MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 502 November 1978

Constraints

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Abstract:

We present an interactive system organized around networks of constraints rather than the programs which manipulate them. We describe a language of hierarchical constraint networks. We describe one method of deriving useful consequences of a set of constraints which we call propagation. Dependency analysis is used to spot and track down inconsistent subsets of a constraint set. Propagation of constraints is most flexible and useful when coupled with the ability to perform symbolic manipulations on algebraic expressions. Such manipulations are in turn best expressed as alterations or augmentations of the constraint network. Numerous diagrams ornament the text.

Keywords: constraints, propagation of constraints, declarative languages, dependencies, responsible programs, SKETCHPAD

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This work was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

* Fannie and John Hertz Fellow

** Jolly Good Fellow

Introduction

Programming languages are usually organized around unilateral computation. Programs perform predetermined operations on their inputs to produce desired outputs or to evolve a desired process. Physical systems, on the other hand, are usually specified as sets of constraints among several variables. A constraint such as x*y+z=3 is no more about how to compute x given y and z than it is about how to compute z given x and y. Programs are often written to extract useful information from a set of constraints. Such programs depend strongly on the forms of the constraints and are committed to solving for particular variables. For example, programs for electrical circuit analysis are usually not good for circuit synthesis as well.

We present an interactive system organized around networks of constraints rather than the programs which manipulate them. We describe a language of hierarchical constraint networks. It supplies a set of primitive constraint types and a means of building compound constraint systems by combining instances of simpler ones. Constraint networks of this kind can often be used to elegantly describe the specifications of ordinary programs and data structures as well as mathematical models of physical systems.

We also describe one method of deriving useful consequences of a set of constraints which we call propagation. Propagation automatically takes advantage of the sparseness of most constraint networks. Propagation is not prematurely committed to solving for any particular predetermined set of unknowns, nor is it committed to using any particular constraint in a predetermined way. In fact propagation analysis may be freely intermixed with the addition of new constraints and the deletion of old ones. This requires that the system keep track of how its conclusions (and intermediate results) are consequences of the particular constraints they were derived from. Dependency analysis is also used to spot and track down inconsistent subsets of a constraint set. Propagation of constraints is most flexible and useful when coupled with the ability to perform symbolic manipulations on algebraic expressions. Such manipulations are in turn best expressed as alterations or augmentations of the constraint network.

The Language of Constraints

A language is a means of communication of ideas. A language generally has a "theme", the class of ideas which it is optimized for communicating. For example, most computer languages are designed for expressing algorithms. They are optimized for communicating imperative, procedural notions.(Other Languages) The theme of the constraint language is declarative. It is good for expressing descriptions of structural relationships and physical situations.

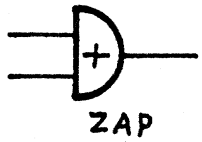
Every language has a set of primitive entities which represent the elementary notions of the domain, and rules of composition by which compound entities are constructed.

Simple Constraints

We will now proceed to illustrate the entities of the constraint language by an annotated interaction with the constraint language interpreter. Input to the interpreter is in lower case, following the prompt "==>".

One kind of primitive (i.e., built-in) constraint we might have is an "adder" which constrains three numbers (called the addend, augend, and sum of the adder) in such a way that the addend plus the augend must equal the sum. Put another way, it constrains the augend to equal the difference of the sum and the addend. (The point is that there is no preferred direction of computation.) First, let's make an adder, called ZAP:

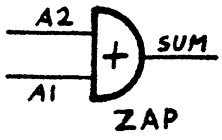
```
==> (create zap adder)
ZAP
```



Every object in the constraint language has a type, and may have parts. We can ask ZAP to tell us about itself:

```
==> (>> type? zap)
ADDER

==> (>> partnames? zap)
(A1 A2 SUM)
```



It seems that ZAP has three parts, the addend (A1), the augend (A2), and the sum (SUM). The parts also have types.

```
==> (>> type? a1 zap)
CELL

==> (>> partnames? a1 zap)
NIL
```

A cell is a primitive entity which hath no parts.[Elements] Cells are used for two things in the constraint language. They are used to hold computational values, and (as we will see later) they can be connected together when building compound constraints. We can use WHAT-IS to discover the value in a cell:

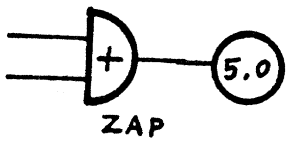
```
==> (what-is (>> sum zap))
Sorry, I don't know it.  I need:
   (>> A1 ZAP)
   (>> A2 ZAP)
to use rule: (>> RULE#1 ZAP)
```

At this point, the system does not know a value for the SUM of ZAP. However, it has told us that if we had given it the A1 and A2 it could have computed a sum using a rule internal to ZAP. Instead, we now give the sum a value.

```
==> (set-parameter (>> sum zap) 5.0)
5.0

==> (what-is (>> sum zap))
(>> SUM ZAP) = 5.0

==> (why (>> sum zap))
Because you told me so.
```



Now that the system knows about the SUM of ZAP, can it tell us about the A1?

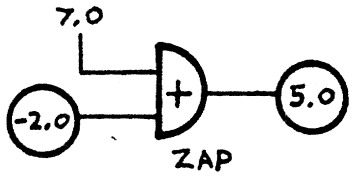
```
==> (what-is (>> a1 zap))
Sorry, I don't know it.  I need:
   (>> A2 ZAP)
to use rule: (>> RULE#2 ZAP)
```

The A1 is not yet completely determined, but it could be determined if the A2 were known. If instead we specify the A1, the A2 should be determined.

```
==> (set-parameter (>> a1 zap) -2.0)
-2.0

==> (what-is (>> a2 zap))
(>> A2 ZAP) = 7.0

==> (why (>> a2 zap))
I used rule (>> RULE#3 ZAP) on the following inputs:
   (>> SUM ZAP)
   (>> A1 ZAP)
```

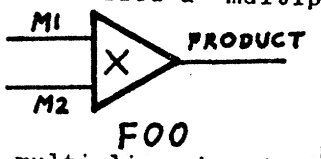


We have just illustrated the use of an adder, a typical entity of the constraint language. A constraint enforces a relationship among several entities. If enough information is known to immediately deduce unknown cell values, those values are computed. These new values may enable further deductions. We call this deductive process "propagation of constraints". (Actually, it is values that are propagated, through a network of constraints.)

Networks of Constraints

Suppose we also have a kind of constraint called a "multiplier":

```
==> (create foo multiplier)
FOO
```



Neither an adder by itself nor a multiplier by itself is a particularly fascinating device. The system provides a means for combining simple constraints into arbitrarily complicated networks.

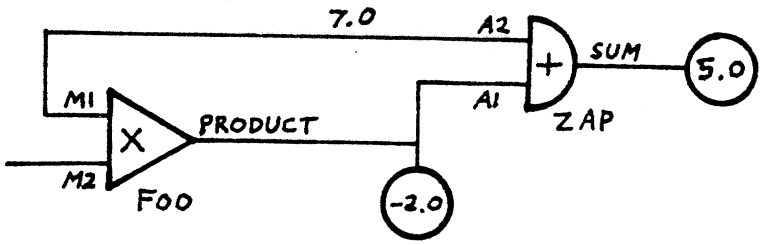
In typical algorithmic computer languages a compound algorithm is built from simpler ones by rules of temporal sequencing and data flow. Algorithms can be temporally concatenated (sequencing), selected among (conditionals), and iterated (do loops). Data flow is indicated explicitly by functional composition and implicitly by side effects on shared variables.

In the constraint language, a compound constraint is built from simpler ones by linking some of their parts. The method of connection is a declaration that two entities are in fact the same.

The "==" primitive allows us to identify any two constraint entities of the same type. In particular, if two cells are identified they are effectively considered to be the same cell -- they are constrained to have the same value.

```
==> (== (>> product foo) (>> a1 zap))
IDENTITY
==> (== (>> m1 foo) (>> a2 zap))
IDENTITY
```

We have linked various parts of our adder ZAP and our multiplier FOO. The resulting network looks like:



Now ZAP had some values in its cells when we linked them to FOO's cells. These values have already been propagated through FOO.

```
==> (what-is (>> m2 foo))
(>> M2 FOO) = -0.285714287

==> (why (>> m2 foo))
I used rule  (>> RULE#5 FOO)  on the following inputs:
   (>> PRODUCT FOO)
   (>> M1 FOO)
```

We can chase these deductions to their ultimate reasons.

```
==> (why (>> product foo))
I used rule  (>> 1<=2)  on the following inputs:
   (>> A1 ZAP)
```

The rule "1<=2" is a manifestation of the linkage between the PRODUCT of FOO and the A1 of ZAP.

```
==> (why (>> a1 zap))
Because you told me so.

==> (why (>> m1 foo))
I used rule  (>> 1<=2)  on the following inputs:
   (>> A2 ZAP)

==> (why (>> a2 zap))
I used rule  (>> RULE#3 ZAP)  on the following inputs:
   (>> SUM ZAP)
   (>> A1 ZAP)
```

It is often convenient to be able to determine the ultimate antecedents of a deduced value. For example, we know the value of the M2 of FOO because of a chain of deductions ultimately derived from our knowledge of the SUM and A1 of ZAP.

