MTR-3661

A VALIDATION TECHNIQUE
FOR COMPUTER SECURITY
BASED ON THE THEORY OF CONSTRAINTS

BY FREDERICK C. FURTEK

DECEMBER 1978

Prepared for

DEPUTY FOR TECHNICAL OPERATIONS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

Project No. 5720
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-78-C-0001

79 02 26 135

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

DANIEL R. BAKER, Captain, USAF
Technology Applications Division

CHARLES J. GREWE, Jr., Lt Colonel, USAF
Chief, Technology Applications Division

FOR THE COMMANDER

NORMAND MICHAUD, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Technical Operations

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
ESD-TR-78-182		

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
A VALIDATION TECHNIQUE FOR COMPUTER SECURITY BASED ON THE THEORY OF CONSTRAINTS	
	6. PERFORMING ORG. REPORT NUMBER
	MTR-3661

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Frederick C. Furtek	F19628-78-C-0001

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
The MITRE Corporation P.O. Box 208 Bedford, MA 01730	Project No. 5720

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Deputy for Technical Operations Electronic Systems Division, AFSC Hanscom AFB, MA 01731	DECEMBER 1978
	13. NUMBER OF PAGES
	65

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
67p.	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

AUTOMATA THEORY	MULTIPLE-VALUED LOGIC
COMPUTER SECURITY	PRIME CONSTRAINTS
CONSENSUS	PRIME IMPLICANTS
DEDUCTION	RESOLUTION
	SWITCHING THEORY

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A validation technique is described that is both necessary and sufficient to test for security compromise. A LISP program documented in this report automates a substantial portion of the validation process. The theory of constraints, which incorporates elements of switching theory and automata theory, provides the mathematical foundation. In addition to detecting compromise, the approach may → next page

(cont.)

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73

20. ABSTRACT (concluded)

cont.

be used to prove a wide range of properties about system behavior. The technique is suited to both hardware and software, and is applicable at various levels of specification.

ACKNOWLEDGMENTS

1

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

SECTION 1

INTRODUCTION

LIMITATIONS OF EXISTING VALIDATION TECHNIQUES

Considerable effort has been directed over the past several years at developing formal techniques for security validation. Although substantial gains have been made, there remain a number of problems:

a. Most of the work so far has concentrated on validating software, and very few of the results are applicable to hardware.

b. The existing security tests are stronger than is necessary. It is therefore possible for a secure system to be rejected as insecure.

c. Present techniques have difficulty in handling the subtle compromises due to 'modulation' and 'timing' channels.

d. None of the techniques so far developed has been completely automated. Typically, a large part of the validation effort requires human intervention.

e. Most existing validation approaches are applicable at only a single level of detail - for instance, the operating-system level or the assembly-language level. No approach is applicable across all levels of detail.

f. The issues of 'data integrity' and 'denial of service' have not been addressed.

It was with these limitations in mind that MITRE began looking for a more suitable approach to formal security validation.

A NEW APPROACH

The initial effort at MITRE in overcoming these limitations produced some promising results. It was discovered that the compromise of information is intimately related to the concept of a 'prime constraint'. With prime constraints it was now possible to state a necessary and sufficient condition for the existence of compromise. A compromise would therefore be detected if, and only

6

if, one, in fact, existed. This emerging technique also appeared to be general enough to handle both hardware and software, and flexible enough to be applied at any level of detail. In short, it looked as if prime constraints would provide the key to overcoming many of the limitations of existing techniques.

There was, however, one major obstacle: There was no effective procedure known to test for the existence of a prime constraint of a specified type. Without this ability it was not possible to test for compromise. Most of the difficulty stemmed from the fact that a finite system could have an infinite number of prime constraints, and so it was not possible to exhaustively check each prime constraint for potential compromises.

Clearly, what was needed was a finite representation for a possibly-infinite set of prime constraints and an algorithm for generating that representation. It is just such a representation and algorithm that we describe in this report. The 'prime-constraint graph' of a system is a finite graph in which every prime constraint is represented by a path. With this graph it is a straightforward matter to determine what compromises, if any, are possible.

OUTLINE OF THE REPORT

Section 2 reviews the five basic concepts underlying our model: 'conditions', 'variables', 'states', 'transitions', and 'simulations'. The notions of 'constraint' and 'prime constraint' are also presented, along with the 'deduction theorem' which establishes the relationship between prime constraints and the compromise of information.

The 'prime-constraint graph' of a system is discussed in Section 3, and the user interface of an automated tool for generating the graph is described.

In Section 4 the algorithm for generating the prime-constraint graph is described, and some of the mathematical results underlying the algorithm are presented.

A summary of the results achieved and suggestions for future work are included in Section 5.

SECTION 2

BACKGROUND

A PROBLEM

'Information-flow analysis' [1,2] is currently the primary
technique for establishing the security of computer programs. The
ideas behind this approach are straightforward: The elementary
information flows for each program statement are established first.
In general, if a variable x is a function of the variables
y1,...,yn, then there is said to be a flow from each of the y's to
x. The next step is to take the 'transitive closure' of the flow
relation. This means that if there is a flow from x to y and a flow
from y to z, then there is assumed to be a flow from x to z. Once
all of the information flows have been established, it is then
simply a matter of determining whether any of the flows produces a
security violation. In the case of multi-level security, this
involves checking to see that there is no flow from a variable with
a high security level to a variable with a low security level.

The claim that is made for information-flow analysis is that if
a program has a flow violation, then that violation will be exposed.
The converse, however, is not true. For a secure program,
information-flow analysis may detect a flow violation that, in fact,
is not a compromise. Consider the following simple program (in
which ● denotes modulo-2 addition):

```
Boolean: a,b,c,d
b:=a
c:=a
d:=b●c
```

Information-flow analysis establishes the following flows:

```
a----->b
a----->c
b----->d
c----->d
a----->d
```

Now if Variable 'a' is at a higher security level than Variable 'd',
then a flow violation is detected because there is a flow from 'a'
to 'd'. It turns out, however, that when the assignment d:=b●c is
made, d is __always__ assigned the value '0' - regardless of the value
of a. ('b' and 'c' will always be equal when the assignment is

8

made.) Under these circumstances, we would like to say that there
is no flow from a to d, but conventional information-flow analysis
is unable to accept that conclusion.

The problem is that information flow is not transitive. Flows
from x to y and from y to z do not necessarily imply a flow from x
to z. What is needed is a more sophisticated notion of information
flow, one that does not assume transitivity.

The concept of a prime constraint provides us with just the
tool we need. Although prime constraints do not deal explicitly
with the notion of information flow, they give us something more
valuable. They tell us under what circumstances 'deductions' can be
made, and, as we have shown [3], deductions are the key to
determining whether or not a system is secure. (In the example
above, knowing the value of d after the last assignment is made does
not permit us to deduce anything about a.)

OUR MODEL

The model upon which our approach is based has already been
described in earlier papers [3,4,5]. For reference, we review here
the essential definitions.

A system is asssumed to have associated with it a finite set of
variables, with each variable ranging over a finite set of values.
A condition is an assignment of a value to a variable. A state is
any set of conditions containing exactly one condition for each
variable.

We assume that each system has a finite set of state
transitions which define the allowable behavior for the system. A
simulation is any state sequence for which every ordered pair of
consecutive states is a state transition. We place no restrictions
on the initial state of a simulation.

A term is any set of conditions such that no variable has all
its conditions in that set. For a term t, states(t) denotes the set
of states s such that for each variable with conditions in t, s
contains one of those conditions. The empty set is a term of
special interest since states({}) is the set of all states.

9

A <u>clause</u> is a finite (possibly empty) sequence of terms. For a clause x, <u>seq(x)</u> denotes the set of state sequences w such that:[1]

1. w is the same length as x, and

2. w(i) is a member of states(x(i)).

CONSTRAINTS

The concepts of a 'constraint' and of a 'prime constraint' have already appeared in a number of papers [3,4,5,6,7]. Although the two definitions are simple, they are somewhat counter-intuitive since they deal with <u>excluded</u> state sequences (nonsimulations).

First of all, a <u>constraint</u> is any clause x such that seq(x) contains no simulations. A constraint, in effect, represents an incompatibility among successive states in a simulation. We refer to a constraint of length n as an <u>n-place constraint</u>.

In order to define a prime constraint, we must first introduce the notion of one clause 'covering' another. Clause x is said to <u>cover</u> Clause y if and only if there exists a (consecutive) subsequence z of y such that x and z are the same length and states(z(i)) $\subseteq$ states(x(i)). From this definition it follows that if Constraint x covers Constraint y, then every state sequence excluded by y is also excluded by x. (Note that any extension of a nonsimulation is also a nonsimulation.)

A <u>prime constraint</u> is simply any constraint that is not (properly) covered by another constraint. A prime constraint thus may be viewed as a 'maximally reduced' constraint. The set of two-place prime constraints has a special significance since it is equivalent to the set of state transitions. From the standpoint of security, prime constraints are of interest because they determine precisely what 'deductions' may be made in a system.

DEDUCTIONS

Let us suppose that there is a person with access to a subset of variables A. This means that the person knows for every

----

[1] For a sequence z and an integer i, z(i) denotes the i'th component of z.

10

simulation of the system what the conditions are for each of the variables in A. It is not important whether this access is through observation (reading), modification (writing), or some combination of the two.

Now consider a second subset of variables B which is disjoint from A. We shall assume that all of the knowledge that the person has about the behavior of the variables in B is gained from his knowledge about the behavior of the variables in A and from his knowledge about the structure of the system. The person has no direct access to any of the variables in B.

We now ask the following question: Under what circumstances can access to the variables in A be used to deduce something about the behavior of the variables in B? Before answering that question, we first need to clarify what it means to 'deduce something'. We shall say that access to a set of variables A can be used to deduce something about a disjoint set of variables B if and only if,

> There exist two simulations u and v, both the same length, for which there is no third simulation w, the same length as u and v, such that $w_A = u_A$ and $w_B = v_B$.[2]

This requirement is a formal way of saying that the pattern u_A and the pattern v_B are mutually exclusive. That is, the presence of u_A in a simulation of length n excludes the presence of v_B, and vice versa. Thus, a person observing the pattern u_A in any simulation of length n is able to deduce that the pattern v_B could not also have occurred. But the person knows that v_B could have occurred under different circumstances - for instance, if he had observed v_A. Observing u_A in a simulation x of length n therefore provides additional information about what patterns are possible for x_B. In particular, the observer now knows that $x_B \neq v_B$. (Notice that it is not necessary for the observer to know precisely what x_B is. It is sufficient that he is able to narrow the set of possibilities.)

