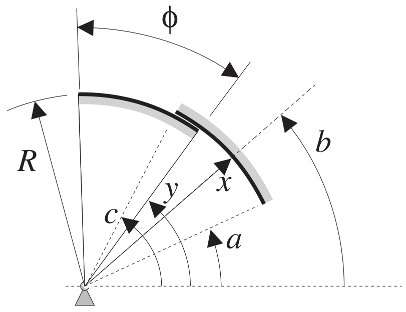


Figure A-18: Geometry used to implement a rotating slider-stop. The position of the slider is measured with displacement x along a fixed axis inclined an angle b from the horizontal. The angle of the rotor is measured positive counterclockwise with angle y.

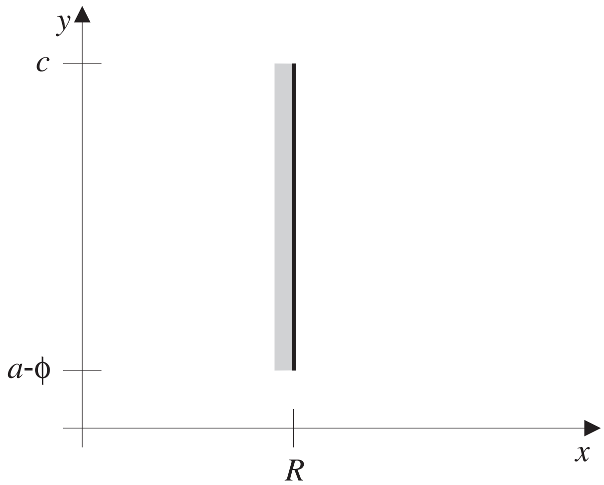


Figure A-19: The qcs-curve implemented by the geometry in Figure A-18.

A.10 Translating Slider-Stop

Implementations for:

- interaction: translation stopped by translating face
- curve length: finite
- curve type: horizontal or vertical

The geometry in Figure A-20 can be used to implement a translating slider-stop. Here we show how to implement vertical curves of type C. To implement vertical curves of type G, the inside and outside of each face must be swapped (i.e., the side of the face that is currently the outside becomes the inside). Horizontal curves are implemented by defining degree of freedom y to be the abscissa of the cs-plane and x to be the ordinate.

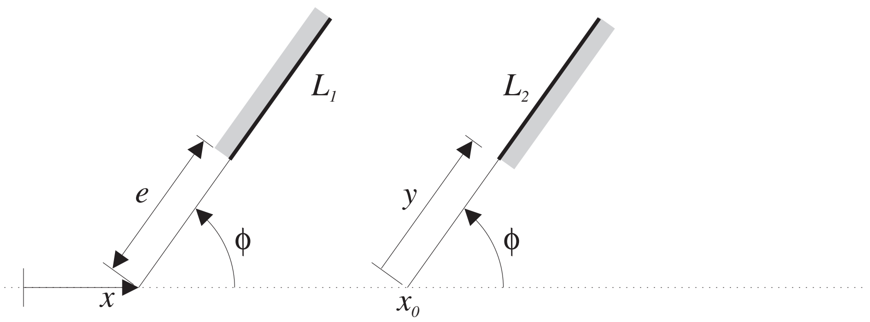


Figure A-20: Geometry used to implement a translating slider-stop. The position of one slider is measured positive to the left with x. The position of the other slider (the one that acts as a stop) is measured by displacement y along a fixed guide inclined an angle ϕ from horizontal. This guide intersects the path of the other slider at x_0.

CONSTRAINTS

No constraints are necessary to ensure that the cs-curve is monotonic. To ensure that the qualitative slope is C, the geometry must satisfy the following constraints:

$$L_1, L_2, e > 0 \tag{A.82}$$

$$\phi > 0 \tag{A.83}$$

$$\phi < \pi \tag{A.84}$$

END POINTS

Figure A-21 shows the qcs-curve implemented by the geometry in Figure A-20.

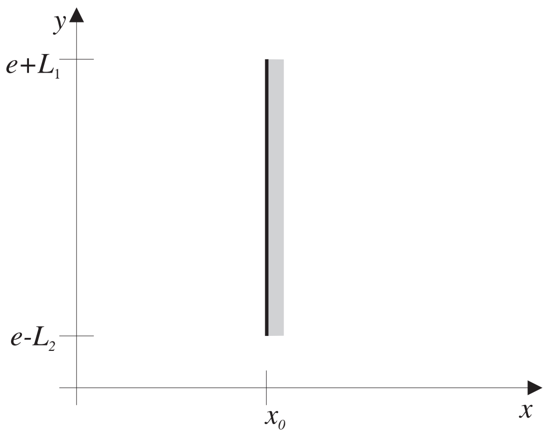


Figure A-21: The qcs-curve implemented by the geometry in Figure A-20.

Appendix B

Sketching Tool

This appendix describes the CAD-like sketching tool (sketcher) we use to create sketches for SketchIT.[1] Figure B-1 shows the sketcher being used to create a sketch of a circuit breaker.

