Automatic Synthesis of Numerical Computer Programs

Richard H Brown

GTE Laboratories, Inc. 40 Sylvan Rd, Waltham, Ma

Abstract

This paper is about rule-based problem solving. It has a motto:

> *The less a rule can do, the better the effects of using it can be predicted.*

This motto suggests the key to obtaining coherent behavior by avoiding exponential behavior (combinatorial explosion) in a problem solving program for the domain of synthesizing numerical computer programs from their non-procedural specifications

Introduction

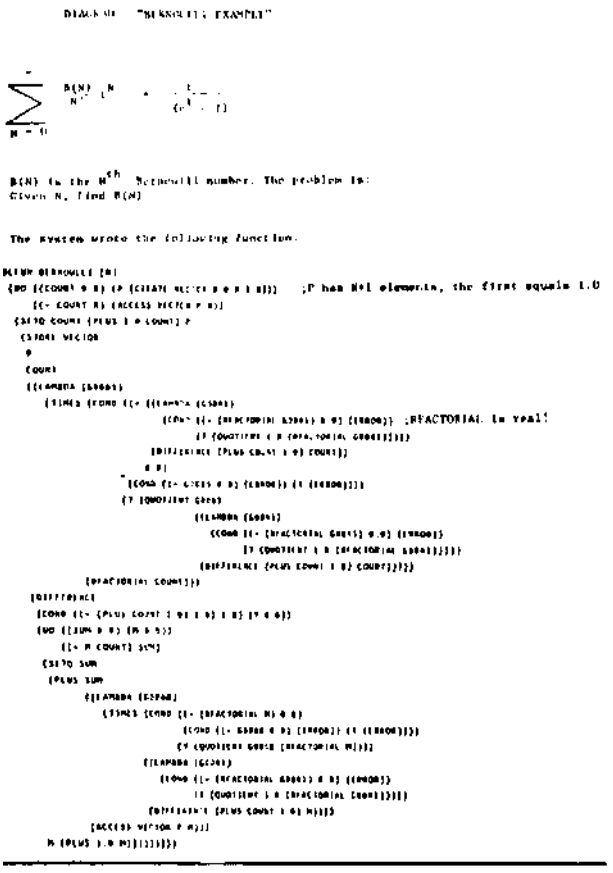
To construct an automatic programming system, several separate but related sub-domains and their corresponding experts are required. These include *program understanding, specification acquisition, program verification,* and *program synthesis.* The last is the domain of expertise of the system described by this paper.

A program synthesis system takes a specification (for example, a predicate calculus description of the relationships between the inputs and the outputs) of a program and produces a program meets those specifications. Diagram "Bernoulli Example" shows an input specification (in mathematical notation) and the LISP code produced by the system.

The fundamental difficulty in constructing problem-solving systems is the so-called "combinatorial explosion" problem. That is to say, given a reasonable measure of how hard the problem at hand *should be,* a "combinatorially explosive" problem solver takes an amount of time proportional to an exponential (or worse!) function of that measure In general, the size of program (and therefore certainly the time needed to write it) is an unbounded function of the size of its specification For purposes of measuring the coherence of a problem solver, a measure of "how hard a problem is" could be the number of rule applications that were *needed* (as opposed to the larger number actually performed) in the solution found.

The result of this research is a collection of techniques (embedded in a synthesis system) that synthesize numerical computer programs. These techniques do *not* depend strongly on the fact that the task is synthesizing programs, so there is hope that they could be adapted to problem-solving in other domains. I conjecture that these techniques in fact run in time proportional to a polynomial function of the number of rules required to solve a synthesis problem A working program was obtained that (on a relatively unloaded PDP-IO) can solve several examples faster than an expert human programmer.

This report describes research done at the MIT AI laboratory. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0505.



Key Principles

There are three basic principles on which this research is based:

I. *Synthesis by analysis* of constraints. Since the use of "networks of constraints" in Statlman and Sussman's EL system [S79], this formalism has become increasingly popular This research uses the idea of propagating code fragments (in addition to the usual arithmetic and algebraic expressions) through a network of constraints in order to write code. Stated another way, the "data base" of constraints connected together to form a network is interpreted many different ways by the synthesis system.