Having formalized the concept of deduction, we can now answer the question posed above.

----

[2] If x is a clause and Q a set of variables, then x_Q denotes the new clause that is obtained from x by removing all conditions not belonging to a variable in Q.

11

Theorem: Access to a set of variables A can be used to deduce something about a disjoint set of Variables B (and vice versa)

if and only if

there exists a prime constraint that contains at least one condition belonging to a variable in A, at least one condition belonging to a variable in B, but no conditions belonging to any other variables.[3]

So we see that the concept of deduction is intimately tied to the concept of a prime constraint. The ability to determine the prime constraints of a system gives us the ability to determine the deductions of a system.

The problem now is to find a way of generating the prime constraints of a system.

----

[3] A proof of this theorem is given in [3].

12

SECTION 3

THE PRIME-CONSTRAINT GRAPH

REPRESENTING PRIME CONSTRAINTS

Because the number of prime constraints associated with a system may be infinite, it is not always possible to simply list them. But as we show in this section and the next, it is possible to generate a (finite) graph, called the prime-constraint graph, that represents completely the set of prime constraints for a system.

To help illustrate the idea of a prime-constraint graph, we consider a four-stage shift register as an example. There are four binary variables: a, b, c, and d. Now if x is one of these variables and v is one of the values '0' or '1', then xv will be used to denote the condition representing the assignment of v to x. Thus, the condition b1 represents the assignment of a '1' to Variable b. The operation of the four-stage shift register is determined by its set of two-place prime constraints:[4]

<{a0} {b1}>   <{b0} {c1}>   <{c0} {d1}>

<{a1} {b0}>   <{b1} {c0}>   <{c1} {d0}>

From these constraints the prime-constraint graph depicted in Figure 1 is generated.

In a prime-constraint graph, each arc is labelled with a term, and thus each path through the graph is associated with a clause. The interpretation of the graph is straightforward: A clause is a prime constraint if and only if it is associated with a maximal path in the prime constraint graph. The requirement that the path be maximal simply means that the path must start at a node with no input arcs and terminate at an node with no output arcs. For the graph shown, there are only a finite number of maximal paths. The prime constraints associated with these paths are:

----

[4] As a notational convenience, we omit commas in our representations of sets and sequences.

```
    +          +          +
    |          |          |
  {a0}|      {b0}|      {c0}|
    |          |          |
    |     {}   |     {}   |
    v----------+--------->+--------->+
    +--------->+--------->+
    |          |          |
  {b1}|      {c1}|      {d1}|
    |          |          |
    |          |          |
    |          |          |
  {a1}|      {b1}|      {c1}|
    |          |          |
    |     {}   |     {}   |
    +--------->+--------->+
    +--------->+--------->+
    |          |          |
  {b0}|      {c0}|      {d0}|
    |          |          |
    v          v          v
    +          +          +
```
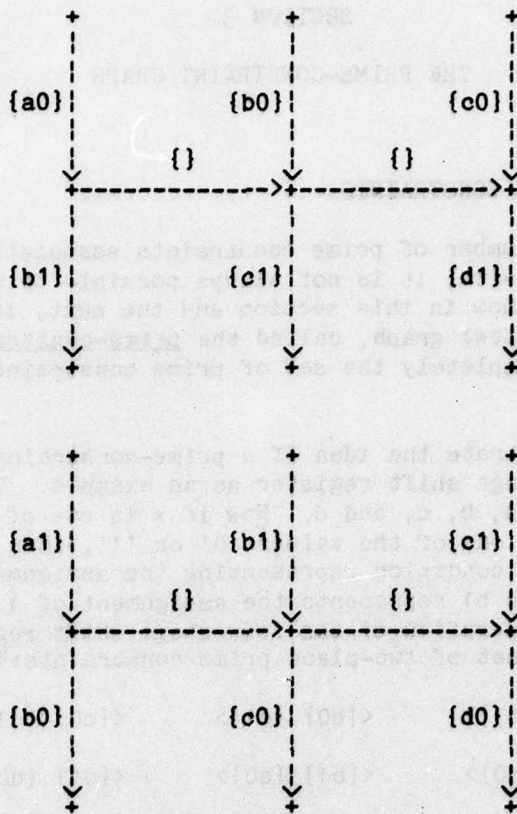
Figure 1.   Prime-Constraint Graph for Four-Stage Shift Register

<{a0} {b1}>   <{b0} {c1}>   <{c0} {d1}>

<{a1} {b0}>   <{b1} {c0}>   <{c1} {d0}>

<{a0} {} {c1}>   <{b0} {} {d1}>

<{a1} {} {c0}>   <{b1} {} {d0}>

<{a0} {} {} {d1}>

<{a1} {} {} {d0}>

A little thought should convince the reader that these prime
constraints do indeed reflect the behavior of a four-stage shift
register. For example, the prime constraint <{b1} {} {d0}> says

that if a state in a simulation contains a b1, then the state two
frames later (if there is one) cannot contain a d0, and therefore
must contain a d1.

In the next subsection we look at what happens when a system
has an infinite number of prime constraints.

LOOPS

It might be supposed that a system with an infinite number of
prime constraints would have to be fairly complicated. This is not
the case. Consider the system with a single binary variable a =
{{a0} {a1}}[5] and a single two-place prime constraint <{a0} {a1}>.

The prime-constraint graph is shown in Figure 2.
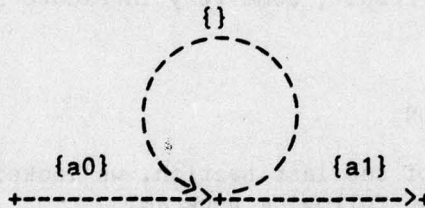
Figure 2. Prime-Constraint Graph with a Loop

The notable feature of this graph is the loop labelled with {}.
It is this loop that permits an infinite number of prime constraints
to be represented. A path from the (unique) starting node to the
(unique) terminating node may traverse the loop any number of times
(including zero), with each such traversal adding another {}. The
set of prime constraints thus forms the following infinite sequence:

----

[5] We have taken the liberty of identifying the set of conditions
belonging to a variable with the variable itself.

15

<{a0} {a1}>

<{a0} {} {a1}>

<{a0} {} {} {a1}>

.
.
.

The interpretation for this sequence of prime constraints is simple:
If the value of Variable a is '0', then it will always be '0'. And
conversely, if the value of Variable a is '1', then it must always
have been '1'.

This example is a trivial one. In general, a prime-constraint
graph is not restricted to a single loop, nor is a loop restricted
to a single arc. As a result, some very intricate structures are
possible.

CHECKING FOR A DEDUCTION

At the beginning of the last section, we looked at the possible
information flows in the following program:

                    Boolean: a,b,c,d
                    b:=a
                    c:=a
                    d:=b●c

We convinced ourselves that there was no flow from Variable a to
Variable d even though a (conventional) information-flow analysis
concluded that there was such a flow and an apparent security
violation. An analysis of this program based on prime constraints
eliminates this discrepancy.

Let us first remove the inessential timing details of the
program by converting it into the closely-related 'flow diagram'
shown in Figure 3. It is then a simple matter to derive a
specification of the program in terms of two-place constraints:

16
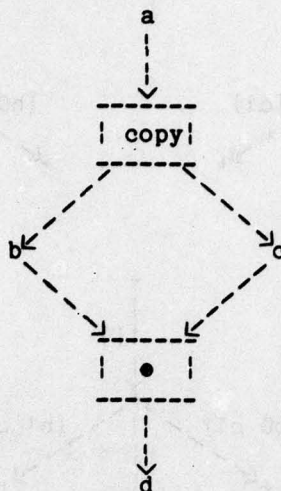
Figure 3. Flow Diagram

$$\langle\{a0\}\ \{b1\}\rangle \qquad \langle\{a0\}\ \{c1\}\rangle$$

$$\langle\{a1\}\ \{b0\}\rangle \qquad \langle\{a1\}\ \{c0\}\rangle$$

$$\langle\{b0\ c0\}\ \{d1\}\rangle$$

$$\langle\{b1\ c1\}\ \{d1\}\rangle$$

$$\langle\{b0\ c1\}\ \{d0\}\rangle$$

$$\langle\{b1\ c0\}\ \{d0\}\rangle$$

The associated prime-constraint graph is shown in Figure 4.

Let us consider some of the deductions possible. From the prime constraint $\langle\{b0\ c1\}\ \{d0\}\rangle$ we see, among other things, that access to Variables b and d permits a deduction about Variable c. Specifically, if the partial simulation in Figure 5(a) is observed, then one can deduce that the partial simulation in Figure 5(b) did not also occur. (By the same token, if the pattern in Figure 5(b) is observed, then one can conclude that the pattern in Figure 5(a) did not occur.)

17

{a0}   {a1}

{b1}   {c1}   {b0}   {c0}

{ }

{b0 c1}   {b1 c0}

{ }

{b0 c0}   {b1 c1}

{d1}

{b0 c1}   {b1 c0}

{d0}
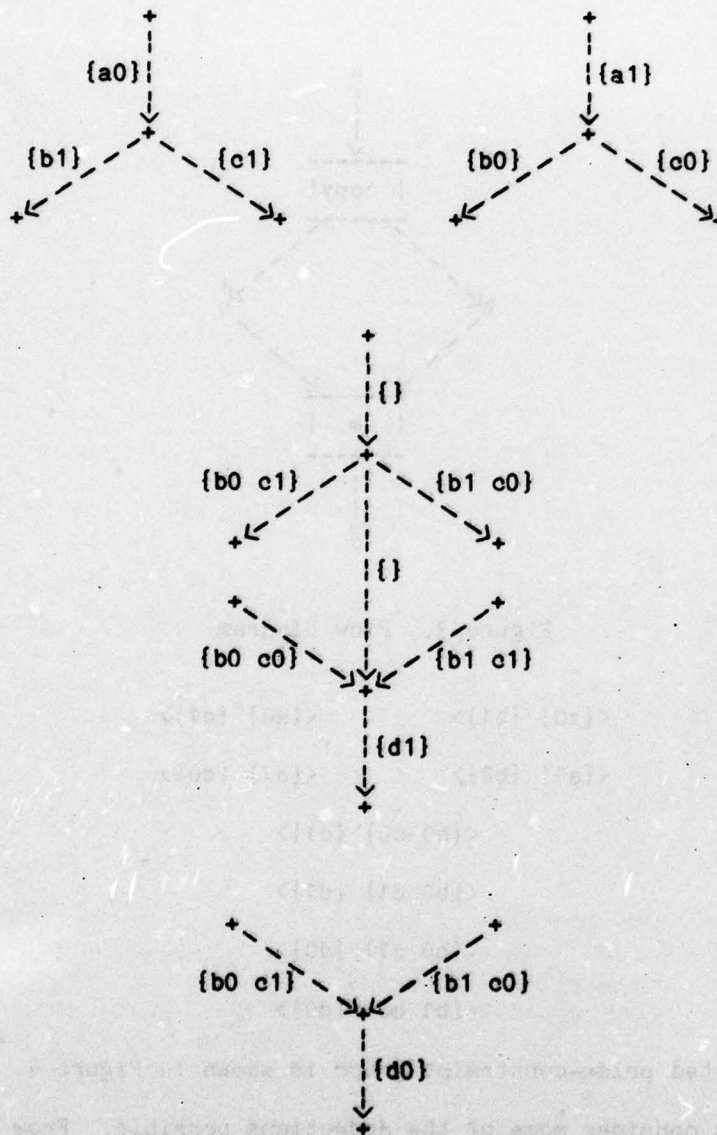
Figure 4. Prime-Constraint Graph for Program Example