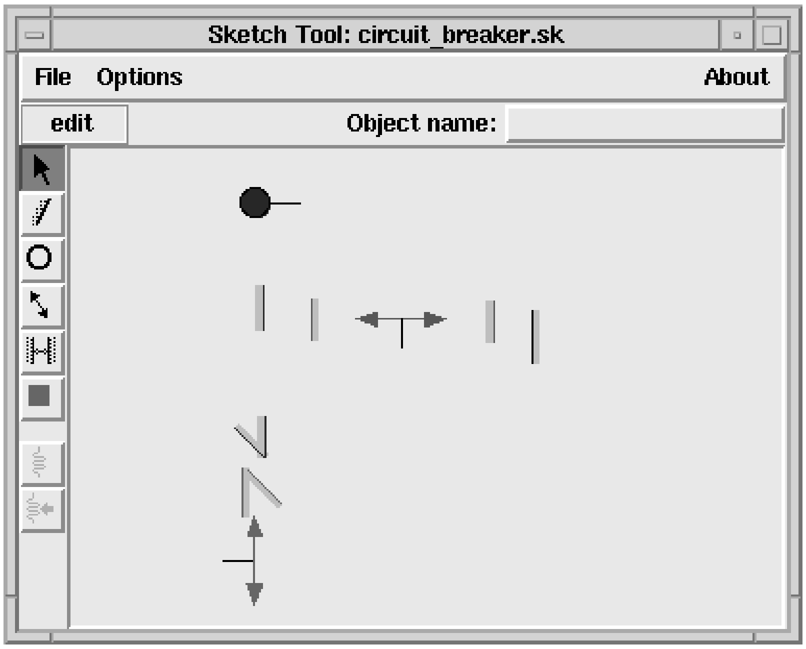


Figure B-1: The sketching tool used to create sketches for SketchIT.

The tool-bar at the left of the sketcher window provides a set of tools and objects. From the top down these are: the selection and editing tool (arrow); the face object for creating faces on bodies; the pivot object; the slider joint object; the tool for

[1] Mike Wessler skillfully and generously wrote the sketching tool.

specifying which pairs of faces interact; the "current body tool" which uses color to indicate the body to which the next object will be attached; the spring object; and the actuator object.

The user constructs a sketch from icons and line segments. The sketcher provides icons for springs, actuators, slider joints, and pivots (the icons correspond to objects from the tool-bar). The pivot icon is a small circle with a short radial line. The radial line serves as a handle for rotating the body attached to the pivot. To attach a spring to a rotating body, one first rotates the body (using the radial line on the pivot) to the approximate neutral position of the spring. Then selecting the spring icon in the tool-bar, and clicking on the pivot, attaches a spring (with appropriate neutral position) to the body. Actuators (with motion limits) are attached in a similar way.

The icon for a slider joint is a two-headed arrow with a short line extending from it. This line has the same purpose as the radial line on the pivot icon: it is used to define neutral positions and motion limits when attaching springs and actuators, respectively.

Because SKETCHIT requires that all of the springs attached to a single body be lumped into a single, equivalent spring, the sketcher allows only one spring to be attached to a body. For simplicity, the sketcher likewise allows only one actuator to be attached to a body. SKETCHIT however can handle multiple actuators attached to a single body as long as only one is active at any given time (it is not physically possible for two or more actuators to be active at the same time unless they all apply exactly the same motion.)

Interacting faces are drawn as line segments. Shading indicates the back side of a face (i.e., the side of the face that is inside the body containing the face). Color coding indicates which faces are attached to which bodies. The designer specifies that a pair of faces is intended to interact by pressing the fifth button (from the top) in the tool bar and selecting the pair of faces.

Because SKETCHIT works in the domain of fixed-axis devices, the sketcher ensures that each body is connected to ground by either a single pivot or a single slider joint.

Appendix C

Computing Numerical C-Space

SketchIT produces the first candidate qc-space for a sketch by abstracting the sketch's numerical c-space. This appendix describes how SketchIT first computes that numerical c-space. We could have used general purpose techniques for computing numerical c-space (see, for example, [34], [1], [26], and [2]), but because SketchIT requires only the locations of the curve end points, we developed simplified, special purpose techniques that compute just this information. The simplifying insight was that for fixed-axis devices with flat faces, we need to handle only six different kinds of interactions: a rotating face interacting with a translating face, two rotating faces, two translating faces, a rotating face interacting with a fixed face, and a translating face interacting with a fixed face.

We implemented only the first and last of these, because that was enough for the three examples we used to test SketchIT, and because computing c-space curves is a relatively well understood problem that was not a focus of this work. We describe the two implemented cases below.

C.1 Interaction: A Rotating Face & a Translating Face

This section describes how we compute the end points and the blocked space side of the cs-curve for a rotating flat face interacting with a translating flat face. The geometry of a typical pair of faces is shown in Figure C-1. The end points of the cs-curve correspond to the largest and smallest displacements of the slider (and rotor) for which the faces still touch. The analysis reduces to one of three special cases summarized below (see Figure C-1 for a definition of the terms):

- Case 1: If $\max(s_a, s_b) < \min(r_a, r_b)$ the two extremes are:
  - $*$ point $\mathbf{R}_a$ on the rotor touching point $\mathbf{S}_a$ on the slider
  - $*$ point $\mathbf{R}_b$ on the rotor touching point $\mathbf{S}_b$ on the slider

- Case 2: If $r_b > r_a$, $s_a > r_a$, and $s_b < r_b$ the two extreme displacements occur when point $\mathbf{R}_b$ on the rotor touches point $\mathbf{S}_b$ on the slider.
- Case 3: If $r_a > r_b$, $s_a < r_a$, and $s_b > r_b$ the two extreme displacements occur when $\mathbf{R}_a$ on the rotor touches point $\mathbf{S}_a$ on the slider.
- Otherwise, the two faces do not touch each other and the cs-curve is not defined.