2 *Diagrams provide Insight.* In terms of the actual implementation, synthesis problems are of course presented as text strings. Internally, the problem is represented as a "network of constraints." What might be termed

"topological configurations" in this internal representation seem to be the real key in deciding what deductions to perform next. By extensively using the diagrams of networks of constraints, much insight into the recognition and use of these "topological configurations" was obtained

3. *Combinatorial Blow-up* should be the central concern People can perform the task of writing code given specifications What ever this task actually involves, people do *not* suffer from combinatorial blow-up. This observation should be the starting point for an attempt to get computers to perform program synthesis: if a combinatorial amount of work is required to perform the task, then the task must be reformulated.

In developing a program synthesis system it is crucial to avoid trying to solve problems whose solution time is exponential (or worse) in the length of the solution derivation *(Solution time* and *length* refer to writing the code, not running the code) Notice this is not the usual time-complexity measure.

How can "combinatorial blowup" be avoided? To see, consider a detailed example proving a statement in the propositional calculus is not a tautology (this is known to be NP-complete). The statement "A or B implies A" is not a tautology because it is false if A is "false" and B is "true". A proof is an assignment of "true" and "false" to the variables in the statement such that the entire statement is false. The best (known) algorithm for finding proofs like this is exponential in the number of variables (the length of the "solution")

One way to limit a deductive system (hoping to make it run in polynomial time) is to limit the expressive power of the "rule language." The term "expressive power" refers not to (an absence of) syntactic sugaring, but to the existence of predicate calculus formulae that cannot (in any way) be translated into rules. As an example of such a restriction, consider a modification to the task of proving the non-tautologic status of certain propositional calculus expressions Suppose the form of the expression is limited to implications of conjunctions of variables - expressions without the "OR" connective *and* without the "NOT" operator (these are Horn clauses, [H74]). Such an expression (like "A and B and C implies A and D") can easily be shown not a tautology by finding a letter variable (D in this case) on the right hand side of the implication that is not among the conjuncts to the left of the implication. If this letter variable is assigned "false" and all others assigned "true", the entire expression evaluates to "false " Expressions of this restricted form can be proven non-tautologies in *linear time* (actually O(n*log(n)) because of the time taken to look up variable names)

Example of the system's capabilities

Naturally one must be concerned that a program synthesis system with a restrictive rule language (and specification language) nonetheless still be able to solve "interesting" problems and write "interesting" code The synthesis system has solved examples concerning the general topics:

1. Systems of equations
2. Inverses and zeros of functions
3. Changing recursive control flow to iterative, and *vice versa*
4. Algorithms involving data structures

The "example space" of numerical computer programming problems is very large, the list of example problems above is very small. Nonetheless, this small list forces the system to demonstrate competence in handling many (if not most) of the concepts of numerical computer programming. These concepts include: convergence, power series, successive approximation, symbolic differentiation (and other symbolic operations), transformations of program control structures, and limits.

Data Structures and Bernoulli Numbers

As an illustration of the system's capabilities, style of reasoning, and rule content, let's look at the deductions leading to the Bernoulli number program (see diagram "Bernoulli Example").

Bernoulli numbers (written as a function B(n) for n a positive number) are defined by the equation

$$\text{SUM from } n=0 \text{ to inf. } B(n)*(t^n/n!) = t / (e^t - 1)$$

Suppose the system is given this equation, and asked to write a program for B(n). The definition of B(n) is "buried" inside an (infinite!) loop. It is revealing to try expressing this problem in the spirit of synthesis systems proposed by Burstall and Darlington[Bu77], Barstow[Ba77], and Manna and Waldinger[M79]. *it cannot be done* without a significant extension of the formalism. One might expect that this would make the problem hard to state, and the rules needed to solve this problem very hard to write. In fact, the representation for loops makes no distinction between "inside" and "outside", and the problem and the rules to solve it can be written straightforwardly.