Returning to the question of information flow from Variable a
to Variable d, we note that there is no prime constraint containing
conditions for both Variables a and d and for no other variables.
From the deduction theorem in Section 2, we therefore conclude that

18

```
              a    b    c    d              a    b    c    d
         -------------------------     -------------------------
Frame 1: |    | b0 |    |    |         |    |    | c1 |    |
         |----|----|----|----|         |----|----|----|----|
Frame 2: |    |    |    | d0 |         |    |    |    |    |
         -------------------------     -------------------------
                   (a)                           (b)
```
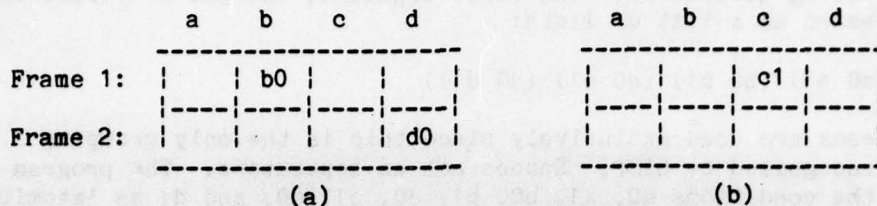
Figure 5. Mutually-Exclusive Patterns

access to Variable d does not permit a deduction about Variable a
(and vice versa). So we see that an analysis of the above program
based on prime constraints is consistent with the conclusion reached
earlier that Variable a is not compromised by access to Variable d.

We might note that access to Variable d does tell us something
interesting. From the prime constraint <{} {} {d1}>, it follows
that if a d1 appears in a simulation, then there could not have
been two or more states preceding the state with the d1. In other
words, a d1 may appear in only the first or second state of a
simulation. The fact that a d1 may appear at all reflects the
possibility that a simulation may be initialized with a d1, or may
be initialized with a b0 and c1 or a b1 and c0.

USING THE TOOL

pcgraph is the name of a LISP program[6] running on the RADC
(Rome Air Development Center) Multics time-sharing system. It
accepts as input a system specification in the form of two
arguments: (1) a set of variables and (2) a set of two-place
constraints. It produces as output the prime-constraint graph of
the system. Note that it is not necessary for the two-place
constraints to be prime. pcgraph will automatically generate the
two-place prime constraints in the course of constructing the
prime-constraint graph.

----

[6] Appendix A contains a description of each of the modules
comprising pcgraph, while Appendix B contains a listing of the
program.

19

To illustrate how pcgraph is used, we consider the example in the preceding subsection. The first argument, the set of variables, is expressed as a list of lists:

((a0 a1) (b0 b1) (c0 c1) (d0 d1))

Parentheses are used exclusively since this is the only grouping symbol recognized by LISP. Spaces act as separators. The program treats the conditions a0, a1, b0, b1, c0, c1, d0, and d1 as 'atomic symbols' and does not attempt to analyze the individual characters making up such a symbol. Thus, the fact that a0 and a1 both contain an 'a' and the fact that b1 and c1 both contain a '1' are totally ignored. What is important is the grouping of conditions. Because a0 and a1 are grouped together in a sublist, they are interpreted as alternative conditions of a single (binary) variable, and similarly for b0 and b1, c0 and c1, and d0 and d1. The only restriction on the choice of symbols for conditions is that no symbol may appear in more than one sublist. This requirement reflects the fact that a condition is associated with a unique variable.

The second argument, the set of two-place constraints, is expressed as:

(((a0) (b1)) ((a1) (b0)) ((a0) (c1)) ((a1) (c0))
 ((b0 c0) (d1)) ((b1 c1) (d1)) ((b0 c1) (d0)) ((b1 c0) (d0)))

Each two-place constraint is of the form: ((conditions) (conditions)). The first set of conditions represents the first term of the constraint, and the second set of conditions the second term. The ordering of the two terms for a constraint is clearly important, but the ordering of conditions within a term and the ordering of constraints is immaterial.

Before invoking pcgraph, it is useful to first create two atomic symbols, one whose value is the set of variables and the other whose value is the set of two-place constraints. (This is most easily done using the 'setq' function.) If we let V be the first atomic symbol and C the second, then pcgraph is invoked by the following command in the LISP environment:

(pcgraph V C)

When pcgraph terminates, it will type out a list of symbols representing all the nodes of the prime-constraint graph except for the unique starting node BEND (for Back END) and the unique terminating node FEND (for Front END). (The prime-constraint graphs given earlier were drawn with multiple starting nodes and multiple terminating nodes merely for convenience.) For our example there

20

are five nodes, excluding BEND and FEND, and so pcgraph upon
terminating will type a list something like:

    (n0123 n0120 n0060 n0043 n0040)

These character strings are internally generated and have no
significance other than to provide unique names for the nodes.

    In order to obtain the arcs of the prime-constraint graph, the
user types:

    (expand ARCS)

The program will then return with a list of arcs. For the example
being considered this list might look like:

    ((a0129 nil back n0060 fore n0120)
    (a0128 (b1 c1) back BEND fore n0120)
    (a0127 (b1 c0) back BEND fore n0123)
    (a0126 (c1 b0) n0060 fore FEND)
    (a0122 (d1) back n0120 fore FEND)
    (a0121 (b0 c0) back BEND fore n0120)
    (a0110 (b0) back n0043 fore FEND)
    (a0108 (c0) back n0043 fore FEND)
    (a0107 (a1) back BEND fore n0043)
    (a0064 nil back BEND fore n0060)
    (a0105 (a0) back BEND fore n0040)
    (a0106 (c1) back n0040 fore FEND)
    (a0109 (b1) back n0040 fore FEND)
    (a0117 (c0 b1) back n0060 fore FEND)
    (a0124 (b0 c1) back BEND fore n0123)
    (a0125 (d0) back n0123 fore FEND))

Each line represents an arc. The first character string in a line
is the name of the arc. As with nodes, this name is internally
generated and has no significance. The second character string
represents the term associated with the arc. And the remaining
strings indicate the two nodes connected by the arc. The initial
node follows the word 'back' and the terminal node follows the word
'fore'. With this interpretation, a check will show that the nodes
and arcs above correspond to the graph in Figure 4.

    The reader now has the background necessary to begin using
pcgraph. The next section is intended for those who want to learn
about the algorithm underlying pcgraph.

SECTION 4

THE ALGORITHM

OVERVIEW

The algorithm for generating the prime-constraint graph of a
system consists of three phases: (1) Initialize, (2) Generate, and
(3) Update. The major task of the Initialize phase is to construct
an initial graph from which the second phase proceeds. In this
construction, each two-place constraint supplied as input is
represented by a pair of arcs. In the Generate phase a technique
known as 'resolution' is used to enlarge the graph to the point
where every prime constraint (of arbitrary length) is represented by
a path from the (unique) starting node to the (unique) finishing
node. This enlarged graph, however, also contains, in general,
paths corresponding to constraints that are not prime. The purpose
of the Update phase is to eliminate these 'non-prime' paths. The
result is a graph in which: (1) every path from the starting node to
the finishing node corresponds to a prime constraint and (2) every
prime constraint corresponds to such a path.

Before discussing the three phases, we must first introduce
three concepts that will allow us to state a necessary and
sufficient condition for a clause to be a prime constraint.

PRIME IMPLICANTS

The notion of a 'prime implicant' is familiar to anyone who has
studied switching theory. First of all, an _implicant_ of a set of
states Q is just a term t such that states$(t) \subseteq$ Q. A _prime implicant_
of Q is any implicant of Q not covered by a 'larger' implicant.

One of our principal uses for prime implicants will be as a
convenient representation for an arbitrary set of states. Consider
a system in which the set of variables is {{a0 a1} {b0 b1 b2}
{c0 c1}}. For the set of states

{{a0 b0 c0} {a0 b0 c1} {a0 b1 c0} {a0 b1 c1} {a0 b2 c1} {a1 b2 c1}}

we have the following set of prime implicants:

{{a0 b0 b1} {b2 c1} {a0 c1}}

This set provides a representation that is equivalent to the listing of states.

In what follows, we will be using two binary operations on sets of prime implicants. If A and B are each a set of prime implicants, then <u>sum(A,B)</u> denotes the set of prime implicants for states(A)∪states(B), and <u>product(A,B)</u> denotes the set of prime implicants for states(A)∩states(B).[7] For the choice of variables given above,

sum({{a0 b1 b2} {b1 c1}} {{b0}}) = {{a0} {b0} {b0 b1 c1}}

product({{a0} {b0 b2}} {{a1} {b1}}) = {{a1 b0 b2} {a0 b1}}.

FEX AND BEX

Consider the clause z = <{a0 b1} {}> from the four-stage shift-register described in Section 3. We are interested in the set of states s such that: for each state sequence w in seq(z), ws is <u>not</u> a simulation. There are eight such states:

{{a0 b0 c1 d0} {a0 b0 c1 d1} {a0 b1 c1 d0} {a0 b1 c1 d1}
{a1 b0 c1 d0} {a1 b0 c1 d1} {a1 b1 c1 d0} {a1 b1 c1 d1}}

The result of appending any one of these states to the front end of any state sequence in seq(z) is a nonsimulation. Furthermore, these are the only states with this property. For example, if {a0 b0 c0 d1} is appended to the front end of <{a0 b1 c0 d1} {a0 b0 c1 d0}>, which is in seq(z), the result is a simulation. If we now convert the above set of states into its equivalent representation as a set of prime implicants, we get {{c1}}. This set of prime implicants is known as fex(z) (for <u>f</u>orwards <u>ex</u>clusion). As might be expected, there is also a dual concept for the back end of a clause. For the same clause z, bex(z) = {{a0}}.