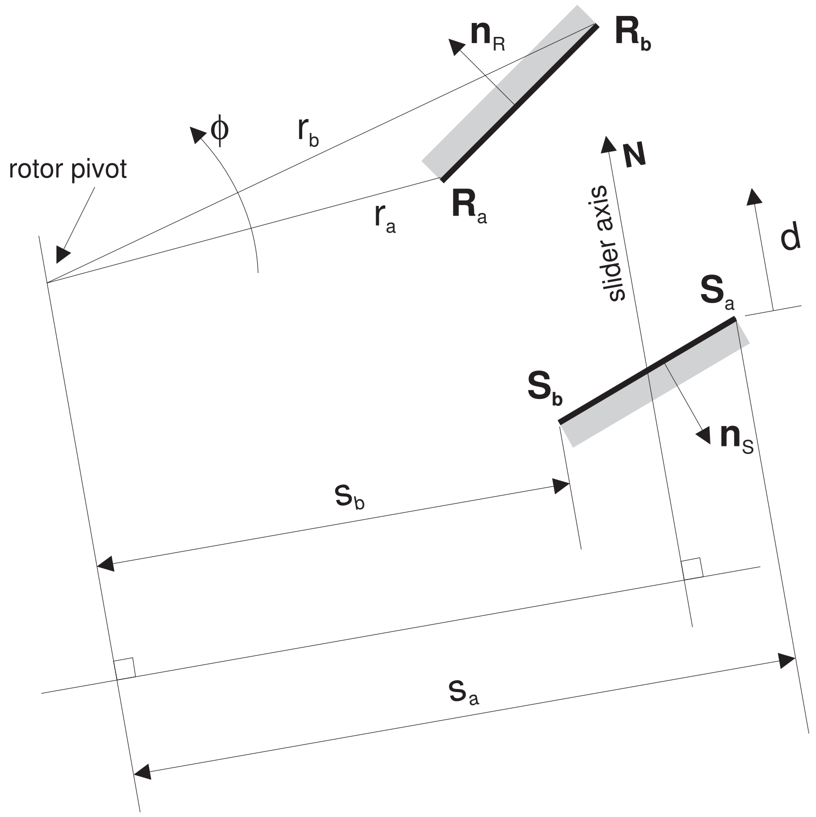


Figure C-1: The interaction of a rotating flat face with a translating flat face. The rotor's angle is ϕ and the slider's position is d. Points $\mathbf{R}_a$ and $\mathbf{R}_b$ denote the ends of the face on the rotor, and are expressed in the rotor's coordinate system. When looking along the rotor face in the direction from $\mathbf{R}_a$ to $\mathbf{R}_b$, the inside of the rotor will be on the left-hand side. $\mathbf{n}_R$ is the inward normal to the face. $\mathbf{S}_a$, $\mathbf{S}_b$, and $\mathbf{n}_S$ are defined similarly. $\mathbf{N}$ is the direction of the slider axis

The first case is the situation depicted in Figure C-2. The distinguishing feature of this case is that the path of the slider intersects the path of the rotor in two separate locations. Hence, there are two separate ranges of engagement. However, only one of the ranges is physically possible. In the example shown in the figure, the range of engagement at the top of the figure is the impossible one: this range of engagement would require the rotor face to touch the back side of the slider face.[1]

For the second and third cases, shown in Figures C-3 and C-4 respectively, the path of the slider intersects the rotor in one location. For the second case, the slider pushes the rotor counterclockwise, while in the third case it pushes the rotor clockwise.

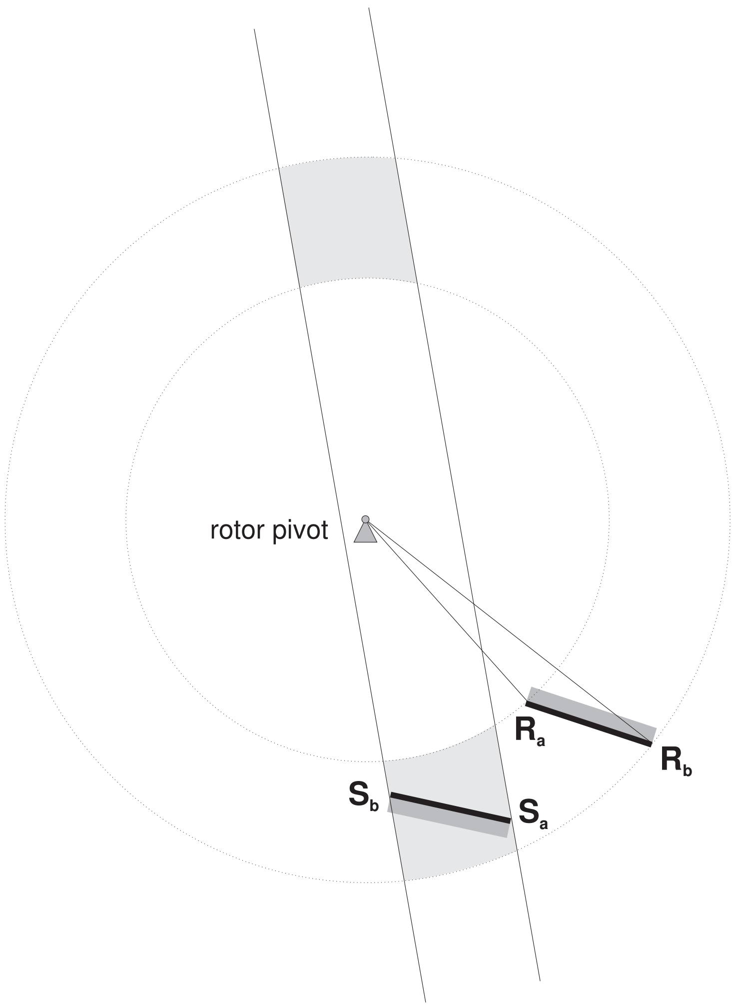


Figure C-2: Case 1: As the shaded regions indicate, the path of the slider intersects the path of the rotor in two separate locations.