The system solves this problem, unsurprisingly, in a more or less standard way. The left side is (or can be viewed as) a polynomial series (power series) in the variable "t". Using the fact (contained in the set of rules provided to the system) that

$$e^t - 1 = \text{SUM from } m=1 \text{ to inf } t^m/m!$$

the defining equation can be written as the multiplication of two power series. Using a general rule for multiplying power series, the system derives

$$\text{SUM from } L=1 \text{ to inf } [\text{SUM from } p=0 \text{ to } L-1\ B(p)*(1 / (P! * (L-p)!))] * t^L = t$$

But from this it can be deduced that, except for L=1, the equation

$$\text{SUM from } p=0 \text{ to } L-1\ [B(p)*(1 / (P! * (L-p)!))] = 0$$

holds. Looking at the equation above, it is clear that if B(0) up to B(L-2) are known, then B(L-1) can be computed. Specifically,

$$B(L-1) = - \text{SUM from } p=0 \text{ to } L-2\ [B(p)*(1 / (p! * (L-p)!))] * (L-1)!$$

For L=1, it must be that

$$\text{SUM from } p=0 \text{ to } L-1\ [B(p)*(1 / (P! * (L-p)!))] = 1$$

so B(0) = 1.

The analysis above can be used to write a program to compute B(n); complete details can be found elsewhere[B81]. If a recursive program is read off of the equations directly, the resulting program will run exponentially in the size of its argument. On the other hand, an algorithm employing a dynamically built table starting with B(0) up to B(n) has a time-cost of about n^3.

There should be no surprise that the synthesis system is able to write the "exponential" Bernoulli number generator. But just as the system uses "canned" loop schemas, it can use "canned" data-structure schemas to lead directly to data-structure specification and realization. The code in diagram "Bernoulli Example" uses a dynamic vector ("P") in an $O(N^3)$ algorithm.

A Sketch of the Synthesis System

In writing axioms for arithmetic, one often finds instead of, say, addition being treated as a function of two arguments, it is written as a predicate of three arguments. To say, for example, that A=X+Y and B=X·Y, one could write something like

(+C X Y A) and (·C B Y X).

One can Interpret these expressions as declaring that certain relations hold among the variables X, Y, A, and B, or as a rule to compute A or B from X and Y.

The rule language for the program synthesis system is based on a representation combining both the declarative and computational interpretations Specifically, a **C device* would be defined as having three *terminals* (arguments) named SUMMANDI, SUMMAND2. and SUM. Some *constraint rules* would be written telling that some terminals can be computed on the basis of others, and giving details about how that computation is effected:

terminal computed	terminals needed	expression
SUM	SUMMAN01.SUMMANDI	(+ SUMMANDI SUMMAND 2)
SUMMANDI	SUM.SUMMAND2	(- SUM SUMMAND2)
SUMMAND2	SUM.SUMMANDI	(- SUM SUMMAND2)

The expressions can be thought of as LISP code, although the system actually uses a separate constraint rule language and interpreter system so that expressions can be evaluated not only to form code, but to provide time-costs, upper and lower bounds, and symbolic expressions

Stating the Problem

A programming problem is given to the synthesis system as a series of devices connected (at their terminals) to various variables (represented by *nodes*). When devices are connected to nodes, a *network* is formed. In addition to a network, the programming problem also specifies which nodes are to be inputs, and which is the output.

Although these networks are given to the system as a sequence of LISP expressions, several important insights can be obtained by examining a network drawn out in a diagram. The network corresponding to

(*C X Y A) and (*C B Y X)

is shown in diagram "two-linear-equations".

Suppose that nodes A and B are specified as inputs, and node Y is specified as the output. A process called *propagation* (a form of local deduction) would examine the constraint rules of the two *C devices (named Dl and D2) to see if any other nodes can be assigned values In this example no new values can be obtained, so some other, more global, deductions apparently must be made

The diagram for this example contains a configuration called a *circuit* This circuit starts at node X, goes to device Dl (direction is *not* important), then to node Y, then to device D2, and finally back to node X. None of the nodes in the circuit have been given values (which is to say that the system doesn't know how to compute them yet). A key observation is that if a problem can be solved, but is not solved by propagation, then the output node must be in a circuit (or can be determined by propagation from nodes that are in a circuit)

A second key observation is that to solve a problem one wants, somehow, for circuits to get smaller (see "Two Linear Equations Solution") The system uses a library of *transformation rules* (or simply *transforms)* to make this happen.