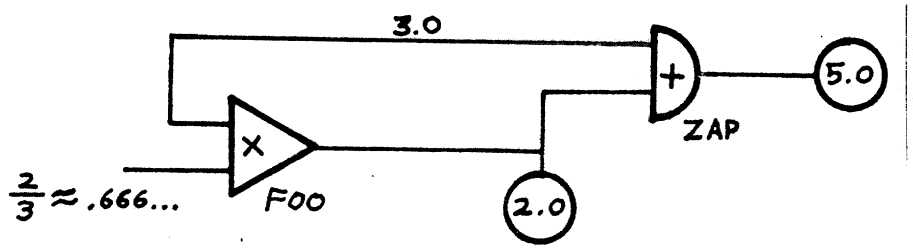
```
==> (premises (>> m2 foo))
(>> SUM ZAP) = 5.0
(>> A1 ZAP) = -2.0
```

If we change a premise all conclusions which depended upon it are automatically retracted and new conclusions are drawn. In a large network only a few values may depend on any one premise. The constraint language interpreter retracts only those values and only the incremental new deductions are made.

```
==> (change-parameter (>> a1 zap) 2.0)
2.0

==> (what-is (>> a2 zap))
(>> A2 ZAP) = 3.0

==> (what-is (>> m2 foo))
(>> M2 FOO) = 0.666666664
```



We cannot so easily change a parameter whose value is a consequence of other known facts. One of the premises upon which the changing parameter depends must be abandoned. The system automatically chases down only the relevant premises and gives us a choice of which we want to retract.

```
==> (change-parameter (>> a2 zap) 19.0)
Which of the following assumptions will you change?
   1 (>> A1 ZAP) = 2.0
   2 (>> SUM ZAP) = 5.0
ANSWER:
```

Before allowing the system to change anything, we may want to investigate the repercussions of changing it. (Note that the prompt has changed to "-->" indicating that the system is waiting for an answer to its question.)

```
--> (results (>> sum zap))
Rule (>> RULE#3 ZAP) got (>> A2 ZAP) = 3.0

--> (results (>> a1 zap))
Rule (>> 1<=2) got (>> PRODUCT FOO) = 2.0
Rule (>> RULE#3 ZAP) got (>> A2 ZAP) = 3.0
```

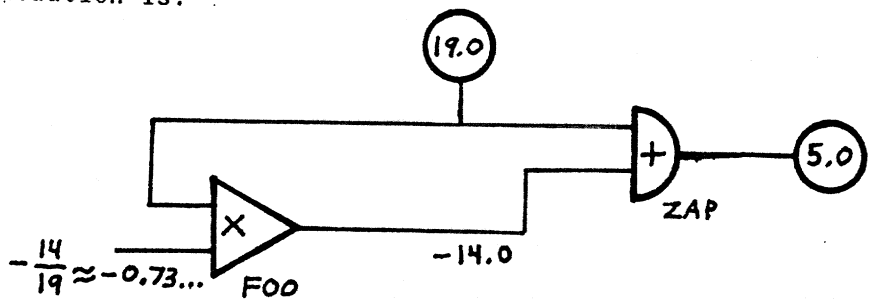
We decide to retract assumption #1, that the A1 of ZAP is 2.0.

```
--> (answer 1)
19.0
```

The system finished changing the parameter to 19 and restored the prompt to "==>". The new value of the M2 of FOO is now:

```
==> (what-is (>> m2 foo))
(>> M2 FOO) = -0.7368421
```

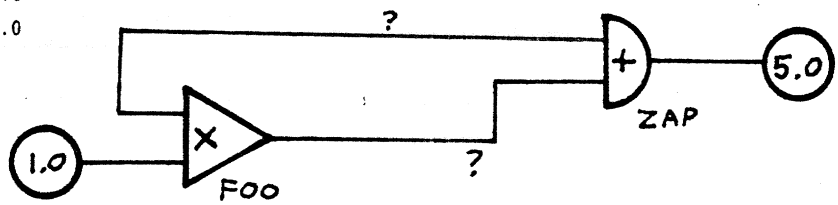
The new situation is:



Local constraint propagation cannot solve every problem of assigning values to cells in our network. Sometimes a more global analysis is necessary. For example, if we change the value of the M2 of FOO to 1.0 and we release the premise concerning the value of the A2 of ZAP, the only consistent value that either the A1 or the A2 of ZAP can take on is 2.5.

```
==> (change-parameter (>> m2 foo) 1.0)
Which of the following assumptions will you change?
   1 (>> A2 ZAP) = 19.0
   2 (>> SUM ZAP) = 5.0
ANSWER:

--> (answer 1)
1.0
```

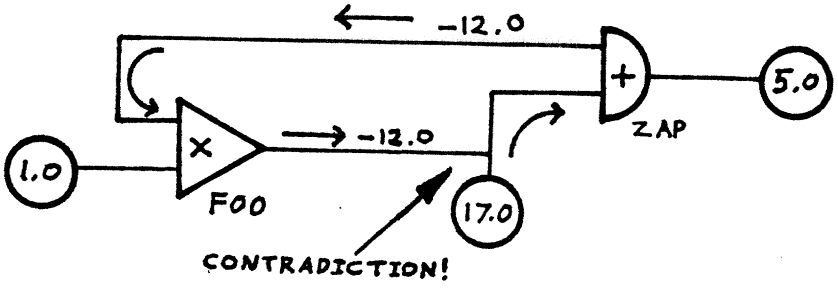



```
==> (what-is (>> a1 zap))
Sorry, I don't know it.  I need:
   (>> PRODUCT FOO)
to use rule: (>> 2<=1)
or, I need:
   (>> A2 ZAP)
to use rule: (>> RULE#2 ZAP)
```

The problem is that neither the adder nor the M2 has enough information to make any deductions by itself. However, if they could collaborate, they could come to a correct conclusion. We will discuss this later.

Inasmuch as the constraint network cannot determine the PRODUCT of FOO, we will specify a value explicitly.

```
==> (change-parameter (>> product foo) 17.0)
Contradiction...disputants are:
   (>> PRODUCT FOO) = 17.0
   (>> CHALLENGER: RULE#3 FOO) = -12.0
Choose an assumption to change:
   1 (>> M2 FOO) = 1.0
   2 (>> SUM ZAP) = 5.0
ANSWER:
```

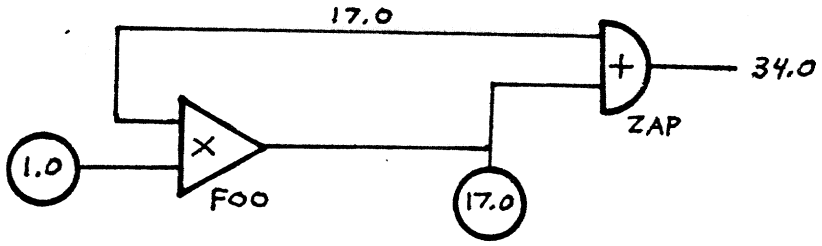


Unfortunately, the very act of making the assumption that the PRODUCT of FOO is 17.0 allows the deduction that the PRODUCT of FOO must (also) be -12.0, given the other assumptions in force. This immediate contradiction is noticed, and the system presents us with a choice of actions.

```
--> (answer 2)
17.0
```

We have chosen to discard assumption 2, and the network readjusts itself.

```
==> (what-is (>> sum zap))
(>> SUM ZAP) = 34.0
==> (what-is (>> a2 zap))
(>> A2 ZAP) = 17.0
```



Now we have seen the rule of composition by which constraints are combined into networks of constraints by identification of parts. We have also seen how it is possible to compute with constraints. The constraint interpreter maintains the "dependencies" which describe how conclusions are drawn from their antecedents. Besides the fact that the dependencies are important to the system for efficient computation (as we shall see), they are useful for debugging purposes, and for helping a user understand the

behavior of his constraint network. We believe that this is a good idea. A discipline should be developed for construction of systems so that they are responsive to their users. They must be responsible for their conclusions and able to explain how the conclusions were derived. This is increasingly important as systems become more complex, thus exceeding the understanding of any one person.

Abstraction and Hierarchy

Using the "==" construction, we can build networks of adders and multipliers which are arbitrarily large, and thus arbitrarily complex. In order to deal with this complexity, we need a way to break up large networks into meaningful pieces. If we are lucky, many of these pieces will be the same, and can be considered single building blocks at a higher conceptual level. We need a way to define such building blocks in terms of smaller ones.

We would like these compound building blocks to behave in the same way as the primitive entities of the language. This allows us to combine such building blocks using the same construction methods. In this way we can build arbitrarily complicated compound objects in a hierarchical manner. The hierarchy allows the complexity at any one level to be limited.

In languages which permit this kind of hierarchical definition there is a rule of abstraction, which specifies how a compound object is to be treated as a simple one. Such a rule must specify a particular combination of simpler objects which is to be abstracted, and an interface between the specified combination and the external appearance of the abstraction as a single object.[Lambda]

The constraint language provides a rule of abstraction which allows us to associate a name with a pattern designating component constraints and "==" specifications for linking them. When a pattern is instantiated, the designated components are recursively instantiated and then linked according to the "==" specifications.