[1]Recall that in the final design, the slider face will be one face on a solid body.

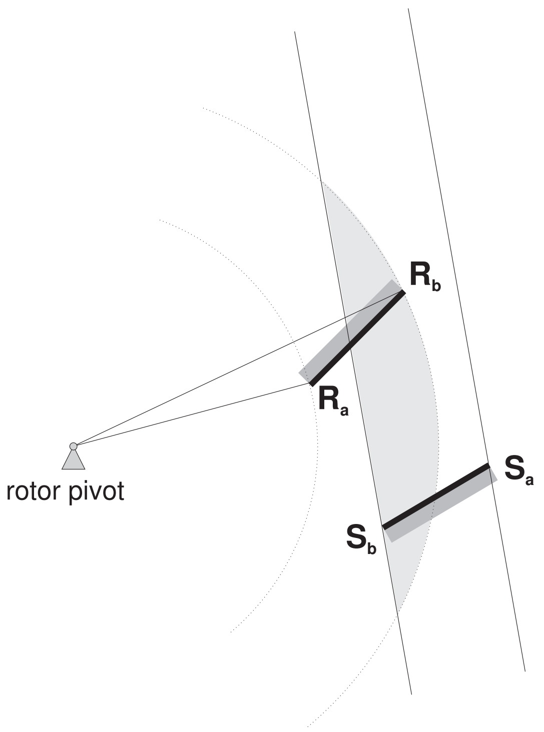


Figure C-3: Case 2: As the shaded region indicates, the path of the slider intersects the path of the rotor in just one location. The slider pushes the rotor counterclockwise.

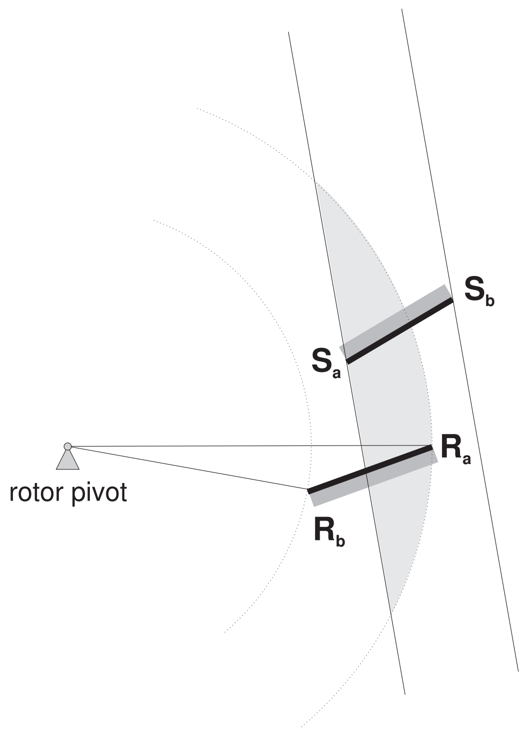


Figure C-4: Case 3: As the shaded region indicates, the path of the slider intersects the path of the rotor in just one location. The slider pushes the rotor clockwise.

All three of these cases require computing the configurations in which a particular point on a rotating body touches a particular point on a translating body. For example, the second case requires computing the two configurations in which point $\mathbf{R}_b$ on the rotor touches point $\mathbf{S}_b$ on the slider.

To obtain a general solution to the problem, we use the coordinate systems shown in Figure C-5 (variables shown in bold are vector quantities). Point $\mathbf{Q}_R$ is an arbitrary point on the rotor (rotating body), $\mathbf{Q}_S$ is an arbitrary point on the slider (translating body). For the two points to touch, they must be equidistant from the rotor pivot (as well as satisfying other criteria described below). If we define $\mathbf{Z}$ as a vector from the pivot to point $\mathbf{Q}_S$, and r as the radius from the pivot to point $\mathbf{Q}_R$, the condition is:

$$\mathbf{Z} \cdot \mathbf{Z} = r^2 \tag{C.1}$$

Next, we express $\mathbf{Z}$ in terms of the locations of the rotor pivot and slider guide in the global coordinate system, and the location of the slider point relative to the slider guide (we express $\mathbf{Z}$ in the global coordinate system):

$$\mathbf{Z} = \mathbf{P}_S - \mathbf{P}_R + \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \left(\mathbf{Q}_S + \begin{bmatrix} d \\ 0 \end{bmatrix} \right) \tag{C.2}$$

Substituting this expression for $\mathbf{Z}$ into Equation C.1 produces a new expression that we solve to obtain two values (the equation is quadratic) for d; these represent the two positions of the slider for which the point on the rotor can touch the point on the slider.

For each of these two slider positions, there is unique position of the rotor for which the point on the rotor touches the point on the slider. To compute these rotor positions we use the fact that, during contact, the vector $\mathbf{Z}$ from the pivot to point $\mathbf{Q}_S$ is the same as the vector from the pivot to point $\mathbf{Q}_R$:

$$\mathbf{Z} = r \begin{bmatrix} \cos(\phi + \phi_0) \\ \sin(\phi + \phi_0) \end{bmatrix} \tag{C.3}$$

To solve for the rotor angle, ϕ, corresponding to one of the two values for d, we substitute the value of d into Equation C.2 to obtain a value for $\mathbf{Z}$. Then, we substitute this value of $\mathbf{Z}$ into Equation C.3 and solve for ϕ.