----

[7] The 'states' function defined earlier for a single term is extended to a set of terms in the natural way. For a set of terms A,

$$states(A) = \bigcup_{t \in A} states(t)$$

For two sets of terms A and B, we shall say that A <u>covers</u> B if and only if states(A) ⊇ states(B).

23

Our interest in fex and bex is motivated by an important result that provides us with a necessary and sufficient condition for a clause to be a prime constraint.

Theorem: A clause z of length n is a prime constraint if and only if the following three conditions are satisfied for $1 \leq i \leq n$:[8]

1. $z(i)$ is a member of $sum(fex(z^-(i)), bex(z^+(i)))$.

2. $z(i)$ is a member of $fex(z^-(i))$ only for $i=n$.

3. $z(i)$ is a member of $bex(z^+(i))$ only for $i=1$.

The first condition says, in effect, that no term of z can be enlarged without yielding a non-constraint. The second condition says that z cannot be shortened on the front end without producing a non-constraint. And the third condition says that z cannot be shortened on the back end without yielding a non-constraint.

To illustrate the preceding theorem we consider three clauses from our shift-register example. Shown in Figures 6 through 8 are the fex's and bex's for the three clauses. For the clause <{a0} {} {c1}> we see that all three of the conditions in the theorem are satisfied for each of the three terms, and we conclude that this clause is a prime constraint. For the clause <{a0} {a1} {c1}>, however, we see that Condition 1 is not satisfied for $i=2$ since {a1} is not a member of $sum(fex(<{a0}>), bex(<{c1}>))$. And for the clause <{} {b0} {c1}> we see that Condition 3 is not satisfied for $i=2$ since {b0} is a member of $bex(<{c1}>)$ but $i \neq 1$. We conclude that neither of these last two clauses is a prime constraint.

NODES AND ARCS

Throughout the construction of the prime-constraint graph, we shall be dealing with a changing set of ARCS and a changing set of NODES. A node n is defined by two sets of prime implicants, denoted FEX(n) and BEX(n). There are two nodes of special interest, BEND (for Back END) and FEND (for Front END), where,

----

[8] $z^-(i)$ denotes the maximal subclause of z preceding the i'th term. $z^+(i)$ denotes the maximal subclause of z following the i'th term.

24

	i=1	i=2	i=3
$z(i)$	{a0}	{}	{c1}
$z^-(i)$	<>	<{a0}>	<{a0}{}>
$fex(z^-(i))$	{}	{{b1}}	{{c1}}
$z^+(i)$	<{}{c1}>	<{c1}>	<>
$bex(z^+(i))$	{{a0}}	{{b0}}	{}
$sum(fex(z^-(i)),bex(z^+(i)))$	{{a0}}	{{}}	{{c1}}

Figure 6. Fex's and Bex's for z = <{a0} {} {c1}>

	i=1	i=2	i=3
$z(i)$	{a0}	{a1}	{c1}
$z^-(i)$	<>	<{a0}>	<{a0}{a1}>
$fex(z^-(i))$	{}	{{b1}}	{{b0}{c1}}
$z^+(i)$	<{a1}{c1}>	<{c1}>	<>
$bex(z^+(i))$	{{a0}}	{{b0}}	{}
$sum(fex(z^-(i)),bex(z^+(i)))$	{{a0}}	{{}}	{{b0}{c1}}

Figure 7. Fex's and Bex's for z = <{a0} {a1} {c1}>

FEX(BEND) = {}   FEX(FEND) = {{}}
BEX(BEND) = {{}}   BEX(FEND) = {}

An <u>arc</u> a is defined by two nodes and a term, denoted, respectively,
BACK(a), FORE(a), and TERM(a).

The interrelationship between nodes and arcs is expressed by
five properties that are preserved at every step of the algorithm:

	i=1	i=2	i=3
$z(i)$	{}	{b0}	{c1}
$z^-(i)$	<>	<{}>	<{}{b0}>
$fex(z^-(i))$	{}	{}	{{c1}}
$z^+(i)$	<{b0}{c1}>	<{c1}>	<>
$bex(z^+(i))$	{{}}	{{b0}}	{}
$sum(fex(z^-(i)),bex(z^+(i)))$	{{}}	{{b0}}	{{c1}}

Figure 8. Fex's and Bex's for z = <{} {b0} {c1}>

Property 1: For each node n and for each clause z
associated with a path leading from BEND to n, fex(z)
covers FEX(n).

Property 2: For each node n and for each clause z
associated with a path leading from n to FEND, bex(z)
covers BEX(n).

Property 3: For each arc a, TERM(a) is a member of
sum(FEX(BACK(a)), BEX(FORE(a))).

Property 4: For each arc a, TERM(a) is a member of
FEX(BACK(a)) only when FORE(a) = FEND.

Property 5: For each arc a, TERM(a) is a member of
BEX(FORE(a)) only when BACK(a) = BEND.

We should add that when the algorithm terminates the following
two properties, which are stronger versions of Properties 1 and 2,
will hold:

Property 1': For each node n and for each clause z leading
from BEND to n, FEX(n) = fex(z).

Property 2': For each node n and for each clause z leading
from n to FEND, BEX(n) = bex(z).

26

We then see that, at termination, Properties 3, 4, and 5 guarantee that each clause associated with a path leading from BEND to FEND is a prime constraint. (Recall the theorem earlier in this section.)

THE INITIALIZE PHASE

In discussing the algorithm, it will be helpful to consider the following simple example:

    variables = {{a0 a1} {b0 b1} {c0 c1} {d0 d1} {e0 e1}}

    constraints = {<{a0 b0} {c1}>
                    <{c0} {d1}>
                    <{a0} {e1}>
                    <{e0} {d1}>   }

The first step of the algorithm is to construct a graph in which each of the two-place constraints supplied as an input is represented by a path. Let $z=<t_1\ t_2>$ be such an input constraint. Because z is a constraint, it must be that $states(t_1) \subseteq states(bex(<t_2>))$ and $states(t_2) \subseteq states(fex(<t_1>))$. We, therefore, construct this two-arc path to represent z:

```
    BEND       t1      n      t2      FEND
    +--------------->+--------------->+
```

where $BEX(n) = \{t_1\}$ and $FEX(n) = \{t_2\}$. Note that in constructing this subgraph we are preserving Properties 1 - 5.

When this construction is applied to each of the two-place constraints in our example, we get the initial graph shown in Figure 9 where,

    BEX(n1) = {{a0 b0}}   FEX(n1) = {{c1}}
    BEX(n2) = {{c0}}   FEX(n2) = {{d1}}
    BEX(n3) = {{a0}}   FEX(n3) = {{e1}}
    BEX(n4) = {{e0}}   FEX(n4) = {{d1}}

Once the Initialize phase of our algorithm is completed, the techniques of 'resolution' and 'extension' are used to generate the additional nodes and arcs needed to represent an arbitrary prime constraint.

27

```
BEND    {a0 b0}    n1    {c1}    FEND
+--------------->+--------------->+

BEND    {c0}    n2    {d1}    FEND
+--------------->+--------------->+

BEND    {a0}    n3    {e1}    FEND
+--------------->+--------------->+

BEND    {e0}    n4    {d1}    FEND
+--------------->+--------------->+
```

Figure 9. Initial Graph

RESOLUTION

Resolution, also known as consensus, is a technique from propositional logic and switching theory for generating the prime implicants of a Boolean expression. (A variation of this technique is used in the implementation of the sum and product operations.) We show here how a natural extension of resolution can be used to generate the prime constraints of a system. To define this new concept, we must first introduce the 'glb' and 'lub' operations.

Suppose that t1 and t2 are terms such that the intersection of states(t1) and states(t2) is nonempty. Under these circumstances, t1 and t2 have a greatest lower bound with respect to the 'covers' partial order. This bound, which is denoted glb(t1,t2), turns out to be the unique prime implicant of states(t1) $\cap$ states(t2). In the case where the intersection of states(t1) and states(t2) is empty, the glb of t1 and t2 simply does not exist.

Consider a system in which the set of variables is {{a0 a1 a2} {b0 b1}}. We then have, for example,

glb({a0},{b1}) = {a0 b1}

glb({a0},{a1 a2}) does not exist

glb({a0 a1},{a1 a2}) = {a1}

glb({},{a1 a2 b0}) = {a1 a2 b0}.

28

Now let t1 and t2 be two arbitrary terms. The least upper bound (with respect to the 'covers' partial order) of t1 and t2 always exists and is denoted lub(t1,t2). The lub of t1 and t2 is the unique prime implicant of those states s such that every condition in s appears either in a state of states(t1) or in a state of states(t2). Returning to the set of variables given just above, we have,

$$lub(\{a0\},\{b1\}) = \{\}$$

$$lub(\{a0\},\{a2\}) = \{a0\ a2\}$$

$$lub(\{a0\},\{a1\ a2\}) = \{\}$$

$$lub(\{a0\ a1\},\{a1\ a2\}) = \{\}$$

$$lub(\{a0\ b1\},\{a1\ b1\}) = \{a0\ a1\ b1\}$$

$$lub(\{\},\{a1\ b0\}) = \{\}.$$

Let us suppose now that there are two clauses x and y, both of length n, and a variable v such that for some $m \leq n$:

1. $x(m) \cap v$ and $y(m) \cap v$ are incomparable. That is, $x(m) \cap v \not\subseteq y(m) \cap v$ and $y(m) \cap v \not\subseteq x(m) \cap v$.

2. $glb(x(m)-v,y(m)-v)$ exists.

3. $glb(x(i),y(i))$ exists for $1 \leq i \leq n$ and $i \neq m$.

The clause z, of length n, defined as follows is then a resolvent of x and y:

1. $z(m) = lub(x(m) \cap v, y(m) \cap v) \cup glb(x(m)-v,y(m)-v)$

2. $z(i) = glb(x(i),y(i))$ for $1 \leq i \leq n$ and $i \neq m$.