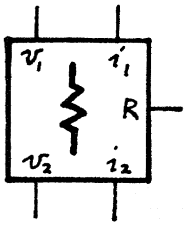
For example, we can combine adders and multipliers to produce a kind of compound contraint called a "resistor". An (ideal) resistor is essentially an object which enforces numerical constraints among two node potentials, two currents, and a resistance:

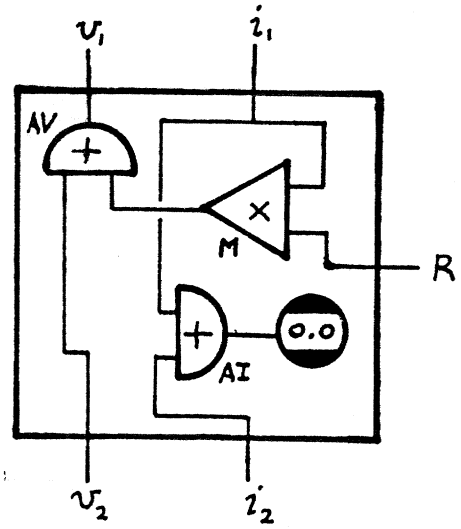
$$(v_1 - v_2) = i_1 R$$

$$i_1 + i_2 = 0$$

We notate this as follows:

```
==> (constraint resistor
                ((v1 number)
                 (v2 number)
                 (i1 number)
                 (i2 number)
                 (resistance number)
                 (av adder)
                 (ai adder)
                 (m multiplier))
                (== v1 (>> sum av))
                (== v2 (>> a1 av))
                (== (>> a2 av) (>> product m))
                (== i1 (>> m1 m))
                (== resistance (>> m2 m))
                (== i1 (>> a1 ai))
                (== i2 (>> a2 ai))
                (constant (>> sum ai) 0.0))
RESISTOR
```





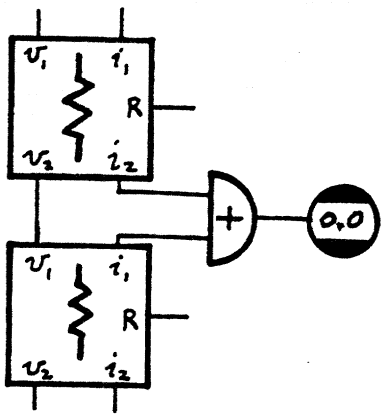
The keyword "constraint" is followed by a name, a list of component names and types, and a set of linkages. The expression (constant <something> <value>) is an abbreviation for forcing <something> to have the given computational value. {ThingLab}

With this definition we can make a resistor in the same way that we make an adder:

```
==> (create r43 resistor)
R43
```

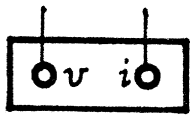
This causes R43 to be the name of an instance of the pattern RESISTOR. In instantiating the pattern, instances of components such as adders and multipliers are also effectively created. We can use instances of resistor to build a voltage divider:

```
==> (create r44 resistor)
R44
==> (create kcl1 adder)
KCL1
==> (== (>> v2 r43) (>> v1 r44))
IDENTITY
==> (== (>> i2 r43) (>> a1 kcl1))
IDENTITY
==> (== (>> i1 r44) (>> a2 kcl1))
IDENTITY
==> (constant (>> sum kcl1) 0.0)
0.0
```



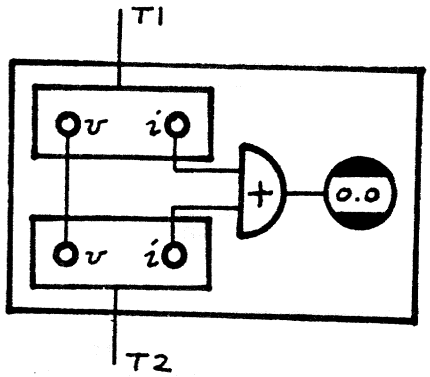
It is silly to use adders and numerical connectors to hook together resistors. We should not be mixing conceptual levels. In the world of electrical circuits, elements like resistors have terminals which connect to nodes. A terminal has a potential ("voltage") on it and a current into it, packaged up together. A node connects two or more terminals so as to constrain their potentials to be the same, and the sum of the currents into them to be zero.

```
==> (constraint terminal ((v number) (i number)))
TERMINAL
```



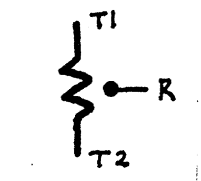
The "terminal" constraint is trivial. All it does is package up two numbers so that they can be referred to as a unit.

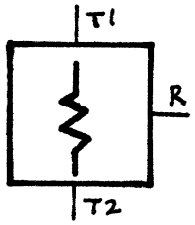
```
==> (constraint 2-node
                ((t1 terminal)
                 (t2 terminal)
                 (kcl adder))
                (== (>> v t1) (>> v t2))
                (== (>> i t1) (>> a1 kcl))
                (== (>> i t2) (>> a2 kcl))
                (constant (>> sum kcl) 0.0))
2-NODE
```



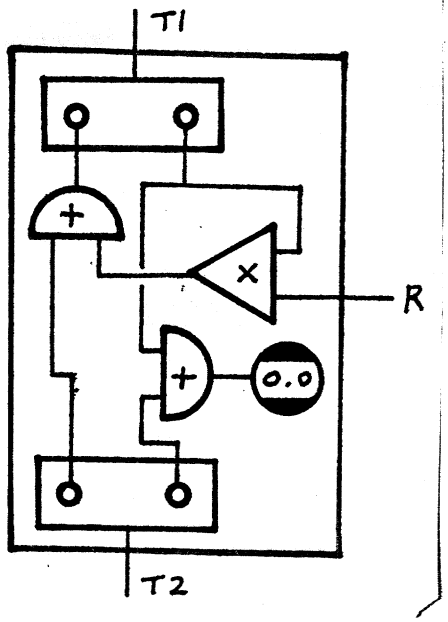
A "2-node" is a two-terminal node. It has an adder for enforcing the sum-of-currents constraint (Kirchoff's Current Law). The equal-potentials constraint (Kirchoff's Voltage Law) is enforced by direct identification.

```
==> (constraint resistor
                ((t1 terminal)
                 (t2 terminal)
                 (resistance number)
                 (av adder)
                 (ai adder)
                 (m multiplier))
                (== (>> v t1) (>> sum av))
                (== (>> v t2) (>> a1 av))
                (== (>> a2 av) (>> product m))
                (== (>> i t1) (>> m1 m))
                (== resistance (>> m2 m))
                (== (>> i t1) (>> a1 ai))
                (== (>> i t2) (>> a2 ai))
                (constant (>> sum ai) 0.0))
```





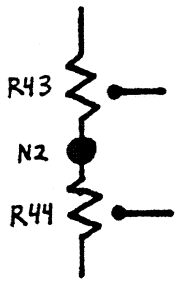
RESISTOR



This definition is as before, except that terminals are used.

Now we can simply create two resistors and a 2-node, and connect them up by identifying the terminals of resistors with terminals of the node.

```
==> (create r43 resistor)
R43
==> (create r44 resistor)
R44
==> (create n2 2-node)
N2
==> (== (>> t1 r44) (>> t2 n2))
IDENTITY
==> (== (>> t2 r43) (>> t1 n2))
IDENTITY
```



In the constraint language, it is possible to identify compound constraints, using "==", in exactly the same way that primitive objects like numbers are identified. The meaning of equality for compound objects is that two objects are equal if their corresponding components are equal. Thus when two terminals are made equal, their voltages are identified, and so are their currents.

(In the examples involving electrical components we have been, and will be, carefully skirting a difficult issue regarding the directions of currents. In the constraint language, things "connected together" are identified as being the same thing. In the electrical world, while electrically connecting two electrical terminals makes their voltages the same (Kirchoff's Voltage Law), it does not make their currents the same. The currents are instead subject to the more complex constraint that they must sum to zero. This is the reason we use nodes as an intermediate

connector for enforcing Kirchoff's Current Law. Nodes must be used carefully to keep the current directions straight. Two constraints describing electrical devices cannot be directly connected, but must be connected through a node. Additionally, two nodes cannot be directly connected, except through a constraint representing a device. The examples we give here are correct, but it is very easy to misuse the language.)

We can use this structure to compute parameters of the circuit. The two resistors form a voltage divider. We can, for example, specify the voltages at the ends and middle of the divider, and the resistance of one resistor. The algebraic constraints of the circuit will then permit the deduction of the resistance of the other resistor:

```
==> (set-parameter (>> v t2 r44) 0.0)
0.0
==> (set-parameter (>> v t1 r44) 3.0)
3.0
==> (set-parameter (>> v t1 r43) 10.0)
10.0
==> (set-parameter (>> resistance r44) 9.0)
9.0
==> (what-is (>> resistance r43))
(>> RESISTANCE R43) = 21.0
```

This is similar to computations we have already seen. The "real work" is being done by adders and multipliers which are part of the description we have provided of resistors.

As before, there can be problems with not being able to compute something with constraints for which we in principle have enough information. For example, given the voltage at the top and the bottom, and the resistances of both resistors, it should be possible to compute the voltage at the midpoint (the "divided voltage"):

```
==> (forget-parameter (>> v t1 r44))
NIL
==> (set-parameter (>> resistance r43) 21.0)
21.0
==> (what-is (>> v t1 r44))
Sorry, I don't know it.  I need:
   (>> V T2 N2)
to use rule: (>> 1<=2)
or, I need:
   (>> SUM AV R44)
to use rule: (>> 1<=2 R44)
```

Forgetting the voltage at the midpoint caused the system also to forget the resistance of R43, which had been deduced from this voltage. Reinstating the resistance value should permit the deduction of the midpoint voltage,

but as before this cannot be done locally by the constraints. There are several ways around this problem. One is to introduce multiple views of the circuit; we will examine this technique in the next section. Another is to use symbolic algebra in the computations; this will be examined later.