The analysis produces two possible configurations because there are two sets of values for d and ϕ. A set of values is valid only if, in that configuration, the normals to the faces are in opposite directions. If the normals were in the same direction, one face would have to be touching the back side of the other, but this is not physically possible. If, for example, the rotor and slider in Figure C-2 are touching in a configuration inside the shaded region at the top of the figure, the rotor will be touching the back side of the slider face. Using the normals to the faces defined in Figure C-1, the criterion for selecting valid configurations is:

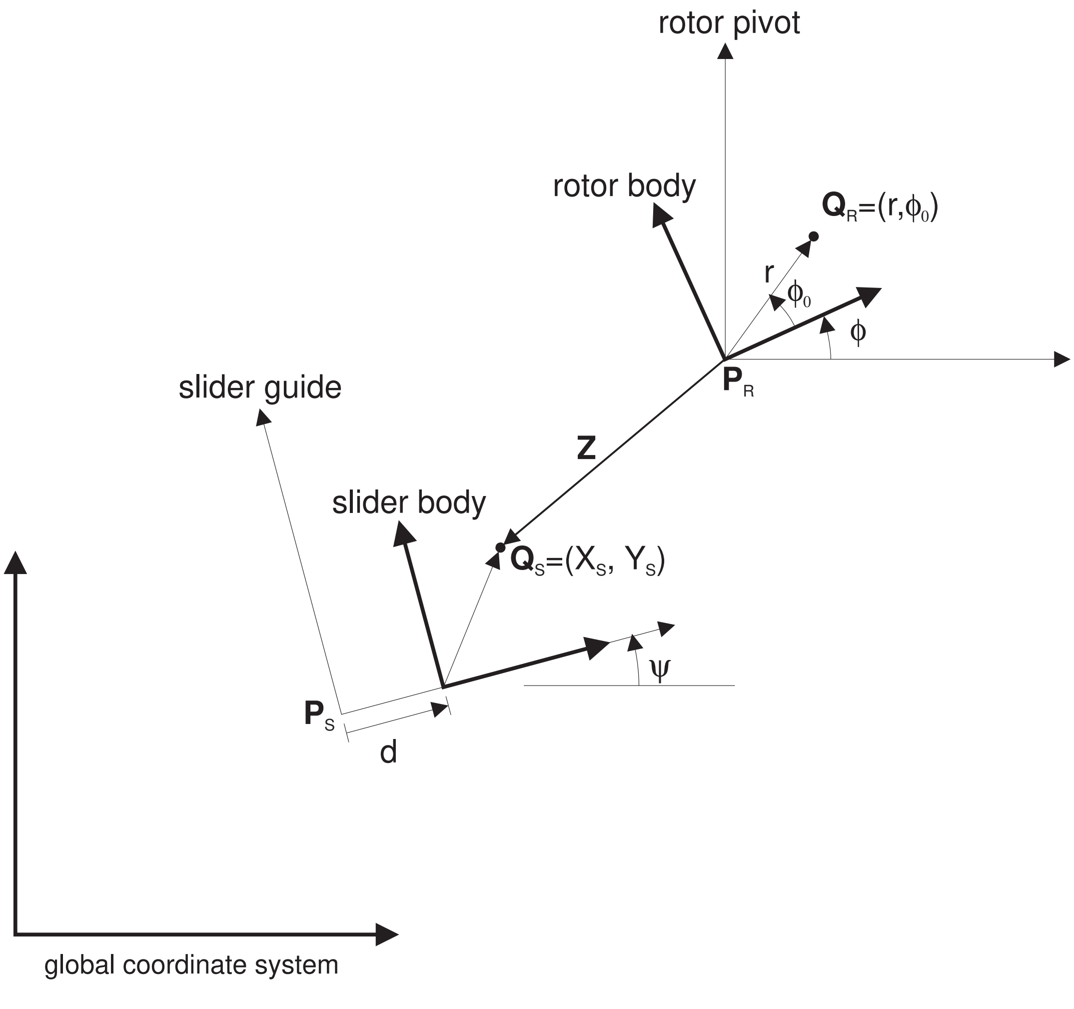


Figure C-5: Coordinate systems used for computing the configurations in which a point on a rotating body touches a point on a translating body. The coordinate system of the rotor pivot is located at point $\mathbf{P}_R$ in the global coordinate system. The displacement of the rotor body is denoted by ϕ. Point $\mathbf{Q}_R$ is the point on the rotor expressed in the coordinate system of the rotor body (expressed in polar coordinates). The coordinate system of the slider guide is located at the point $\mathbf{P}_S$ and has inclination ψ with respect to the global coordinate system. The slider body coordinate system translates along the x-axis of the slider guide coordinate system with displacement d. Point $\mathbf{Q}_S$ is the point on the slider expressed in the slider body coordinate system.

$$\mathbf{n}_R \cdot \mathbf{n}_S < 0 \tag{C.4}$$

To compute the end points of the cs-curve, we select the appropriate case from the three listed previously, thereby determining the configurations corresponding to the two end points. We then use Equations C.1 through C.4 to solve for the positions of the rotor and slider corresponding to each of these configurations. The positions thus obtained are the coordinates of the two end points.