Transformation rules

It is the language for writing these transformation rules that has been restricted A network can be considered a conjunction of constraints (a *constraint* is a device interpreted as a "predicate" of all its terminals) Constraints must all be satisfiable (be "true") in the sense that there must exist some assignment of values to nodes so that all constraints (considered to be predicates) evaluate to "true." A transformation rule is a statement to the effect that one network can be transformed into another (called the "instantiation network"). Of course the entire transformation must be "true" when considered as a mathematical statement But to prevent expressing a disjunction, a stronger restriction is imposed the instantiation network, considered in isolation, must have the property that all its constraints (expressed as devices) are satisfiable

For example, one transformation the system has in its librarv savs that nr-2r.

((*C R R S)) => ((*C 2 R S))

This transform could be drawn as shown in diagram "doubling."

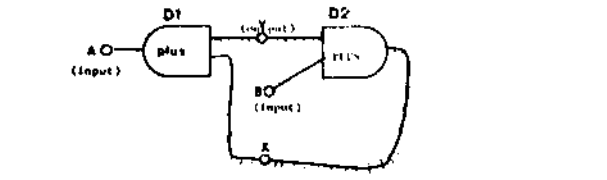


DIAGRAM "TWO LINEAR EQUATIONS"

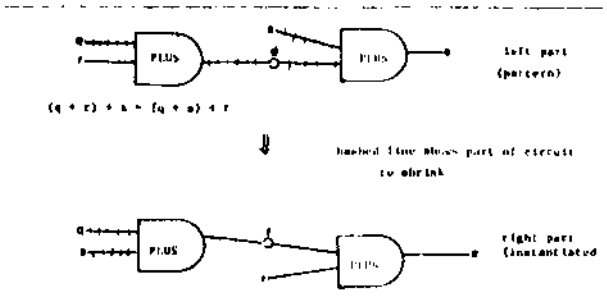


DIAGRAM "FINDY ASSOCIATIVITY"

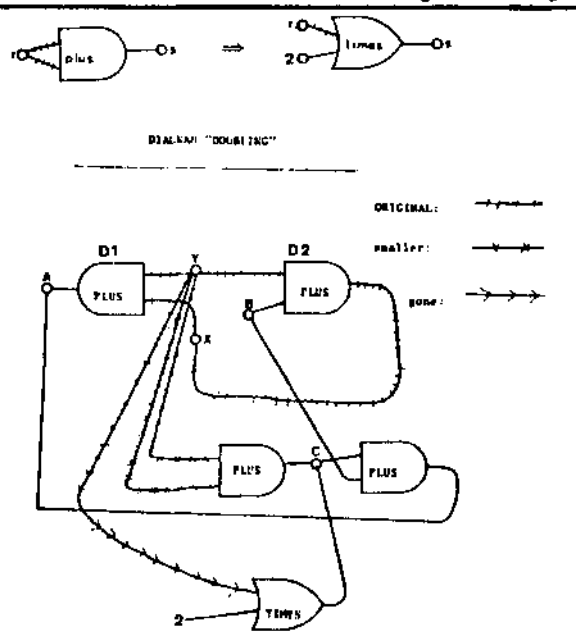


DIAGRAM "TWO LINEAR EQUATIONS SOLUTION"

An important observation to make concerning this diagram is that if S is known and node R is not, then the left side of the diagram contains a circuit (with only one node R) but the right side does not The "doubling" transformation apparently "removes" a circuit when S is known and R is not.

Another transform, expressing the mathematical fact that

(Q,+R)*S=(Q,*S)*R

is shown in diagram "funny associativity." There is nothing special about this transformation, other than it happens to help solve the current problem Written as constraints, this transform is

((+C R QD) (+C D S E» => <(+C S Q F) (+C R F E».