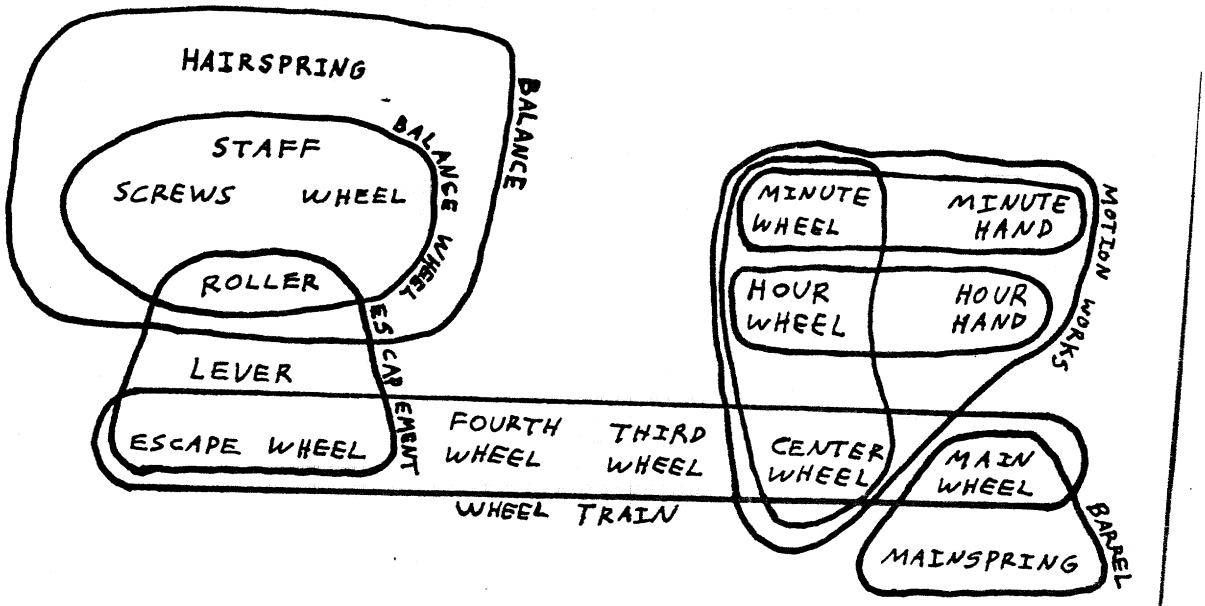
Almost-Hierarchical Systems

To be powerful, a description must impose a conceptual structure on a system for the purpose of limiting the complexity involved in understanding or designing it. If a system cannot be immediately understood in its entirety (by "Gestalt") then it must be understood in pieces. The descriptive structure directs attention to sets of components of the system which together constitute a single conceptual piece.

Often a system can be partitioned into pieces which are more or less disjoint and which together cover the entire system. The total system can be understood by understanding the pieces and by understanding the composition by which the pieces constitute the system. Similarly, each piece may be similarly partitioned. In this way we derive a single tree-like decomposition of the system. Such a tree we call a hierarchy.

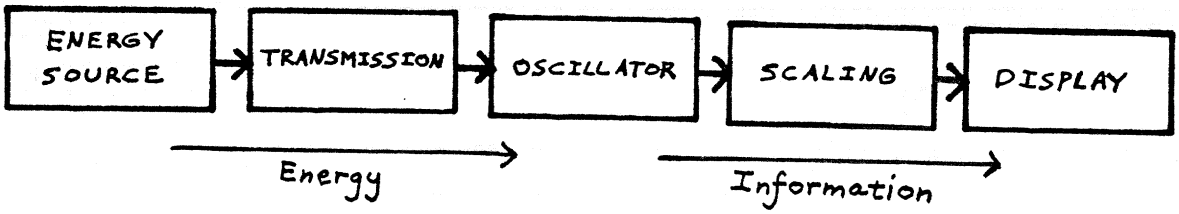
It is likely, however, that at any stage there is more than one useful partitioning of a piece. If so, then a single hierarchy does not suffice to indicate all the conceptual pieces of interest in the system. Pieces whose sub-pieces are localized within one hierarchy will probably have its sub-pieces widely dispersed throughout another.

For example, a mechanical timepiece has among its major parts a mainspring, several wheels or "gears" (main, center, third, fourth, hour, minute, and escape wheels), a lever, a balance/hairspring assembly, and hands. These parts are usually grouped into sets which are physically connected: the mainspring and main wheel together form the barrel; the center, hour, and minute wheels plus the hands form the motion works; the main, center, third, fourth, and escape wheels form the wheel train; and so on. Also, some of the parts are themselves complicated assemblies, for example the balance.

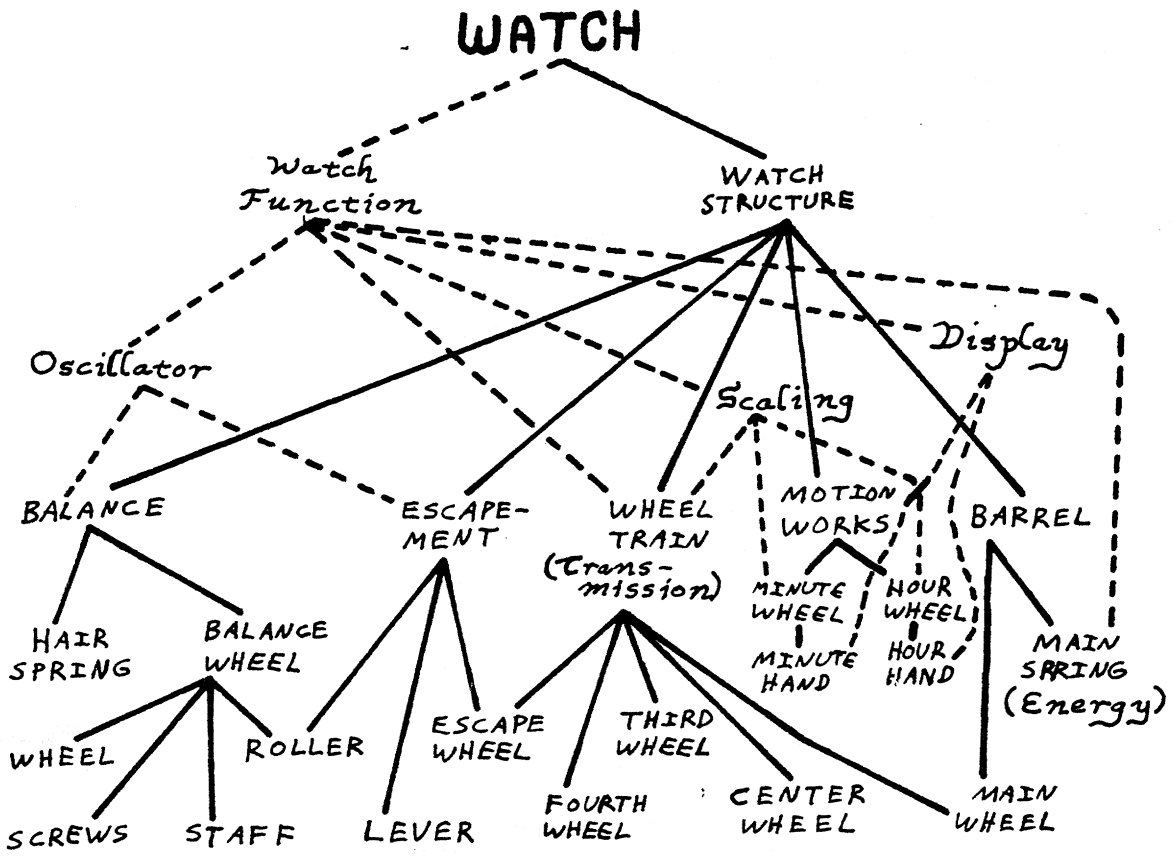


Notice that there is some overlap between the main structural groupings where they interface.

A timepiece also may be <u>functionally</u> described as a fivefold decomposition:



Conceptually these five functional parts are disjoint. In every real mechanical timepiece, however, the same physical part may serve in more than one functional unit. The wheel train which transmits energy from the mainspring to the escapement also performs part of the scaling between the escapement's oscillations and the display (dial); but the wheel train is <u>not</u> part of the oscillator module which appears between the transmission and scaling modules. The escape wheel is part of both the oscillator module and the wheel train. Other parts are also shared among modules.



In this diagram, the structural and functional hierarchies are (partially) shown together. Notice that each not is a strict hierarchy, but almost is; and that the two almost-hierarchies are not the same, but are similar. In some places a single unit in one hierarchy is the same as a single unit in another (for example, the wheel train and transmission coincide). In other places what is a single unit in one hierarchy is spread out in the other (for example, the single functional idea of scaling is spread out in the structural hierarchy, while the wheel train and motion works structural units each have various purposes in the functional hierarchy).