Once we have the end points, we compute the qualitative slope and blocked space side of the cs-curve. We express the result as one of the curve types shown in Figure 3-2. Without loss of generality, we assume that the slider's displacement, d, is the abscissa of the cs-plane, and the rotor's displacements, ϕ, is the ordinate. Then, we compute the angle α (measured from horizontal) of a straight line connecting the two end points. Finally, we use these rules to match the cs-curve to one of the qcs-curves in Figure 3-2:

- If $\alpha = 0, \pi$:
  - ∗ If $(\mathbf{R}_a \times \mathbf{n}_R) \cdot \hat{k} > 0$ (where $\hat{k}$ is a unit vector in the global z-direction) then Type A
  - ∗ Else Type E
- If $0 < \alpha < \pi/2$ or $\pi < \alpha < 3\pi/2$:
  - ∗ If $\mathbf{N} \cdot \mathbf{n}_S < 0$ then Type B
  - ∗ Else Type F
- If $\alpha = \pi/2, 3\pi/2$
  - ∗ If $\mathbf{N} \cdot \mathbf{n}_S < 0$ then Type C
  - ∗ Else Type G
- If $\pi/2 < \alpha < \pi$ or $3\pi/2 < \alpha < 2\pi$
  - ∗ If $\mathbf{N} \cdot \mathbf{n}_S < 0$ then Type D
  - ∗ Else Type H

C.2 Interaction: A Translating Face & a Stationary Face

This section describes how we compute the location and blocked space side of the cs-curve for a translating flat face interacting with a stationary flat face. As we saw in Figure 2-4, the cs-curve for this kind of interaction is an infinite boundary. The location of an infinite boundary is expressed in terms of the point at which it crosses one of the coordinates axes in the cs-plane (for a finite cs-curve, on the other hand, the location is expressed in terms of the two end points).

The geometry of a typical pair of faces is shown in Figure C-6. Because there is only one degree of freedom (translation of the slider), there is only one configuration in which the faces touch. This configuration, which corresponds to an end point of one face touching the other face, defines the location of the infinite boundary.

Our strategy for computing this is to compute all of the mathematically possible configurations in which an end point of one face touches the other face, and then to select from these the one configuration that is physically possible.

We begin by expressing the locations of the end points of the stationary face in the slider guide coordinate system (vectors are in bold, and superscripts indicate the x and y-components of vector quantities.):

$$\mathbf{T}_{Rot} = \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \tag{C.5}$$

$$\bar{\mathbf{F}}_a = \mathbf{T}_{Rot}\,(\mathbf{F}_a - \mathbf{P}_S) \tag{C.6}$$

$$\bar{\mathbf{F}}_b = \mathbf{T}_{Rot}\,(\mathbf{F}_b - \mathbf{P}_S) \tag{C.7}$$

Next, we sort the end points of the faces with respect to the y-direction of the slider guide coordinate system:

$$\mathbf{S}_{top} = \begin{cases} \mathbf{S}_a & \mathbf{S}_b^Y < \mathbf{S}_a^Y \\ \mathbf{S}_b & \text{otherwise} \end{cases} \tag{C.8}$$

$$\mathbf{S}_{bot} = \begin{cases} \mathbf{S}_b & \mathbf{S}_b^Y < \mathbf{S}_a^Y \\ \mathbf{S}_a & \text{otherwise} \end{cases} \tag{C.9}$$

$$\bar{\mathbf{F}}_{top} = \begin{cases} \bar{\mathbf{F}}_a & \bar{\mathbf{F}}_b^Y < \bar{\mathbf{F}}_a^Y \\ \bar{\mathbf{F}}_b & \text{otherwise} \end{cases} \tag{C.10}$$

$$\bar{\mathbf{F}}_{bot} = \begin{cases} \bar{\mathbf{F}}_b & \bar{\mathbf{F}}_b^Y < \bar{\mathbf{F}}_a^Y \\ \bar{\mathbf{F}}_a & \text{otherwise} \end{cases} \tag{C.11}$$

The next step is to compute each of the mathematically possible configurations in which an end point of one face touches the other face. Computing one of these configurations is equivalent to intersecting two lines: a line that passes through the point and is parallel to the slider axis, and a line defined by the face.

We illustrate the process by computing the configuration in which point S_{top} on the slider touches the fixed surface. This configuration is (mathematically) possible only if the point is in line with the face:

$$\begin{array}{l} \mathbf{S}^Y_{top} \leq \bar{\mathbf{F}}^Y_{top} \\ \mathbf{S}^Y_{top} \geq \bar{\mathbf{F}}^Y_{bot} \end{array} \tag{C.12}$$

If Equation C.12 is satisfied, then contact is possible, and we compute the position the slider must occupy for the contact to occur:

$$d = \begin{cases} \bar{\mathbf{F}}^X_{top} - \mathbf{S}^X_{top} & \text{if } \bar{\mathbf{F}}^X_{top} = \bar{\mathbf{F}}^X_{bot} \\ \bar{\mathbf{F}}^X_{bot} + \frac{\bar{\mathbf{F}}^X_{top} - \bar{\mathbf{F}}^X_{bot}}{\bar{\mathbf{F}}^Y_{top} - \bar{\mathbf{F}}^Y_{bot}} \left(\mathbf{S}^Y_{top} - \bar{\mathbf{F}}^Y_{bot} \right) - \mathbf{S}^X_{top} & \text{otherwise} \end{cases} \tag{C.13}$$

We perform a similar analysis for the other three point-face combinations by making the obvious variable substitutions in Equations C.12 and C.13.

If the complete analysis determines that none of the four point-face contacts is possible, then there is no interaction between the two faces. If the analysis determines that point-face contacts are possible, we must determine which of the mathematically possible contacts is physically possible, and hence determine the location of the infinite boundary. We use the following test: If the inward normal to the slider face ($\mathbf{n}_S$) points in the negative x-direction of the slider body coordinate system, the location is the minimum of the d values for the point-face contacts. In addition, assuming the slider position is the abscissa of the cs-plane, the blocked side of the infinite boundary is the same as that of qcs-curve C in Figure 3-2. On the other hand, if the inward normal does not point in the negative x-direction, the location is the maximum of the d values, and the blocked side is the same as that of qcs-curve G.