The left part of this diagram (the *pattern)* matches the problem in the diagram "two linear equations" with the correspondence

Q-> Y, R «> B, S -> Y, E -> A

The result of applying the "funny associativity" transform is to *add* a copy of the instantiation network to the problem network (of course. this may trigger other rules -- combinatorial explosion is a danger the system controls, see below and [B8I]). A new node must be created for the node F because it does not appear in the left-hand pattern part of the transform. The solution is found in the data base after applying "funny associativity" followed by "doubling" (diagram "two linear equations solution")

An important observation to make concerning the "funny associativity" diagram is that, in the left (pattern) part, there are exactly two circumstances in which using this transform will result in shrinking a circuit. These are first if nodes "Q" and "S" are known, and nodes "R" and "E" are unknown, and second if nodes "R" and "E" are known and nodes "Q," and "S" are unknown. The second situation is extant in the initial problem.

Limiting expressive power and combinatorial blowup

The mathematical facts contained in the "funny associativity" and "doubling" are of the form that the left side (a conjunction) *implies* the right side (another conjunction). All transforms have this simple form Disjunctions (use of OR) and negations (use of NOT) are not allowed The expressive power of the transformation rule language has been limited Furthermore, it should be remembered that propositional statements of this restricted form (implications of conjunctions) are precisely those for which the non-tautology proof was easy (i.e. Horn clauses)

To see that the expressive power *has* been limited, notice that even though implication can be rewritten as shown, it cannot be used to encode disjunction

A => B is equivalent to ~A OR B

Trying to encode a disjunction using negation doesn't work because negations are not allowed;

(~A) -> B (equivalent to A OR B).

Direct use of negation is not allowed. But maybe indirectly one could encode negation

C -> "FALSE" (equivalent to ~C OR "FALSE", equivalent to ~C)

But consider what such a transformation rule would have to say: From C one can deduce that some constraint holds among some nodes when in fact it can never be satisfied! This violates the imposed condition that all constraints in a network are satisfiable So a disjunction cannot be constructed in this (or any other) way.

By using this restricted form of transformation rule two separate kinds of combinatorial explosion (exponential behavior) have been avoided. The first concerns the general problem of proof by contradiction and data base splits. The second concerns legitimate but useless deductions.

If a problem-solver is faced with a fact like "x implies y or z" a common reaction is (if x is true in the current data base) to assume y and see if a contradiction results. If one does, then (one way or another) the deductive system "backs up" and decides that "not y" and "z" are both the case. Since in effect the problem solver is thus searching a tree, there is a potential for exponential behavior - an exponential number of data bases must be searched (in general) as a function of the depth of the tree.

In sharp contrast, the transformation rule language has been restricted so that it is impossible to state a rule that might give rise to a tree of data bases Reasoning by contradiction cannot be used by the deductive system because the rules are not expressive enough to say what should be done if a contradiction is found.

One could worry that the number of rules might increase exponentially. Even worse, that knowledge required might be unexpressible But this turns out not to happen. For example, looking at Barstow's [Ba77] set of rules, almost all are implications of conjunctions, and the exceptions can be restated in this form without increasing the number of rules

Another kind of exponential behavior arises in the problem of finding the value of a variable Vf given the values of VI through Vn when the following is true:

(Vf ♦(VI *(V2* … *(Vf ♦ …))))-0.

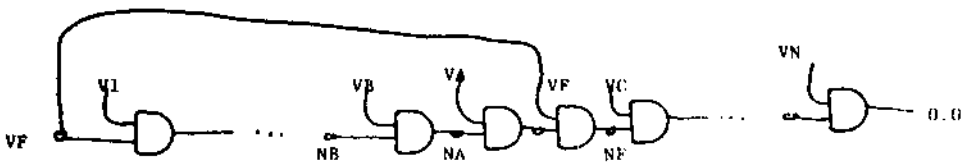
Suppose that the system can interchange two variables without changing the nesting. By applying the interchange operation, all sequences of Vf and VI through Vn can be generated, and there are 0.5*(n*2)' such sequences If the system can solve the problem after getting the two Vf variables next to each other, then in fact no more than n interchanges actually need to be performed. Since the factorial function grows faster than an exponential, there is a kind of exponential behavior to avoid See diagram "Interchange Problem" for a demonstration of how knowledge of *circuits* solves this combinatorial problem.