It would be possible to design a timepiece which exhibited much less structural overlap between functional units, so that the structural and functional hierarchies would be almost identical. While such a timepiece would certainly exhibit "structured" design, in the sense of "structured programming", it would be much less reliable (due to unnecessary duplication of parts to avoid overlap), and would be very difficult to make small enough to ride comfortably on one's wrist.

Other engineered systems, such as electrical circuits and computer programs, exhibit a similar sharing of parts among uses. A strictly hierarchical description can be only an approximation to the true structure

of an object. Conversely, when designing an object, a good engineer may start with a hierarchical plan, but will then slightly destroy the hierarchy when interfacing the substructures. This occurs because any particular hierarchical description emphasizes some groupings of features at the expense of other equally valid or important ones. A given hierarchy may be violated because of constraints not visible within that hierarchy.

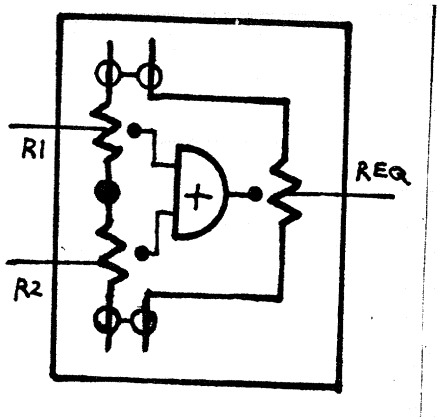
A good description explicitly acknowledges the sharing of structure in engineered devices. This can be captured as a locally hierarchical structure in which each feature is described and related to "lower-level" features which implement that feature, even though those sub-features may be spread throughout some other (also locally hierarchical) part of the description and used to implement other features as well.

The constraint language naturally permits one to express almost-hierarchical descriptions, because its rules of composition are explicitly stated in terms of the sharing of parts. The rule of abstraction is hierarchical, but the rule of composition permits one to violate the hierarchy in any desired manner.

In fact, we have already used this feature to describe the connection of two resistors. One terminal of each resistor is shared with a terminal of the node. Terminals are not very interesting as they have no non-trivial internal structure. However, let us extend that example to demonstrate a more interesting almost-hierarchy.

We first express the idea that two resistors in series are equivalent to a single resistor:

```
==> (constraint series-resistors
                ((r1 resistor)
                 (r2 resistor)
                 (req resistor)
                 (a adder))
                (== (>> t1 r1) (>> t1 req))
                (== (>> t2 r2) (>> t2 req))
                (== (>> a1 a) (>> resistance r1))
                (== (>> a2 a) (>> resistance r2))
                (== (>> sum a) (>> resistance req)))
SERIES-RESISTORS
```



(This does not say that (>> t1 req) is electrically connected to (>> t1 r1). It says that (>> t1 req) is (>> t1 r1). This differs from the situation where two resistors are connected through a node. Here we are talking about two views of the same terminal.)

This definition does not express the fact that we ordinarily apply it only to resistors which share a common node. This is a restriction on usage of the constraint, rather than a restriction to be enforced by the

constraint. If it is used properly, however, it will enforce the desired series constraint. We now construct an instance of the constraint, and use it to describe the resistors R44 and R43 connected earlier.

```
==> (create sr1 series-resistors)
SR1
==> (== (>> r1 sr1) r43)
IDENTITY
==> (== (>> r2 sr1) r44)
IDENTITY
```

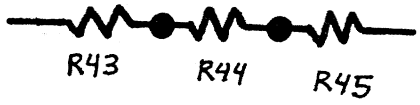
Now there are two simultaneous, redundant descriptions of what is going on between the upper and lower terminals of the circuit. Neither description is regarded as being "more real" than the other; we merely say that one or the other is more useful for some purpose. Both descriptions, as one resistor and as two, are equally valid. We can speak of `(>> resistance req sr1)`, the resistance of the resistor in the single-resistor description, just as well as of `(>> resistance r43)` or `(>> i t1 r44)`, which are quantities of the two-resistor description.

These redundant descriptions have important computational applications. For example, in the voltage divider circuit the alternative description of the divider as a single resistor permits the deduction of the current through the divider. This in turn permits the deduction of the voltage at the midpoint!

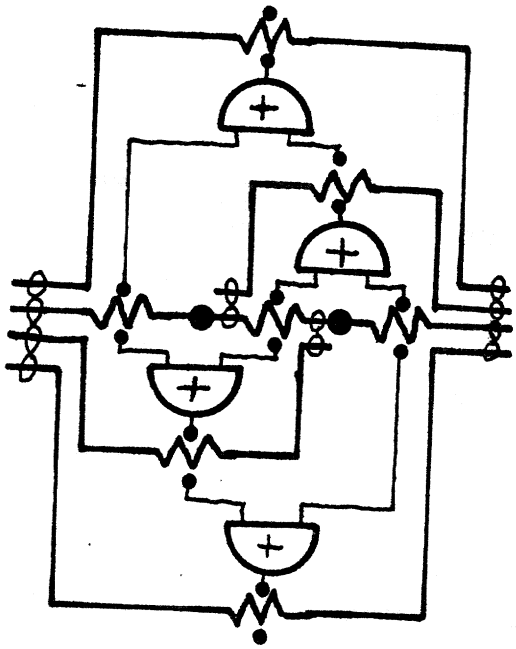
```
==> (what-is (>> v t1 r44))
(>> V T1 R44) = 3.0
```

What we have done here is to introduce an alternative point of view of the circuit. While in principle it contains no extra information, the new viewpoint is better organized for certain purposes (such as determining the current through the divider). The constraint language permits us to introduce many such redundant viewpoints so that they can cooperate in solving a problem.

Let us add another resistor R45 to our circuit:



We can describe various series combinations within this circuit:



In this manner the flat three-resistor circuit can be hierarchically decomposed in at least three different ways: (1) as three separate resistors; (2) as the series combination of the upper two, in series with the lowest; (3) as the uppermost, in series with the series combination of the lower two. In a given circumstance one point of view may be more useful than another. Moreover, the various points of view do not fit neatly into a single hierarchy. They can be organized into an almost-hierarchy.

Constraints and Algebra

It might seem that in writing down and using constraints we are just writing algebraic expressions in a different and not even convenient form. This is true to some extent, but it is not the entire picture. The constraint language is a kind of "structured algebra" in that we have almost-hierarchical structures which allow us to manufacture and combine systems of algebraic constraints to make larger systems. Additionally, we will see that networks of constraints are more fundamental than the algebraic notation used to describe them, in that there may be many equivalent sets of equations which describe any given constraint diagram, but there is precisely one constraint diagram for any given set of equations.

Algebra has two components of interest. One is a notation for expressing constraints, and the other is a set of transformation rules which permit the derivation of consequences of the constraints.

The reader may have noticed that the constraint language is rather

verbose. In defining the numerical constraints in the RESISTOR constraint definition, we had to write out eight lines:

```
(== (>> v t1) (>> sum av))
(== (>> v t2) (>> a1 av))
(== (>> a2 av) (>> product m))
(== (>> i t1) (>> m1 m))
(== resistance (>> m2 m))
(== (>> i t1) (>> a1 ai))
(== (>> i t2) (>> a2 ai))
(constant (>> sum ai) 0.0))
```

Using an algebraic notation, we might have written simply:

```
(>> v t1) - (>> v t2) = (>> i t1) * resistance
(>> i t1) + (>> i t2) = 0.0
```

Algebraic convention normally dictates that simpler variable names than "(>> v t1)" be used. We have used the same long names in the algebraic equations not only for the sake of a "fair" comparison, but because names of this form are critical to the almost-hierarchical composition of the constraints.

Algebraic notation is expression-oriented. It achieves conciseness through the use of functional composition. The relationship of arguments to function is implicitly expressed by position, permitting the elimination of explicit description of the connections. For example, in expressing an addition constraint in the constraint language, we explicitly mention the three "pins" A1, A2, and SUM of the adder, and explicitly specify what they are linked to. In algebraic notation we simply write "a + b". The "+" denotes an adder constraint. The three connections to the adder constraint are expressed by position. Whatever is to the left of the "+" is connected to the A1 pin; whatever is to the right of the "+" is connected to the A2 pin; and the SUM pin is connected to the pin of whatever operator of which the entire expression stands as an argument. If we write "(a + b) * c", then the SUM pin of the adder constraint is connected to the M1 pin of the multiplier constraint.

The connections indicated in an algebraic expression form a tree, with connections to identifiers at the leaves, and a single loose connection coming "out the top". To connect the loose ends of two expressions, we use the equality symbol "=". An equation, unlike an expression, is a constraint network with no loose ends (or rather, the loose ends are explicitly indicated by the presence of identifiers).