To illustrate this idea, consider the clauses

$$x = \langle\{a0\}\ \{a0\ b0\}\ \{\}\rangle$$

$$y = \langle\{a0\ a2\ b1\}\ \{a2\}\ \{a1\ b0\}\rangle$$

over the variables $\{a0\ a1\ a2\}$ and $\{b0\ b1\}$. Let m = 2 and let v = $\{a0\ a1\ a2\}$. We then see that,

29

$$x(2) \cap v = \{a0\}$$

$$y(2) \cap v = \{a2\}$$

$$lub(x(2) \cap v, y(2) \cap v) = \{a0 \ a2\}$$

$$x(2) - v = \{b0\}$$

$$y(2) - v = \{\}$$

$$glb(x(2) - v, y(2) - v) = \{b0\}$$

$$glb(x(1), y(1)) = \{a0 \ b1\}$$

$$glb(x(3), y(3)) = \{a1 \ b0\}.$$

The clause $z = \langle \{a0 \ b1\} \ \{a0 \ a2 \ b0\} \ \{a1 \ b0\} \rangle$ is thus a resolvent of x and y.

The essential properties of a resolvent are contained in the following result.

Theorem: If z is a resolvent of the clauses x and y, then,

1. $seq(z) \subseteq seq(x) \cup seq(y)$

2. $seq(z) \nsubseteq seq(x)$

3. $seq(z) \nsubseteq seq(y)$

This result is primarily of interest to us for the case where both x and y are constraints.

Corollary: If z is a resolvent of the constraints x and y, then,

1. z is also a constraint.

2. z is not covered by x.

3. z is not covered by y.

Resolution can thus be used to generate new constraints from existing ones.

EXTENSION

As we have just seen, resolution can be applied only to clauses of the same length, and then only when the terms to be resolved are corresponding terms. These restrictions thus prevent the constraints <{a0} {b1}> and <{b0} {c1}> in our shift-register example from being resolved. In resolving constraints, however, we can take advantage of a useful property:

Property: Any extension of a constraint is also a constraint.

Let us now append a {} to the front end of the constraint <{a0} {b1}> and a {} to the back end of the constraint <{b0} {c1}>. The two new constraints,

$$<\{a0\}\ \{b1\}\ \{\}>$$
and
$$<\{\}\ \{b0\}\ \{c1\}>,$$

can then be resolved (using the b1 and b0) to produce the new constraint,

$$<\{a0\}\ \{\}\ \{c1\}>.$$

It can be shown that, together, resolution and extension are sufficient to generate any prime constraint of a system from the set of two-place constraints.

FEX'S AND BEX'S OF A RESOLVENT

Let z be a resolvent of the clauses x and y. We wish to know what the relationship is between the fex's and bex's of z and the fex's and bex's of x and y. As it turns out, that relationship is fairly interesting.

Theorem: If x and y are clauses of length n, and z is obtained from x and y by resolving a variable in the m'th terms of x and y, then,

1. $fex(z^-(i))$ covers $sum(fex(x^-(i)),fex(y^-(i)))$ for $1 \leq i < m$.

2. $fex(z^-(i))$ covers $product(fex(x^-(i)),fex(y^-(i)))$ for $m < i \leq n$.

31

3. $bex(z^+(i))$ covers $sum(bex(x^+(i)), bex(y^+(i)))$
   for $m \leq i \leq n$.

4. $bex(z^+(i))$ covers $product(bex(x^+(i)), bex(y^+(i)))$
   for $1 \leq i < m$.

So we see that the relationship between $fex(z^-(i))$, $fex(x^-(i))$, and $fex(y^-(i))$ depends upon whether $i \leq m$ or $i > m$, and that the relationship between $bex(z^+(i))$, $bex(x^+(i))$, and $bex(y^+(i))$ depends upon whether $i \geq m$ or $i < m$.

Let us consider the clauses $x = <\{a0\}\{b1\}\{\}>$ and $y = <\{\}\{b0\}\{c1\}>$ from the shift-register example. As we noted earlier, the clause $z = <\{a0\}\{\}\{c1\}>$ can be obtained from x and y by resolving the variable $\{b0\ b1\}$ in the second terms of x and y. The above theorem can be checked by comparing the fex's and bex's for x, y, and z as given in Figures 10 and 11.

	i=1	i=2	i=3
$fex(x^-(i))$	{}	{{b1}}	{{}}
$fex(y^-(i))$	{}	{}	{{c1}}
$sum(fex(x^-(i)), fex(y^-(i)))$	{}	{{b1}}	n.a.
$product(fex(x^-(i)), fex(y^-(i)))$	n.a.	n.a.	{{c1}}
$fex(z^-(i))$	{}	{{b1}}	{{c1}}

Figure 10. Fex's for Clauses x, y, and z

THE GENERATE PHASE

In the Generate phase, the ideas of the preceding three sections are used to enlarge the graph produced by the Initialize phase so that every prime constraint (of arbitrary length) is represented by a path from BEND to FEND. The procedure consists of a series of resolutions.

32

	i=1	i=2	i=3
bex(x+(i))	{{a0}}	{}	{}
bex(y+(i))	{{}}	{{b0}}	{}
sum(bex(x+(i)),bex(x+(i)))	n.a.	{{b0}}	{}
product(bex(x+(i)),bex(y+(i)))	{{a0}}	n.a.	n.a.
bex(z+(i))	{{a0}}	{{b0}}	{}

Figure 11. Bex's for Clauses x, y, and z

The first step in a resolution is to find a pair of arcs a and b in the current graph such that for some variable v:

1. $TERM(a) \cap v$ and $TERM(b) \cap v$ are incomparable.

2. $glb(TERM(a)-v, TERM(b)-v)$ exists.

The next step is to find two paths Pa and Pb in the current graph such that:

3. Pa and Pb are the same length.

4. Arc a and Arc b appear in the same relative positions of Pa and Pb, respectively.

5. Either Pa or Pb begins at BEND, and either Pa or Pb ends at FEND.

6. For each pair of arcs a'≠a and b'≠b that appear in the same relative positions of Pa and Pb, respectively, $glb(TERM(a'), TERM(b'))$ exists.

If all six requirements are met, then a new path Pc is tentatively constructed. (It will be added to the graph only if certain additional requirements are satisfied.) In the construction, each pair of corresponding nodes in Pa and Pb is transformed into a node of Pc, and each pair of corresponding arcs in Pa and Pb is transformed into an arc of Pc. The rules for the creation of this new path are as follows:

33

1. Corresponding nodes m and n in Pa and Pb that <u>precede</u> Arcs a and b are transformed into the node q where,

$$FEX(q) = sum(FEX(m), FEX(n))$$

$$BEX(q) = product(BEX(m), BEX(n))$$

2. Corresponding nodes m and n in Pa and Pb that <u>follow</u> Arcs a and b are transformed into the node q where,

$$FEX(q) = product(FEX(m), FEX(n))$$

$$BEX(q) = sum(BEX(m), BEX(n))$$

3. Arcs a and b are transformed into the arc c where,

$$TERM(c) = lub(TERM(a) \cap v, TERM(b) \cap v) \\ \cup glb(TERM(a)-v, TERM(b)-v)$$

BACK(c) is the result of transforming
        BACK(a) and BACK(b).

FORE(c) is the result of transforming
        FORE(a) and FORE(b).

4. Corresponding arcs a'≠a and b'≠b in Pa and Pb are transformed into the arc c' where,

$$TERM(c') = glb(TERM(a'), TERM(b'))$$

BACK(c') is the result of transforming
        BACK(a') and BACK(b').

FORE(c') is the result of transforming
        FORE(a') and FORE(b').

Having constructed Pc, we must now check to see if Properties 1-5 (on p. 21) are satisfied by the nodes and arcs of Pc. In the case of Properties 1 and 2, our method of construction, together with the theorem in the preceding subsection, guarantees that these two properties will be satisfied. For Properties 3, 4, and 5, however, there is no such assurance, and these properties must be individually checked for each arc of Pc. If Properties 3, 4, and 5 are satisfied, then the path Pc is added to the graph. (Note that some parts, or all, of Pc may already exist.)

The process of resolution just described is repeated for other arcs and other paths. The process continues until no new nodes and

no new arcs can be created. At that point the Generate phase is terminated.

To illustrate the process of resolution, let us return to the example considered for the Initialize phase. Let a be the arc,

```
        n1        {c1}        FEND
        +---------------->+
```

and let b be the arc,

```
        BEND        {c0}        n2
        +---------------->+
```

and let v be the variable {c0 c1}. We see immediately that TERM(a)∩v and TERM(b)∩v are incomparable, and that glb(TERM(a)-v,TERM(b)-v) exists. Now let Pa be the path consisting of just Arc a, and Pb the path consisting of just Arc b. We observe immediately that Requirements 3, 4, and 5 are satisfied. Requirement 6 is trivially satisfied because there are no other arcs besides a and b. In constructing the path Pc, the pair of nodes n1 and BEND are transformed into the node n1, and the pair of nodes FEND and n2 are transformed into the node n2. (Recall that FEX(BEND)={}, BEX(BEND)={{}}, FEX(FEND)={{}}, and BEX(FEND)={}.) Arcs a and b are transformed into the arc c where TERM(c)={}, BACK(c)=n1, and FORE(c)=n2. The result is the path Pc:

```
        n1        {}        n2
        +---------------->+
```

Since FEX(n1)={{c1}} and BEX(n2)={{c0}}, we see that Properties 3, 4, and 5 are satisfied. And so the arc comprising Pc is added to our graph.

The graph at the end of the Generate phase, when all possible resolutions have been exhausted, is shown in Figure 12.