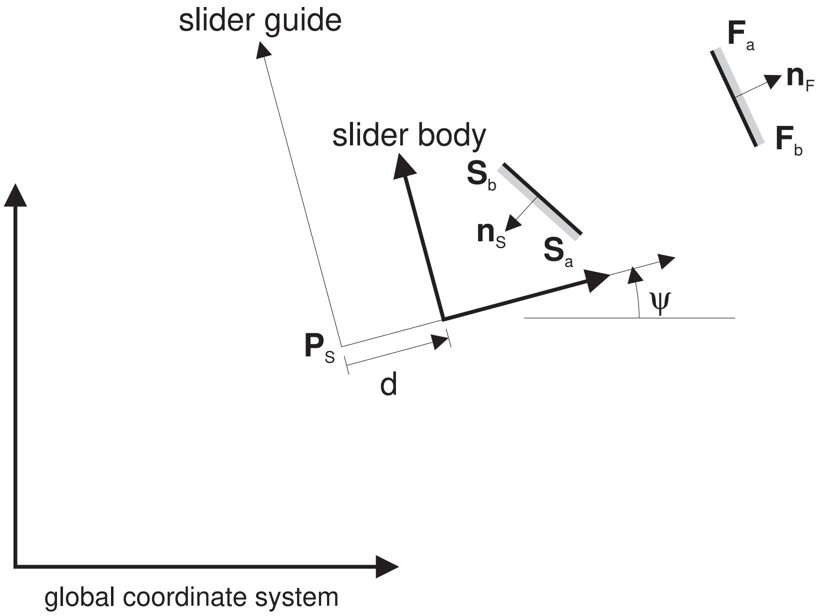


Figure C-6: The interaction of a translating flat face and a stationary flat face. The slider's position is d. The coordinate system of the slider guide is located at the point $\mathbf{P}_S$, and has inclination ψ with respect to the global coordinate system. Points $\mathbf{S}_a$ and $\mathbf{S}_b$ denote the ends of the slider face, and are expressed in the slider body coordinate system. When looking along the slider face in the direction from $\mathbf{S}_a$ to $\mathbf{S}_b$, the inside of the rotor will be on the left-hand side. $\mathbf{n}_S$ is the inward normal to the face. $\mathbf{F}_a$, $\mathbf{F}_b$, and $\mathbf{n}_F$ are defined similarly, but are expressed in the global coordinate system.

Bibliography

[1] R. C. Brost. Analysis and planning of planar manipulation tasks. Technical Report CMU-CS-91-149, Carnegie Melon University, 1991.

[2] Michael E. Caine. The design of shape from motion constraints. Technical Report 1425, MIT AI Lab., September 1993.

[3] Kenneth W. Chase and Alan R. Parkinson. A survey of research in the application of tolerance analysis to the design of mechanical assemblies. *Research in Engineering Design*, 3:23–37, 1991.

[4] Stephen H. Crandall, Dean C. Karnopp, Edward F. Kurtz, Jr., and David C. Pridmore-Brown. *Dynamics of Mechanical and Electromechanical Systems.* Robert E. Krieger Publishing Co., Malabar, Florida, 1968.

[5] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[6] Johan de Kleer. *Causal and Teleological Reasoning in Circuit Recognition.* PhD thesis, Massachusetts Institute of Technology, September 1979.

[7] Johan de Kleer and John Seely Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[8] John R. Dixon, Eugene C. Libardi, Jr., Steven C. Luby, and Mohan Vaghul. Expert systems for mechanical design: Examples of symbolic representations of design geometries. *Engineering with Computers*, 2(1):1–10, 1987.

[9] Richard James Doyle. Hypothesizing device mechanisms: Opening up the black box. Technical Report 1047, MIT AI Lab., 1988.

[10] Arthur G. Erdman and George N. Sandor. *Mechanism Design: Analysis and Synthesis*, volume 1. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[11] Brian Falkenhainer and Kenneth D. Forbus. Compositional modeling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.

[12] Boi Faltings. Qualitative kinematics in mechanisms. *Artificial Intelligence*, 44:89–119, 1990.

[13] Boi Faltings. Qualitative models in conceptual design: A case study. In *Reasoning with Diagrammatic Representations, Papers from the 1992 Spring Symposium, Technical Report SS-92-02*, pages 69–74. AAAI Press, 1992.

[14] Boi Faltings. A symbolic approach to qualitative kinematics. *Artificial Intelligence*, 56:139–170, 1992.

[15] Boi Faltings, Emmanuel Baechler, and Jeff Primus. Reasoning about kinematic topology. In *Proceedings IJCAI-89*, pages 1331–1336, 1889.

[16] Boi Faltings and Kun Sun. Computer-aided creative mechanism design. In *Proceedings IJCAI-93*, pages 1451–1457, 1993.

[17] Ken D. Forbus, Paul Nielsen, and Boi Faltings. Qualitative spatial reasoning: The clock project. Technical Report 9, Northwestern University, The Institute for the Learning Sciences, 1991.

[18] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[19] Rakesh Gupta and Mark J. Jakiela. Simulation and shape synthesis of kinematic pairs via small-scale interference detection. *Research in Engineering Design*, 6:103–123, 1994.

[20] Edward J. Haug. *Computer-Aided Kinematics and Dynamics of Mechanical Systems, Volume I: Basic Methods*. Allyn and Bacon, Boston, 1989.

[21] Jon Kieth Hirschtick. Geometric feature extraction using production rules. Master's thesis, Massachusetts Institute of Technology, 1986.

[22] Jack Hodges. Naive mechanics, a computational model of device use and function in design improvisation. *IEEE Expert*, 7(1):14–27, February 1992.