The advantages of algebraic notation are obvious. What do we pay for conciseness? Asymmetry. To a mathematician, the following three statements say the same thing:

$$E = K + U \qquad K = E - U \qquad U = E - K$$

Each "binary" operator must in principle have three forms, one for each of the three connections that may come "out the top" as opposed to being connected to "arguments". (In the case of addition and subtraction two forms suffice because one of them is commutative. Consider, however, the triplet of equations

$$x = y^z \qquad y = \sqrt[z]{x} \qquad z = \log_y x$$

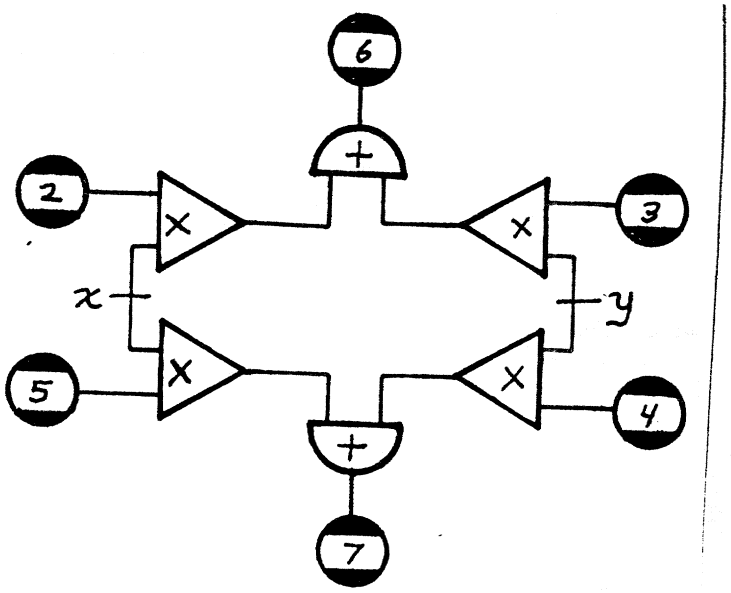
each of which expresses the same constraint in a different form. More generally, an n-ary operator with no symmetries on its pins requires n+1 distinct forms.) The constraint language preserves symmetry of expression and economizes on primitive notations: both additions and subtractions are expressed by the same constraint. The cost of this symmetry and economy in the constraint language is verbosity.

Algebra provides rules for transforming among equivalent representations of a constraint. Given this, in some sense it doesn't matter how they are written down. Equations express truth.

Given a set of equations, we can easily draw the (unique) constraint diagram which they represent. For example, for the set

$$2 * X + 3 * Y = 6 \qquad 5 * X + 4 * Y = 7$$

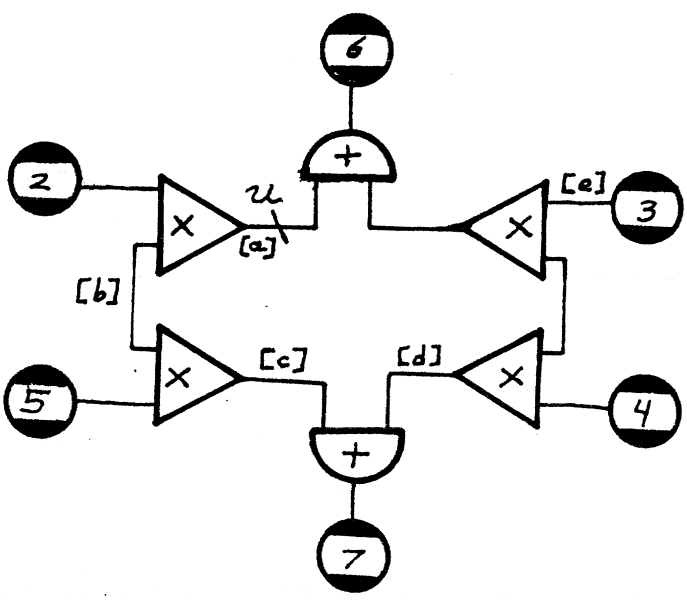
we draw the diagram



Each equation represents a piece of the network which is an unrooted tree, the result of connecting two rooted trees at their roots. The purpose of the identifiers X and Y in the equations is to cut the loops in the network in order to make them printable as (necessarily tree-like) expressions.

When the separate tree-like pieces are printed, the identifiers indicate the cross-branch and inter-tree connections.

For any given network, there may be more than one way to cut the loops. For example, we could use another variable U:



We can then write this cut of the network as

```
6 = U + 3 * ((7 - 5 * (U / 2)) / 4)
```

Even for a given assignment of identifiers, one can write many different equations by choosing which (two-ended!) connection to use for the equality; the two parts of the equation tree are considered to be rooted at the ends of the chosen connection. Using the connections marked {a}, {b}, {c}, {d}, {e} we get these equations:

```
{a}  U = 2 * ((7 - 4 * ((6 - U) / 3)) / 5)
{b}  U / 2 = ((7 - 4 * ((6 - U) / 3)) / 5)
{c}  5 * (U / 2) = (7 - 4 * ((6 - U) / 3))
{d}  (7 - 5 * (U / 2)) = 4 * ((6 - U) / 3)
{e}  3 = (6 - U) / ((7 - 5 * (U / 2)) / 4)
```

All of these equations represent the same network of relationships.

It is of course possible to use more identifiers than necessary. One (U) suffices for our example, and it could have been put in any of six places. We originally derived the diagram from two equations in two identifiers X and Y. We could use more than two if we desired to, for some reason; we could also assign more than one identifier to a connection, and equate the redundant identifiers. Taken to the extreme, this approaches the connection specifications of the constraint language, wherein each quantity has multiple names which are equated.

In some diagrams a single connection may have more than two "ends"; i.e. the same quantity is constrained in more than two ways. This occurs in the RESISTOR definition above. Now the implicit connections in expression notation have only two ends. When a diagram with many-ended connectors is to be printed as a set of equations, such connectors must have identifiers assigned to them, because only identifiers can express multiple-ended connections in algebraic notation. After such identifiers have been assigned, additional ones may or may not be needed to cut remaining loops.

The various ways of algebraically notating a set of relationships are equivalent. A network of relationships simply exists. It is in this sense that we mean that constraint diagrams are more fundamental than sets of equations. Many algebraic laws (but not all) simply provide for transformations among sets of equations which represent the same diagram.

When we begin to use the network for some computational purpose, however, then the point of view becomes important. If we wish to compute some "output" value(s) given certain "input" values, where all the values are related by some network of constraints, we must organize an information flow within the network. Intermediate computations must proceed along an acyclic path from inputs to output(s).

So-called "algebraic" programming languages find algebraic notation a convenience because the tree implicit in the functional composition is guaranteed acyclic, and there is an asymmetry to each connection (from "out the top" to "into an argument slot") which can be construed to imply the direction of information flow. The notation therefore requires the programmer not only to express relevant relationships but also simultaneously to specify the particular computational use to which the relationships will be put. Thus, for example, Ohm's Law is the equation "V = I R", but the user of an "algebraic" programming language cannot simply write this algebraic equation; he must put it in a particular form (for example, "I := V / R") which explicitly directs the computation.

This may serve as a definition of the distinction between "imperative" and "declarative" languages. An imperative language requires the programmer to explicitly break loops in the network of constraints, and to explicitly organize the information flow within the network. A declarative language requires only the statement of relevant relationships, and the computational organization is specified separately or performed more or less automatically.

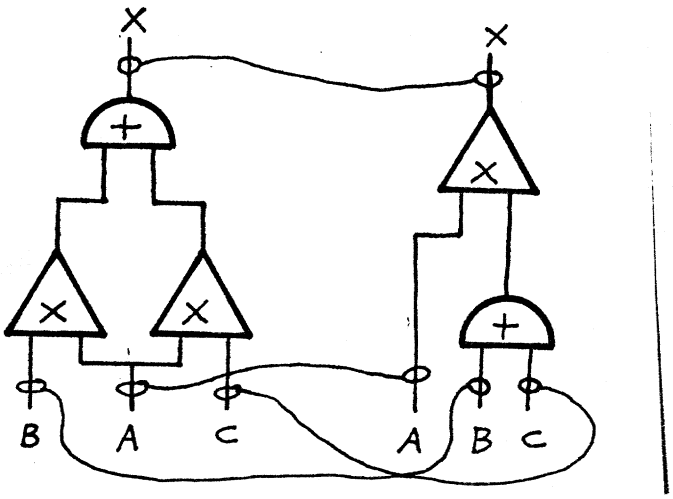
Besides requiring the programmer to specify the computational flow as a tree, most "algebraic" programming languages require expressions to denote single results.(Multiple Values) For example, there is generally no way to define an integer division "function" which returns both quotient and remainder (which is a shame, because typical algorithms for computing either actually compute both, particularly those used in many contemporary

computers) and embed a call to such a "function" in an expression. This is an inherent property of trees.