THE UPDATE PHASE

Although the graph produced at the end of the Generate phase has a path for every prime constraint, the converse is not

35

```
     BEND    {a0 b0}    n1    {c1}    FEND
      +--------------->+--------------->+
                       |
                       |{}
                       |
                       ↓
     BEND     {c0}           {d1}    FEND
      +--------------->+--------------->+
                       n2



     BEND     {a0}     n3    {e1}    FEND
      +--------------->+--------------->+
                       |
                       |{}
                       |
                       ↓
     BEND     {e0}           {d1}    FEND
      +--------------->+--------------->+
                       n4
```
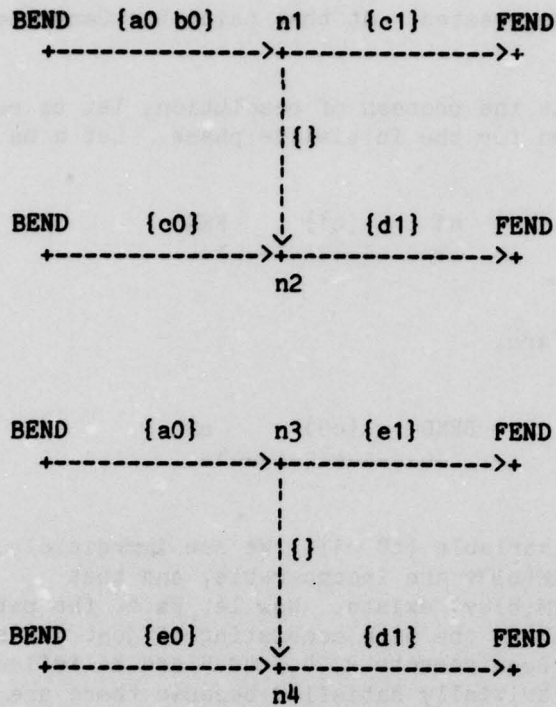
Figure 12. Graph at End of Generate Phase

necessarily true. It is possible for a path from BEND to FEND to
represent a constraint that is not prime. The graph in Figure 12
provides an illustration of this point. Consider the two paths:

```
    BEND    {a0 b0}    n1     {}     n2    {d1}    FEND
     +--------------->+--------------->+--------------->+

    BEND     {a0}     n3     {}     n4    {d1}    FEND
     +--------------->+--------------->+--------------->+
```
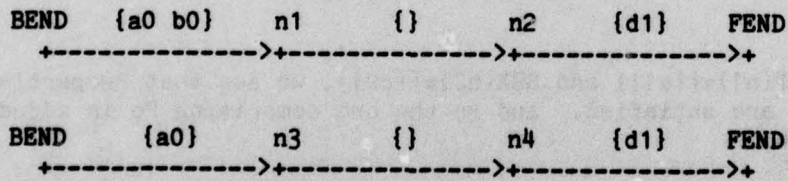
The constraints associated with these two paths are
<{a0 b0} {} {d1}> and <{a0} {} {d1}>. Since the first constraint is
properly covered by the second, the first constraint cannot be
prime.

The problem stems from the fact that the FEX of a node may not be equal to the fex of each path leading from BEND to that node. And similarly, the BEX of a node may not be equal to the bex of each path leading from that node to FEND. In the case of our example, BEX(n1) = {{a0 b0}} but bex(<{} {d1}>) = {{a0}}. The purpose of the Update phase is to eliminate this discrepancy.

Suppose that P1 and P2 are two paths in the current graph. P1 begins at BEND and ends at Node m, and is associated with the clause x. (See Figure 13.) P2 has a terminal node of n, and is associated with the clause z. Now suppose that x and z are the same length, and that x covers z. It follows that fex(z) covers fex(x). From Property 1 we know that fex(x) covers FEX(m), and, therefore, fex(z) must cover FEX(m). The question is whether or not FEX(n) covers FEX(m). If it does, then there is no problem. If, however, FEX(n) does not cover FEX(m), then it is necessary to change the terminal node of P2.

```
        BEND                x                   m
P1:     +----------->+    . . .    +----------->+



                          z                     n
P2:     +----------->+    . . .    +----------->+
```
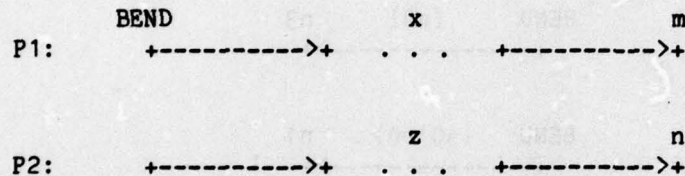
Figure 13. Updating a FEX

The transformation is represented in Figure 14. A new node n' is created where FEX(n') = sum(FEX(n),FEX(m)) and BEX(n') = BEX(n). (Note that n' may be an already existing node.) The last arc of P2 is replaced by an arc leading to n', and emergent arcs are added to n' to 'duplicate' the emergent arcs of n. Once the transformation is completed, the newly-created arcs are checked to see that they satisfy Properties 3, 4, and 5. Any arcs not satisfying these properties are deleted. Finally, any arc or node that is no longer part of a path leading from BEND to FEND is removed. This entire process is repeated until no further 'updatings' of FEX's are possible. A similar procedure is then performed for the BEX's of nodes. When this task is completed, the algorithm terminates, and the resulting graph is the prime-constraint graph for the system that was supplied as input.

37
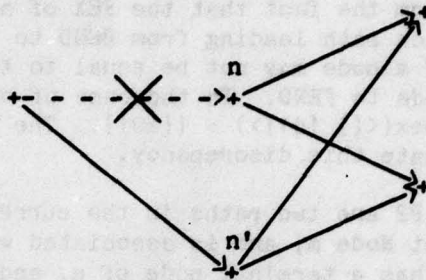
Figure 14. Splitting a Node

To illustrate the process of updating, we return to our **sample** system. Let P1 and P2 be the two paths:

$$
\begin{array}{cccc}
 & \text{BEND} & \{a0\} & \text{n3} \\
\text{P1:} & \texttt{+--------------->+} &
\end{array}
$$

$$
\begin{array}{cccc}
 & \text{BEND} & \{a0\ b0\} & \text{n1} \\
\text{P2:} & \texttt{+--------------->+} &
\end{array}
$$
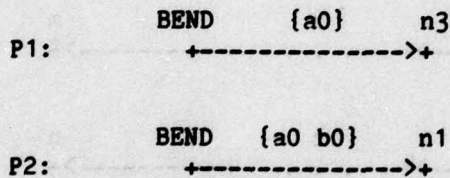
Since P1 and P2 meet the necessary requirements and since
FEX(n1)={{c1}} does not cover FEX(n3)={{e1}}, we proceed with an
update. A new node n5 is created where FEX(n5) =
sum({{c1}},{{e1}}) = {{c1} {e1}} and BEX(n5) = BEX(n1) = {{a0 b0}}.
After the required arcs are attached to n5 and after the last (and
only) arc of P2 is deleted, we have the structure shown in
Figure 15. We see that Node n1 and its two emergent arcs are no
longer contained in paths leading from BEND to FEND, and so they are
deleted.

We next consider the following two paths:

$$
\begin{array}{ccccc}
\text{n3} & \{\} & \text{n4} & \{d1\} & \text{FEND} \\
\texttt{+--------------->+--------------->+}
\end{array}
$$

$$
\begin{array}{ccccc}
\text{n5} & \{\} & \text{n2} & \{d1\} & \text{FEND} \\
\texttt{+--------------->+--------------->+}
\end{array}
$$
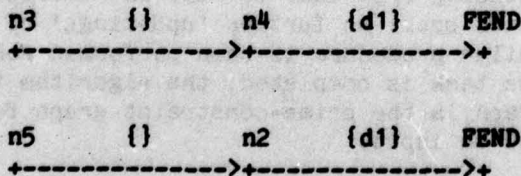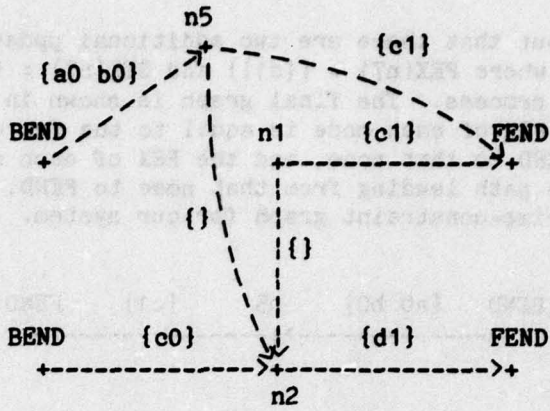
Figure 15. Updated Graph

We observe that BEX(n5) = {{a0 b0}} does not cover BEX(n3) = {{a0}}.
It is therefore necesssary to create a new node n6 where FEX(n6) =
FEX(n5) = {{c1} {e1}} and BEX(n6) = sum({{a0 b0}}, {{a0}}) = {{a0}}.
The resulting structure is shown in Figure 16. We note, however,
that the arc leading from BEND to n6 does not satisfy Property 3
since {a0 b0} is not a member of {{a0}}. This arc, and also the arc
leading from n6 to n4, are therefore deleted.
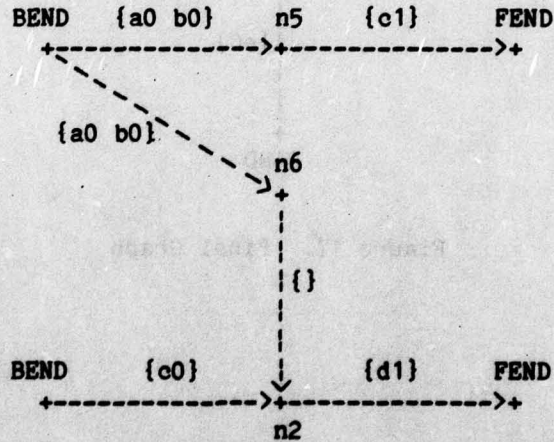
Figure 16. Newly Updated Graph

It turns out that there are two additional updates to be made.
A new node n7, where $FEX(n7) = \{\{d1\}\}$ and $BEX(n7) = \{\{c0\} \{e0\}\}$, is
created in the process. The final graph is shown in Figure 17. In
this graph the FEX of each node is equal to the fex of each path
leading from BEND to that node, and the BEX of each node is equal to
the bex of each path leading from that node to FEND. We have thus
produced the prime-constraint graph for our system.

```
 BEND     {a0 b0}     n5       {c1}      FEND
  +--------------->+--------------->+
```

```
 BEND      {a0}       n3       {e1}      FEND
  +--------------->+--------------->+
                   |{}
                   |
 BEND      {c0}     n7 {d1}             FEND
  +--------------->+--------------->+
                   ^
                   |
                   |{e0}
                   |
                   +
                  BEND
```
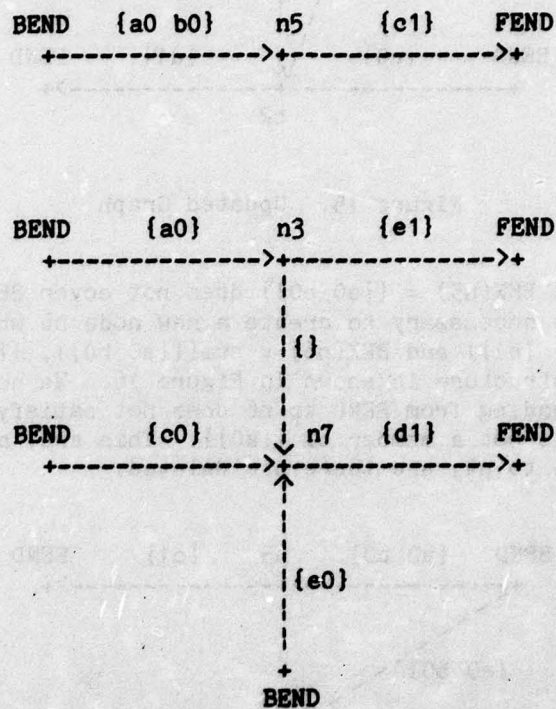
Figure 17. Final Graph

40

SECTION 5

CONCLUSIONS

ACCOMPLISHMENTS

In this report we have described a new technique for security validation based on the concept of a prime constraint. We have also documented a computer program that automates an important part of the analysis: the generation of a system's prime-constraint graph. We have seen that the set of prime constraints, as represented by the prime-constraint graph, allows us to determine what deductions are possible in a system. Validation of security is then just a matter of determining which of the deductions, if any, constitute a compromise.