Because the form of transformation rule is so simple, one can determine *in advance* the effect of applying a transformation with respect to shrinking or removing circuits. The doubling transformation, for example, removes the circuit containing the node R. Before an applicable transformation is actually used, the system performs a few simple tests to see if the transformation will shrink or remove a circuit of current interest (these tests involve examining nodes to see if they are known or unknown). If it won't, then the transform's use is postponed. In this way, the form of exponential behavior described above is avoided. *// disjunctions were allowed on the right of implications, this style of analysis would be impossible.*

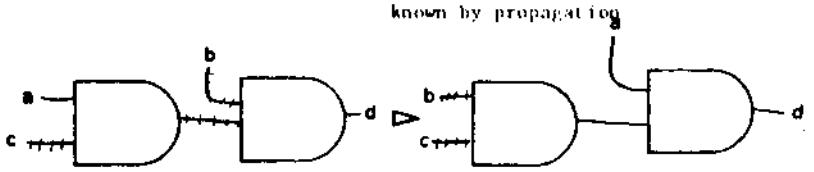
The network-matching problem is known to be NP-complete, and indeed, finding a pattern of N nodes in a graph of M nodes takes time on the order of M . Therefore, matching transformations against the "problem network" is exponential in the sue of the transformation pattern; but this sue is *bounded a priori* because the size N of the longest rule the user wrote is independent of the particular problem (size M) being solved Conversely, the conjectured bound on the synthesis system's run time is a polynomial (in the number of transformations needed) of order "the longest transformation pattern"

Specifications involving iteration

One would like to be able to reason directly about looping control structures, but the undecidability of almost any question concerning looping procedures makes the situation hopeless. Since one cannot in general reason about looping control structures, the approach taken has been to "bury" looping in device rules.



VA,VB,VI,...,VN ARE KNOWN, SOLVE FOR VF
nodes NA, NB, ... are unknown, node NF and nodes to right are known by propagation

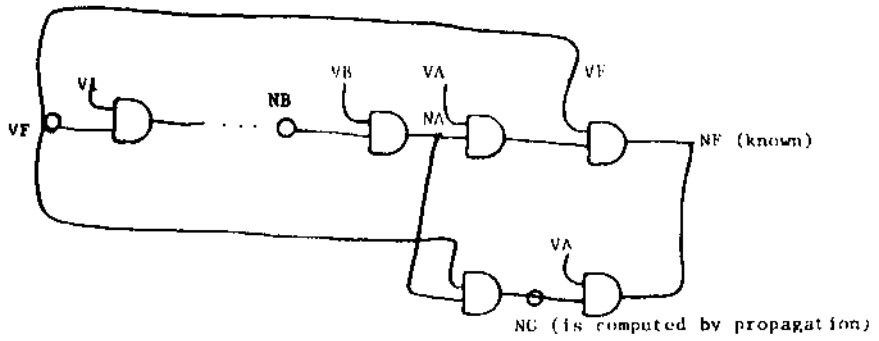


Above is the "interchange rule". A priori, this shrinks a circuit c--b, so system only (but see [B81]) applies this rule if nodes c,b are unknown and nodes a,d are known.

The only site meeting application requirements has matching:

c - NA, a - VA, b - VF, d - NF

AFTER APPLICATION AND PROPAGATION, CIRCUIT WILL LOOK LIKE THIS:



NOW THE ONLY SITE FOR "INTERCHANGE RULE IS:

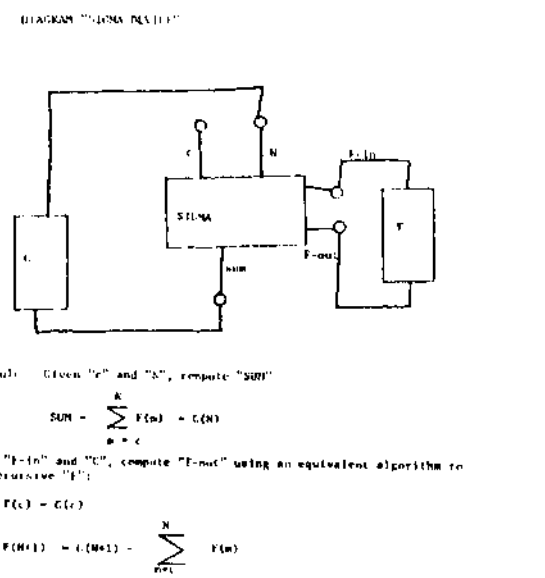
c - NB, a - VB, b - VF, d - NG.