We pointed out earlier that algebra provides transformation rules for "shifting perspective" on a relationship, for example transforming "E = K + U" into "K = E - U". Not all algebraic transformations are this simple. Some do not merely view the same network in a different way, but produce a different network which has the same meaning by virtue of non-trivial relationships between the meanings of the component constraints. A good example of this is the distributive law of multiplication over addition:

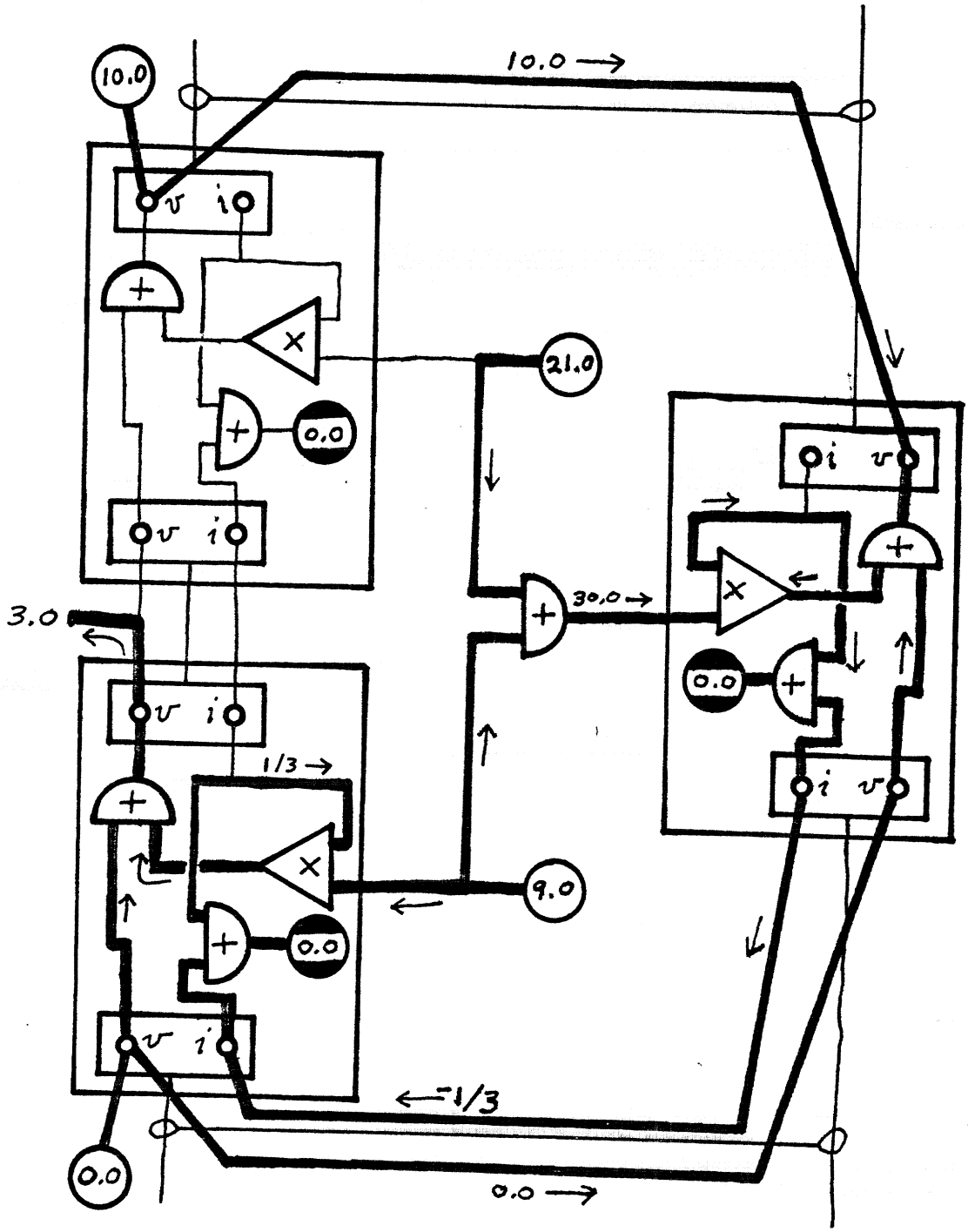
$$X = A\,B + A\,C \quad <=> \quad X = A\,(B + C)$$

The networks for these two equations look like this:



These networks are topologically distinct: one has a loop, and the other does not. Consider now trying to compute A given X, B, and C. The second network can be used straightforwardly, because a spanning tree can be imposed on it, rooted at A, with all leaves given, specifying a complete computational flow. Such a tree cannot be imposed on the first network. Any spanning tree rooted at A has A as a leaf (a spanning tree rooted at a connection extends out only one end of the connection), and so is unsuitable for computing A.

This is why the voltage divider failed to compute the midpoint voltage given the endpoint voltages and the two resistances. The constraints imposed by the resistor definition contained loops which prevented the system from finding an effective computational flow within the network. Imposing the SERIES-RESISTORS equivalence introduced extra paths which provided an effective spanning tree for the network in terms of the particular givens.



Similarly, when we "solve an equation" by using the distributive law, we in effect impose another piece of network on the original network which preserves the semantics of the relationships but permits a spanning tree to be found.

Computing with Constraints

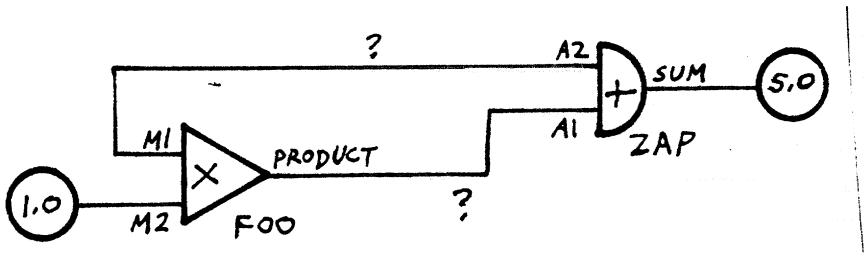
All of the computations we have performed with constraint networks can be described as simple propagation of known values. Values are propagated through primitive constraints and through equalities by "one-step deductions".(Propagation) Primitive constraints are processes containing local cells which "continually" monitor the cells. A cell may at any time have a value associated with it. A primitive constraint will notice if any of its cells gets a value, and if it can locally determine the value of one of its cells from the values of its other cells, it will associate the deduced value with that cell. A cell can also get a value by being equal to another cell which gets a value. Finally, a cell can be arbitrarily assigned a value by the user or by a CONSTANT declaration (cf. RESISTOR above).

If a cell has an associated value, it also has associated with it a reason.(Dependencies) If the value was obtained from other cells through a primitive constraint, then the reason mentions those other cells and the particular rule of the primitive constraint used to make the deduction. If the value was obtained through an equality with another cell, the reason mentions that cell. If the value was arbitrarily assigned, the reason points to the user or to the system (CONSTANT). These reasons ("dependencies") are easily constructed as the deductions are made; they are essentially simple markers indicating the direction of information flow within the network. These markers can be traced at any time for such purposes as finding the antecedent assumptions of a deduced value, or locating the consequences of a value. These may be in turn used for explanation or incremental forgetting of a user-supplied assumption and all of its consequences (as we have seen). In addition, if one traces from a value back to its antecedent assumptions, the part of the network traced, with the directionality imposed by the dependency markers, constitutes a formula for computing the value.(Compiling Constraints)

It is possible that a cell be assigned a value for more than one reason. If two values collide at a cell they must be the same. If they are not, and if the network represents a satisfiable set of constraints, there is an inconsistent set of assumed values. The dependencies can be used to determine the subset of the set of assumed values is inconsistent. This can be helpful in dealing with very large networks with lots of assumed values, where any one contradiction typically depends on only a small subset of the many assumptions.

In a satisfiable constraint network, the values which will be associated with cells are independent of the order of propagations through the network. Thus we may think of them as occurring in parallel, though they are implemented by means of a queue or agenda in our current system.

As we have seen, there are simple networks for which propagation fails to assign all of the values one might desire. For example:



We have already discussed how redundant descriptions may be used to bypass such loops by superimposing an equivalent tree structure. Another, related strategy involves the use of symbolic algebraic manipulation. If we enable the primitive constraints to propagate symbolic expressions as well as numerical values, we can use powerful algebraic manipulators to propagate past loops. If for example, in the problem above, we put the symbolic value "X" on (>> A1 ZAP) then ZAP may deduce that its A2 is 5-X. "Simultaneously" FOO may deduce that its M1 is X/1. These values are on cells declared to be equal; thus an equation may be formed, X/1 = 5-X, and solved algebraically. Once the value of X has been determined, then the numerical values for the propagated symbolic expressions can be computed (or at least expressions involving X and possibly other variables can perhaps be simplified).

Algebraic techniques of this sort were used in such systems as EL, ARS, and SYN.{Algebraic Propagation} It is easy to see, however, that propagation of symbolic expressions in effect makes copies of tree-like portions of the network; each expression is a history of the portion of the network through which its pieces were derived. The algebraic techniques applied to these expressions might as well have been applied directly to the network. Instead of applying the distributive law to an algebraic expression, for example, one might as well just attach that extra point of view to the network. The implications of this idea have just barely begun to be explored.

Acknowledgements

We would like to thank Paul Penfield for getting us started on this paper. Jon Allen, Alan Borning, Richard Brown, Jon Doyle, and Paul Penfield read and commented on draft versions. One of us (Steele) is supported by a Fannie and John Hertz Foundation graduate fellowship.

Notes

{Algebraic Propagation}. We have used propagation of algebraic expressions in our work on analysis and synthesis of circuits.{Propagation} Although we have used algebraic manipulation techniques (extracted from MACSYMA [MACSYMA]) far more powerful than people generally use, we have found that

these techniques are not, by themselves, powerful enough to solve many interesting problems which people can solve. People generally solve these problems by organizing the solution so that only simple algebra is required (often by using canned theorems, whose proofs perhaps required tremendously complex algebra (done once and for all by a clever person whose name is probably attached to the theorem), but whose application does not require complex algebra). To avoid tremendous symbolic computations, computers must also have good methods of avoiding doing all except the most trivial algebra problems. We have used multiple redundant descriptions [Sussman 1977] to encapsulate ways of looking at a problem which are organized so as to obviate the need for extensive algebraic manipulation.