There are various criteria for making that determination. In the case of multi-level security, the most straightforward approach is to assign a security level to each of the 'visible' variables, and then check for a prime constraint that shows a variable at one security level being compromised by other variables at lower security levels. The important thing to note here is that a 'deduction' is now the means for determining compromise rather than the existence of an 'information flow'.

COMPLEXITY

The most pressing problem that needs to be addressed is complexity. We know that while the prime-constraint graph grows quite modestly for some systems,[9] such as shift registers and many control structures, it grows exponentially for other systems, such as adders and multipliers. It appears that this exponential growth is most likely to occur in those systems corresponding to Boolean functions for which the number of prime implicants grows exponentially (i.e., adders and multipliers). There is also a large class of systems for which we simply do not know how serious the complexity problem is. For example, we have no idea how complex the prime-constraint graph of a typical operating system would be.

----

[9] The growth of the prime-constraint graph is measured relative to the number of two-place prime constraints.

41

There are a number of ways of dealing with complexity. One can simply choose to model a system in less detail, or one can try to configure a system so that the essential aspects of behavior become tractable. (Partitioning a system into separate 'control' and 'data flow' components is one approach.) None of these techniques, however, really solves the problem of complexity since there will be everyday systems that remain completely intractable. The only real solution, if there is one, will have to come from further research and a deeper understanding of the theory of constraints.

HIERARCHICAL VALIDATION

Another problem, which is related to the complexity of the prime-constraint graph, is the computational load placed on the program for generating that graph. In some cases, it is possible for the execution time of the program to be enormously long even though the final prime-constraint graph is of modest size.

In a hierarchically-structured system it is possible to get around this problem through a 'hierarchical validation'. One begins by generating the 'external prime-constraint graph' for each of the lowest-level modules. This graph is obtained from the complete prime-constraint graph by eliminating all arcs having conditions belonging to 'hidden' variables. These graphs are then 'merged' by applying the Generate and Update phases described in Section 4 to the composite graph. The process is repeated for each successive level. The final result is the external prime-constraint graph for the top-most module(s). This final graph represents those prime constraints that involve only externally-visible variables. These prime constraints are the only ones we really need be concerned with in order to perform a security validation. What makes this approach attractive, in contrast to the prevailing methodology for security validation, is that the same concepts and the same techniques are applicable across all levels.

SUPPORTING TOOLS

In order to make the tool described in this report more readily accessible, we have to develop programs for both preprocessing and postprocessing. Preprocessing is needed to convert a system description in an applications-oriented language, such as a program-specification language or a hardware-description language, into the language of constraints. Postprocessing is needed to analyze the prime-constraint graph for various properties. For security validation, this analysis entails checking for compromise.

Once the preprocessing and postprocessing programs are implemented, we will then have a completely automated tool for security validation.

REFERENCES

1. Denning, D.E., "A Lattice Model of Secure Information Flow," <u>Comm</u>. <u>of</u> <u>the</u> <u>ACM</u>, Vol. 19, No. 5, May 1976, pp. 236-242.

2. Denning, D.E. and Denning, P.J., "Certification of Programs for Secure Information Flow," <u>Comm</u>. <u>of</u> <u>the</u> <u>ACM</u>, Vol. 20, No. 7, July 1977, pp. 504-513.

3. Furtek, F.C., "Constraints and Compromise," <u>Foundations of</u> <u>Secure</u> <u>Computation</u>, Academic Press, N.Y., 1978, pp. 189-204.

4. Furtek, F.C., <u>A</u> <u>Theory</u> <u>of</u> <u>Constraints</u>, M78-204, The MITRE Corporation, November 1977.

5. Furtek, F.C., <u>Constraints</u>, M78-205, The MITRE Corporation, December 1977.

6. Millen, J.K., "Constraints and Multilevel Security," <u>Foundations</u> <u>of</u> <u>Secure</u> <u>Computation</u>, Academic Press, N.Y., 1978, pp. 205-222.

7. Millen, J.K., <u>Causal</u> <u>System</u> <u>Security</u>, ESD-TR-78-171, Electronic Systems Division, AFSC, Hanscom AFB, MA, October 1978.

APPENDIX A

MODULE DESCRIPTIONS

In each module description, an interpretation is provided for each argument. The LISP structure corresponding to each interpretation is as follows:

set - list
condition - atomic symbol (with 'variable' property)
variable - atomic symbol (with 'resolved' and 'unresolved' properties)
term - set of conditions
node - atomic symbol (with 'fex', 'bex', 'fore', and 'back' properties)
arc - atomic symbol (with 'fore' and 'back' properties)
pnode - atomic symbol (with 'fex', 'bex', 'fore' or 'back', and 'flag' properties)

The following descriptions assume that the reader is familiar with the programming language LISP and with the algorithm described in Section 4.

union2   2 arguments: each argument: a set

    (union2 X Y) returns the union of X and Y.

union1   1 argument   argument: a set of sets

    (union1 Q) returns the union of the sets in Q.

intersect2   2 arguments   each argument: a set

    (intersect X Y) returns the intersection of X and Y.

intersect1   1 argument   argument: a set of sets

    (intersect1 Q) returns the intersection of the sets in Q.

45

subtract   2 arguments   each argument: a set

    (subtract X Y) returns X - Y.

includes   2 arguments   each argument: a set

    (includes X Y) returns t if X contains Y, and nil otherwise.

same   2 arguments   each argument: a set

    (same X Y) returns t if X and Y represent the same set, and nil
    otherwise.

like   2 arguments   each argument: a set of sets

    Identical to 'same' except that each argument is treated as a
    set of sets.

pairs   2 arguments   each argument: a set

    (pairs X Y) returns the Cartesian product of X and Y.

varsyms   1 argument   argument: a term

    (varsyms trm) returns a list of those variables with conditions
    in trm.

inout   2 arguments   1st argument: a variable
                                         2nd argument: a term

    (inout var trm) returns a dotted-pair, the car of which is a
    the set of conditions in trm belonging to var, and the cdr of
    which is the set of conditions in trm not belonging to var.

46

covers   2 arguments   each argument: a term

    (covers trm1 trm2) returns t if trm1 covers trm2, and nil
    otherwise.

embraces   2 arguments   1st argument: a set of terms
                                          2nd argument: a term

    (embraces Q trm) returns t if one of the terms in Q covers trm,
    and nil otherwise.

contains   2 arguments   each argument: a set of terms

    (contains X Y) returns t if X covers Y, and nil otherwise. It
    is assumed that X is the set of all prime implicants for the
    set of states covered by X.

glb   2 arguments   each argument: a term

    (glb trm1 trm2) returns the greatest lower bound of trm1 and
    trm2. If there is no glb, then 'no' is returned.

resolve   3 arguments   1st argument: a variable
                                            2nd & 3rd arguments: terms

    (resolve var trm1 trm2) returns the resolvent of trm1 and trm2
    using var as the resolving variable.

resolvents   2 arguments   1st argument: a term
                                            2nd argument: a set of terms

    (resolvents trm Q) returns a dotted-pair. Its car is the set
    of all resolvents between trm and each of the terms in Q. Its
    cdr is the set of all terms in Q not covered by trm.

primes   2 arguments   each argument: a set of terms

    (primes X Y) returns the set of prime implicants for the union of states(X) and states(Y).

sum   2 arguments   each argument: a set of terms

    Identical to 'primes' except that 'sum' assumes that the terms in each argument are relatively prime.

product   2 arguments   each argument: a set of terms

    (product X Y) returns the set of prime implicants for the intersection of states(X) and states(Y).

setvariable   1 argument   argument: a set of conditions

    (setvariable v) creates a variable whose value is v. The variable is given two properties, 'resolved' and 'unresolved', which are set to nil. The new variable is returned.

getnode   4 arguments   1st & 3rd arguments: 'fex' & 'bex'
                                                    or
                                      'bex' & 'fex'
                            2nd & 4th arguments: sets of terms

    (getnode ex1 x1 ex2 x2) causes a search for an existing node whose ex1 property is x1, and whose ex2 property is x2. If no such node is found, then one is created. The name of the existing or newly-created node is returned.

```
getarc          5 arguments      1st argument: a term
                                 2nd & 4th arguments: 'fore' & 'back'
                                                              or
                                                      'back' & 'fore'
                                 3rd & 5th arguments: nodes
```

(getarc trm dir1 d1 dir2 d2) causes a search for an existing
arc whose value is trm, whose dir1 is d1, and whose dir2 is d2.
If no such arc is found, then one is created. The name of the
existing or newly-created arc is returned.

```
init            1 argument       argument: a two-term list
```

Interprets its argument as a two-place constraint, and creates
a two-arc path from BEND to FEND.

```
initialize      2 arguments      1st argument: a set of sets
                                                of conditions
                                 2nd argument: a set of two-term
                                               lists
```

Performs a 'setvariable' for each set of conditions in the
first argument. Creates the BEND and FEND nodes. Performs an
'init' for each of the two-term lists in the second argument.

```
check           2 arguments      1st argument: a term
                                 2nd argument: a set of terms
```

(check trm Q) returns t if trm is not properly covered by any
of the terms in Q, and nil otherwise.

```
pairnodes       5 arguments      1st & 2nd arguments: nodes
                                 3rd, 4th, & 5th arguments:
                                              'bex', 'fex', & 'BEND'
                                                      or
                                              'fex', 'bex', & 'FEND'
```