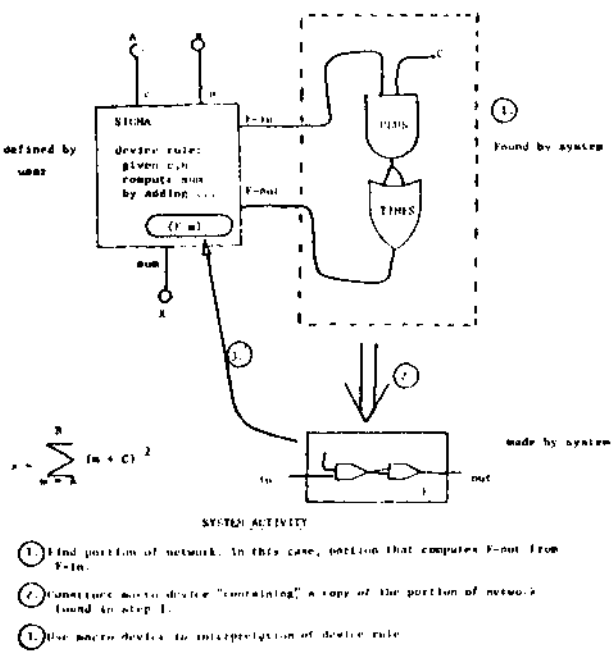
DIAGRAM "INTERCHANGE PROBLEM"

For example, there is a SIGMA device (see diagram "Sigma Device") that contains prepackaged looping control structures for computing SUM from numbers C, N, and a function F:

SUM - sum M from C to N of F(M)

The device rule for the SIGMA device contains a "blank" to be filled in with "code" for the function called F to be found by the synthesis system When the system finds how to compute F, that knowledge is packaged into a structure called a *macro-device*





The key to being able to "bury" all iteration and recursion into prepackaged control structures is being able to tell the synthesis system how to find macro-devices This is always done by telling the system that the macro-device has certain nodes as inputs, and others as outputs. In the case of SIGMA, terminal F-in is macro-device F's input, and terminal F-out is F's output.

Diagram "Introduction to Macro Devices" summarizes how the system responds to a device rule in SIGMA asking for a macro-device F Essentially, three *steps are* involved, *first find* the portion of the network that can perform the required computation, then package this portion of the network into a macro-device, and finally use the macro-device in device rule interpretation.

By using the macro-device mechanisms, devices for various kinds of iteration and recursion, as well as devices like SIGMA, devices for doing bisection searches, and devices for power series manipulation have been written. Transformation rules involving these devices can ignore the fact that the devices' definitions involve looping control structures.

References

[Ba77] Barstow, D. *Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules,* Stanford AI Memo 308,1977.

[B8I] Brown, R. *Coherent Behavior from Incoherent Knowledge Sources in the Automatic Synthesis of Numerical Computer Programs,* MIT AI-TR-610, 1981

[Bu77] Burstall, R and Darlington, J "A Transformation System for Developing Recursive Programs," *JACM* Vol 24, No. I. Jan 1977.

[H74] Henschen, L. and Wos, L. "Unit Refutations and Horn Sets," *JACM* Vol. 21. No. 4, Oct 1974.

fM79] Manna, Z. and Waldinger, R. "Synthesis: Dreams -> Programs," *IEEE Transactions on Software Engineering,* Vol. SE5. No. 4, July 1979.

(S79] Stallman, R. and Sussman, G. "Problem Solving About Electrical Circuits" in *Artificial Intelligence: An MIT Perspective* Winston and Brown (eds), MIT Press, 1979.