{Compiling Constraints}. The idea of extracting procedures for computing specific values from a constraint network is being pursued by Richard Brown [Brown 1978]. Borning has also done work in this area.(ThingLab)

{Dependencies}. TOPLE [McDermott 1974] was an early attempt to record the interactions among deductions for the purpose of maintaining consistency in a data base when newly introduced facts conflicted with existing ones. The SRI Computer Based Consultant [Fikes 1975] made use of dependencies to determine the logical support of facts, but did not use them to control search. MYCIN [Shortliffe 1974] [Davis 1976] [Shortliffe 1976] used dependency information to produce explanations, but did not use it for any control purposes. EL [Sussman & Stallman 1975] used dependencies to produce explanations and also to limit the recomputation required in response to incremental changes in the assumptions. Stallman and Sussman described a general means of limiting combinatorial search by the analysis of dependency chains and the use of multiple logical supports for facts [Stallman & Sussman 1977]. This method, called "dependency-directed backtracking", was independently discovered by Jim Stansfield (unpublished Ph.D. thesis draft, University of Edinburgh -- the idea was edited out of the final thesis for unknown reasons). This method is superior to the more familiar "chronological backtracking", introduced by Floyd [Floyd 1967] and best known for its use in AI languages beginning with PLANNER [Hewitt 1972] and Micro-PLANNER [Sussman, Winograd, & Charniak 1971], in that it avoids the irrelevant dependencies assumed by the system on the basis of accidental chronological orderings of assumptions [Sussman & McDermott 1972] [Stallman & Sussman 1977]. Doyle [Doyle 1977] developed a portable "Truth Maintenance System" which encapsulates a careful theory of dependencies, dependency-directed backtracking and non-monotonic inference. A simpler system for dependencies and backtracking was developed in [McAllester 1978]. McDermott and Doyle have produced an elegant semantic theory of non-monotonic inference which is described in [McDermott & Doyle 1978]. Phil London of University of Maryland has used dependencies to aid in keeping track of assumptions in the world model used by a problem solving system [London 1978].

{Elements}. Euclid had a point here.

{Lambda}. A simple example of such a rule is that of lambda-abstraction [Church 1941], in which a lambda-expression packages up a possibly compound expression and an interface specification in the form of a variable name (in LISP, variables names).

{Multiple Values}. There are a few languages which permit a routine to return more than one value (other than by side effect of a reference parameter as in FORTRAN). In SL5 every expression implicitly carries two values: an "ordinary" value and a success/failure flag. Languages such as POP2 and FORTH are stack-based, and routines both receive arguments and return values on the stack. APL and LISP do not have multiple-valued expressions. A common trick is to construct a data structure (an array or a list) which contains several distinct values to be returned; but the called routine must explicitly construct this structure, and the caller must then explicitly decompose it.

{Other Languages}. Some examples of languages which are less organized around the notion of procedure are such simulation systems as DYNAMO, GPSS, and SIMULA.

{Propagation}. "Propagation of constraints" was originally invented as a generalization of "Guillemin's method" of analyzing electrical ladder circuits. It was used in the analysis programs EL [Sussman & Stallman 1975] and ARS [Stallman & Sussman 1977], and in the synthesis program SYN [de Kleer & Sussman 1978]. The basic idea of the method was first described in [Brown 1975] as part of a method for localizing faults in electrical circuits. De Kleer also used propagation analysis in his fault localizer [de Kleer 1976]. Sutherland [Sutherland 1963] appears to have developed a similar technique (the "One Pass Method") for constraint satisfaction in Sketchpad.

{ThingLab}. Our method of specifying an object by a set of subparts and a set of identifications of sub-subparts is essentially the same as that independently developed by Alan Borning, a graduate student at Stanford University. In his Ph.D. thesis (forthcoming), entitled "ThingLab -- A Simulation Laboratory", Borning develops an interactive system (written in Smalltalk [Goldberg & Kay 1976]) for simulating the effects of constraints. ThingLab provides a beautiful and convenient graphics interface controlled by the constraint network. In contrast to the system described here, ThingLab does not retain dependency information, and uses relaxation techniques (rather than algebra or multiple redundant views) to deal with systems not directly amenable to simple propagation. In these respects ThingLab is very similar to Sketchpad [Sutherland 1963]. Unlike Sketchpad, ThingLab provides facilities for incrementally compiling constraints, as well as a non-graphical, programming-language notation for the constraints. [Borning 1977] is a preliminary description of ThingLab's capabilities. What we have referred to as identification of parts Borning, following Sutherland, calls "merging". This is because in both systems data structures representing identified objects are actually merged internally

to become a single data structure. This is not done in our constraint system because the fact of such an identification must be explicitly recorded for the sake of the dependency information (which is not included in the other two systems).

References

[Borning 1977] Borning, Alan. "ThingLab -- An Object-Oriented System for Building Simulations Using Constraints." Proc. Fifth International Joint Conference on Artificial Intelligence (IJCAI-5), MIT (Cambridge, August 1977), 497-498.

[Brown 1975] Brown, Allen L. Qualitative Knowledge, Causal Reasoning, and the Localization of Failures. Ph.D. thesis. MIT (September 1975). Also MIT AI Lab Technical Report 362 (Cambridge, March 1977).

[Brown 1978] Brown, Richard. Automatic Synthesis of Numerical Computer Programs. Unpublished Ph.D. thesis proposal. MIT (September 1978).

[Church 1941] Church, Alonzo. The Calculi of Lambda Conversion. Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Davis 1976] Davis, Randall. Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases. Stanford U. AI Lab Memo AIM-283 (Stanford, July 1976).

[de Kleer 1976] De Kleer, Johan. Local Methods for Localization of Faults in Electronic Circuits. MIT AI Lab Memo 394 (Cambridge, November 1976).

[de Kleer & Sussman 1978] De Kleer, Johan, and Sussman, Gerald Jay. Propagation of Constraints Applied to Circuit Synthesis. MIT AI Lab Memo 485 (Cambridge, September 1978).

[Doyle 1977] Doyle, Jon. Truth Maintenance Systems for Problem Solving. M.S. thesis. MIT (May 1977). ALso MIT AI Lab Technical Report 419 (Cambridge, January 1978).

[Fikes 1975] Fikes, Richard E. Deductive Retrieval Mechanisms for State Description Models. Stanford Research Institute AI Technical Note 106 (Menlo Park, California, July 1975).

[Floyd 1967] Floyd, Robert W. "Nondeterministic Algorithms." J. ACM (October 1967).

[Goldberg & Kay 1976] Goldberg, Adele, and Kay, Alan (eds.). Smalltalk-72 Instruction Manual. Xerox Palo Alto Research Center, SSL 76-6 (Palo Alto, California, 1976).

[Hewitt 1972] Hewitt, Carl E. Description and Theoretical Analysis (Using Schemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot. Ph.D. thesis. MIT (January 1971). Also MIT AI Lab Technical Report 258 (Cambridge, April 1972).

[London 1978] London, Philip E. Dependency Networks as a Representation for Modelling in General Problem Solvers. Ph.D. thesis. U. Maryland Dept. of Computer Science Technical Report 698 (College Park, Maryland, September 1978).

[MACSYMA] Mathlab Group. MACSYMA Reference Manual, Version Nine. MIT Laboratory for Computer Science (Cambridge, December 1977).

[McAllester 1978] McAllester, David A. A Three Valued Truth Maintenance System. MIT AI Lab Memo 473 (Cambridge, May 1978).

[McDermott 1974] McDermott, Drew V. Assimilation of New Information by a Natural Language-Understanding System. B.S./M.S. thesis. MIT (1973). Also MIT AI Lab Technical Report 291 (Cambridge, February 1974).

[McDermott & Doyle 1978] McDermott, Drew V., and Doyle, Jon. Non-Monotonic Logic I. MIT AI Lab Memo 486 (Cambridge, August 1978).

[Shortliffe 1974] Shortliffe, E.H. MYCIN -- A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection. Stanford U. AI Lab Memo AIM-251 (Stanford, October 1974).

[Shortliffe 1976] Shortliffe, E.H. MYCIN: Computer-Based Medical Consultations. Elsevier (New York, 1976).

[Stallman & Sussman 1977] Stallman, Richard M., and Sussman, Gerald Jay. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis." Artificial Intelligence 9 (1977), 135-196.

[Sussman & McDermott 1972] Sussman, Gerald Jay, and McDermott, Drew V. "From PLANNER to CONNIVER -- A Genetic Approach." Proc. AFIPS 1972 FJCC. AFIPS Press (Montvale, N.J., 1972), 1171-1179.

[Sussman & Stallman 1975] Sussman, Gerald Jay, and Stallman, Richard M. "Heuristic Techniques in Computer-Aided Circuit Analysis." IEEE Transactions on Circuits and Systems vol. CAS-22 (11) (November 1975).

[Sussman 1977] Sussman, Gerald Jay. "SLICES: At the Boundary between Analysis and Synthesis." MIT AI Lab Memo 433 (Cambridge, July 1977). Also in IFIP W.G. 5.2. Working Conference on Artificial Intelligence and Pattern Recognition in Computer-Aided Design.

[Sussman, Winograd, & Charniak 1971] Sussman, Gerald Jay, Winograd, Terry, and Charniak, Eugene. Micro-PLANNER Reference Manual. AI Memo 203A. MIT AI Lab (Cambridge, December 1971).

[Sutherland 1963] Sutherland, Ivan E. SKETCHPAD: A Man-Machine Graphical Communication System. MIT Lincoln Laboratory Technical Report 296 (January 1963).