[23] Yumi Iwasaki, Richard Fikes, Marcos Vescovi, and B. Chandrasekaran. How things are intended to work: Capturing functional knowledge in device design. In *Proceedings IJCAI-93*, pages 1516–1522, 1993.

[24] Leo Joskowicz. Simplification and abstraction in kinemactic behaviors. In *Proceedings IJCAI-89*, pages 1337–1342, 1989.

[25] Leo Joskowicz and Sanjaya Addanki. From kinematics to shape: An approach to innovative design. In *Proceedings AAAI-88*, pages 347–352, 1988.

[26] Leo Joskowicz and Elisha Sacks. Computational kinematics. *Artificial Intelligence*, 51:381–416, 1991.

[27] Leo Joskowicz and Elisha Sacks. Configuration space computation for mechanism design. In *Proceedings of the IEEE Robotics and Automation Conference*, 1994.

[28] Leo Joskowicz, Elisha Sacks, and Vijay Srinivasan. Kinematic tolerance analysis. In *3rd ACM Symposium on Solid Modeling and Applications*, 1995.

[29] Thamas R. Kane and David A Levinson. *Dynamics: Theory and Applications.* McGraw Hill, New York, 1985.

[30] Dean Karnopp and Ronald Rosenberg. *System Dynamics: A Unified Approach.* John Wiley and Sons, New York, 1975.

[31] Sridhar Kota and Shean-Juinn Chiou. Conceptual design of mechanisms based on computational synthesis and simulation of kinematic building blocks. *Research in Engineering Design*, 4:75–87, 1992.

[32] Glenn A. Kramer. Solving geometric constraint systems. In *Proceedings AAAI-90*, pages 708–714, 1990.

[33] Benjamin Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–388, 1986.

[34] Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2), February 1983.

[35] Timothy A. Mashburn and David C. Anderson. Automatically deriving behavior constraints for performance variables in mechanical design. *Research in Engineering Design*, 6:85–102, 1994.

[36] J. L. Meriam. *Dynamics.* John Wiley and Sons, Inc., New York, second edition, 1975.

[37] N. Hari Narayanan, Makasi Suwa, and Hiroshi Motoda. How things appear to work: Predicting behaviors from device diagrams. In *Proceedings AAAI-94*, pages 1161–1167, 1994.

[38] P. Pandurang Nayak, Leo Joskowicz, and Sanjaya Addanki. Automated model selection using context-dependent behaviors. In *Proceedings AAAI-92*, pages 710–716, 1992.

[39] Parviz E. Nikravesh. *Computer-Aided Analysis of Mechanical Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[40] M. F. Orelup, J. R. Dixon, and M. K. Simmons. Dominic ii: More progress towards domain independent design by iterative redesign. In *Intelligent and Integrated Manufacturing Analysis and Synthesis, Winter Annual Meeting of ASME*, pages 67–80, 1987.

[41] D. R. Prabhu and D. L. Taylor. Synthesis of systems form specifications containing orientations and positions associated with flow variables. In *Advances in Design Automation, Volume 1, Computer-Aided and Computational Design*, pages 273–279, 1989.

[42] D. Rosen, D. Riley, and A. Erdman. A knowledge based dwell mechanism assistant designer. *Journal of Mechanical Design*, 113:205–212, September 1991.

[43] Elisha Sacks and Leo Joskowicz. Automated modeling and kinematic simulation of mechanisms. *Computer-Aided Design*, 25(2):106–118, February 1993.

[44] D. Serrano and D. C. Gossard. Combining mathematical models with geometric models in cae systems. In *Proceedings of the 1986 ASME International Computers in Engineering Conference and Exhibition*, pages 277–284, 1986.

[45] David Serrano. *Constraint Management in Conceptual Design*. PhD thesis, Massachusetts Institute of Technology, 1987.

[46] Howard Shrobe. Understanding linkages. In *Proceedings AAAI-93*, pages 620–625, 1993.

[47] Thomas F. Stahovich, Randall Davis, and Howard Shrobe. An ontology of mechanical devices. In *Working Notes, Reasoning about Function, 11th National Conference on Artificial Intelligence*, pages 137–140, 1993.

[48] Thomas F. Stahovich, Randall Davis, and Howard Shrobe. Turning sketches into working geometry. In *Proceedings of the Seventh International ASME Conference on Design Theory and Methodology*, 1995.

[49] Thomas Frank Stahovich. Computational tools for conceptual design. Master's thesis, Massachusetts Institute of Technology, 1990.

[50] Richard M. Stallman and Gerald Jay Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Technical Report Memo 380, MIT AI Lab., September 1976.

[51] Robert F. Steidel, Jr. and Jerald M. Henderson. *The Graphic Languages of Engineering*. John Wiley and Sons, New York, 1983.

[52] Devika Subramanian and Cheuk-San (Edward) Wang. Kinematic synthesis with configuration spaces. In *The 7th International Workshop on Qualitative Reasoning about Physical Systems*, pages 228–239, May 1993.

[53] Karl T. Ulrich. Computation and pre-parametric design. Technical Report 1043, MIT AI Lab., 1988.

[54] Richard V. Welch and John R. Dixon. Guiding conceptual design through behavioral reasoning. *Research in Engineering Design*, 6:169–188, 1994.

[55] Daniel S. Weld and Johan de Kleer, editors. *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[56] Brian C. Williams. Qualitative analysis of mos circuits. *Artificial Intelligence*, 24(1–3):281–346, 1984.

[57] Jan Wolter and Periannan Chandrasekaran. A concept for a constraint-based representation of functional and geometric design knowledge. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 409–418. ACM Press, 1991.