(pairnodes m n ex1 ex2 end) computes the 'transform' of m and
n. The two choices for ex1, ex2, and end determine whether the
two nodes are assumed to precede or follow the arcs being

resolved by nresolve. If either m or n is end, then the other node is returned. Otherwise, the fex and bex of a potential new node are computed. The ex1 property of this potential node is set to the sum of m's and n's ex1 properties, and the ex2 property is set to the product of m's and n's ex2 property. If the ex2 property of this potential node is nil, then 'bad' is returned. Otherwise, an atomic symbol, called a 'pnode', is created from ex1, m, and n. If this pnode did not previously exist, or if it had existed but had not been 'merged', then the pnode is returned as the value of pairnodes. If, however, the pnode had existed and had been merged, then the value of the pnode will be the transform of m and n, and it is this node that is returned by pairnodes.

complete	6 arguments	1st argument: a pnode
		2nd argument: a two-arc set
		3rd-6th arguments:
		'bex', 'fex', 'back', & 'BEND'
		or
		'fex', 'bex', 'fore', & 'FEND'

(complete y p ex1 ex2 dir end) attempts to complete the pair of paths that led to y being created. The two arcs in p are the candidates for the next step in this completion. First, the glb of the terms associated with these two arcs is computed. If it does not exist, then complete indicates that no completions are possible by returning without taking any further action. Otherwise, pairnodes is called for the two dir nodes associated with the two arcs in p. If 'bad' is returned, then complete terminates immediately indicating that no completions are possible. Otherwise, two tests are made to insure that the two arcs are suitable for being merged. If either test fails, then complete again returns immediately. If all hurdles are overcome, then the symbol returned by pairnodes is checked. If it is a node or a 'merged' pnode, then a pointer to this node or pnode is added to y in anticipation of a possible merge, and control is returned. If, however, the symbol returned by pairnodes is an 'unmerged' pnode, then complete is called for that symbol and each pair of next-possible arcs. If at the end of these calls the pnode has accumulated at least one pointer, then a pointer to this pnode is added to y. The program then terminates.

merge   3 arguments   1st argument: a pnode
                                2nd & 3rd arguments: 'back' & 'fore'
                                                         or
                                                     'fore' & 'back'

(merge z dir1 dir2) first causes a node to be created for the
fex and bex supplied by z. The value of z is set to this node.
Then each pointer of z is checked to see if it points to a
node, a 'merged' pnode, or an 'unmerged' pnode. The 'unmerged'
pnode is merged immediately. The appropriate arc is then added
between the node associated with the pointer and the node
associated with z. The program then returns.

ncovered   6 arguments   1st argument: an arc
                                2nd argument: a set of arcs
                                3rd-5th arguments: 'back' & 'fore'

(ncovered a Q dir1 dir2 ex done) returns t if a prefix of every
path beginning with Arc a and proceeding in the dir1 direction
is covered by a path beginning with an arc in Q, also
proceeding in the dir1 direction, and terminating at either
BEND or FEND. nil is returned otherwise.

remove   3 arguments   1st argument: an arc
                                2nd & 3rd arguments: 'back' & 'fore'
                                                         or
                                                     'fore' & 'back'

(remove a dir1 dir2) removes Arc a and all those nodes and arcs
in the dir1 direction that are no longer part of a path from
BEND to FEND.

prune   3 arguments   1st-3rd arguments:
                                           'back', 'fore', & 'fex'
                                                     or
                                           'fore', 'back', & 'bex'

Prunes from the current graph those newly-created arcs and
those arcs having a node in common with a newly-created arc
that are superfluous. Note: prune is included only for the
sake of efficiency and is not essential to the algorithm.

51

nresolve   3 arguments   1st argument: a variable
                                       2nd & 3rd arguments: arcs

(nresolve var a1 a2) attempts a resolution centered around Arcs
a1 and a2 with var as the resolving variable. Complete finds
the necessary pairs of paths and merge performs the actual
'merging' of these paths.

generate   1 argument   argument: a variable

(generate v) causes (nresolve v a1 a2) to be performed for each
pair of arcs a1 and a2 that have not been previously nresolved
and that have terms with at least one condition belonging to v.

split   7 arguments   1st & 2nd arguments: nodes
                                       3rd-7th arguments:
                                       'bex', 'fex', 'back', 'fore', & 'FEND'
                                                      or
                                       'fex', 'bex', 'fore', 'back', & 'BEND'

(split m n ex1 ex2 dir1 dir2 end) looks for a pair of nodes
nextm and nextn for which there are two arcs a and b such that:
m is the dir2 of a, n is the dir2 of b, nextm is the dir1 of a,
nextn is the dir1 of b, and the term for a covers the term for
b. The ex2 of nextm is then used to 'update' the ex2 of nextn.

update   5 arguments   1st-5th arguments:
                                       'bex', 'fex', 'back', 'fore', & 'FEND'
                                                      or
                                       'fex', 'bex', 'fore', 'back', & 'BEND'

Updates the fex's and bex's of the nodes produced by the
generate phase.

52

pcgraph   2 arguments   1st argument: a set of sets
                                                     of conditions
                                     2nd argument: a set of two-term
                                                     lists

(pcgraph V C) generates the prime constraint graph for the set
of variables V and the set of two-place constraints C. Initial
function called by the user.

expand   1 argument   argument: 'ARCS'

Expands the set of arcs. Final function called by the user.

APPENDIX B

PCGRAPH LISTING

(ERRATUM: The use of 'fex' and 'bex' in this listing is reversed
from that in the text. A 'fex' in the listing corresponds to a
'bex' in the text, and vice versa.)

```
(defun union2 (X Y)
  (cond ((null Y) X)
    (t (do ((A X (cdr A)) (B Y))
        ((null A) B)
      (cond ((not (member (car A) Y)) (setq B (cons (car A) B))))))))

(defun union1 (Q)
  (do ((A Q (cdr A)) (B nil))
    ((null A) B)
    (setq B (union2 (car A) B))))

(defun intersect2 (X Y)
  (cond ((null Y) nil)
    (t (do ((A X (cdr A)) (B nil))
        ((null A) B)
      (cond ((member (car A) Y) (setq B (cons (car A) B))))))))

(defun intersect1 (Q)
  (cond ((null Q) ARCS)
    (t (do ((A (cdr Q) (cdr A)) (B (car Q)))
        ((null A) B)
      (setq B (intersect2 (car A) B))))))

(defun subtract (X Y)
  (cond ((null Y) X)
    (t (do ((A X (cdr A)) (B nil))
        ((null A) B)
      (cond ((not (member (car A) Y)) (setq B (cons (car A) B))))))))

(defun includes (X Y)
  (do ((A Y (cdr A)))
    ((null A) t)
    (cond ((not (member (car A) X)) (return nil)))))

(defun same (X Y)
  (and (includes X Y) (includes Y X)))

(defun like (X Y)
  (and (do ((A X (cdr A)))
        ((null A) t)
        (cond ((do ((B Y (cdr B))
            ((null B) t)
            (cond ((same (car A) (car B)) (return nil))) (return nil)))))
    (do ((A Y (cdr A)))
        ((null A) t)
        (cond ((do ((B X (cdr B))
            ((null B) t)
            (cond ((same (car A) (car B)) (return nil))) (return nil)))))))
```

```lisp
(defun pairs (X Y)
  (do ((A (mapcar '(lambda (y) (mapcar '(lambda (x) (list x y)) X)) Y) (cdr A)) (B nil))
      ((null A) B)
    (setq B (append (car A) B))))

(defun varsyms (trm)
  (do ((A trm (cdr A)) (B nil))
      ((null A) B)
    (setq B (union2 B (list (get (car A) 'variable))))))

(defun inout (var trm)
  (do ((A trm (cdr A)) (x) (in nil) (out nil))
      ((null A) (cons in out))
    (setq x (car A))
    (cond ((equal (get x 'variable) var) (setq in (cons x in)))
          (t (setq out (cons x out))))))

(defun covers (trm1 trm2)
  (cond ((null trm1) t)
        (t (do ((A trm2 (cdr A)) (U (varsyms trm1)) (V nil) (v))
               ((null A) (includes V U))
             (setq v (get (car A) 'variable))
             (cond ((member v U) (cond ((not (member (car A) trm1)) (return nil)))
                   (setq V (union2 (list v) V))))))))

(defun embraces (Q trm)
  (do ((A Q (cdr A)))
      ((null A) nil)
    (cond ((covers (car A) trm) (return t)))))

(defun contains (X Y)
  (do ((A Y (cdr A)))
      ((null A) t)
    (cond ((not (embraces X (car A))) (return nil)))))

(defun glb (trm1 trm2)
  (do ((A (intersect2 (varsyms trm1) (varsyms trm2)) (cdr A)) (x1 trm1) (x2 trm2) (y nil) (z1) (z2) (z))
      ((null A) (append x1 x2 y))
    (setq z1 (inout (car A) x1))
    (setq z2 (inout (car A) x2))
    (setq z (intersect2 (car z1) (car z2)))
    (cond ((null z) (return 'no)))
    (setq y (append z y))
    (setq x1 (cdr z1))
    (setq x2 (cdr z2))))
```

```lisp
(defun resolve (var trm1 trm2)
  (prog (x1 x2 r1 r2 r g)
    (setq x1 (inout var trm1))
    (setq r1 (car x1))
    (setq x2 (inout var trm2))
    (setq r2 (car x2))
    (cond ((or (includes r1 r2) (includes r2 r1)) (return 'no)))
    (setq g (glb (cdr x1) (cdr x2)))
    (cond ((equal g 'no) (return 'no)))
    (setq r (union2 r1 r2))
    (cond ((same r (eval var)) (setq r nil)))
    (return (append r g))))

(defun resolvents (trm Q)
  (do ((A Q (cdr A)) (B nil) (new nil) (q))
      ((null A) (cons new B))
    (setq q (car A))
    (cond ((not (covers trm q)) (do ((V (intersect2 (varsyms trm) (varsyms q)) (cdr V)) (r))
                                    ((null V) (setq B (cons q B)))
                                  (setq r (resolve (car V) trm q))
                                  (cond ((not (equal r 'no)) (setq new (cons r new))))))))

(defun primes (X Y)
  (do ((A X) (B Y) (J) (trm))
      ((null A) B)
    (setq trm (car A))
    (setq A (cdr A))
    (cond ((not (embraces B trm)) (setq J (resolvents trm B))
          (setq A (append (car J) A))
          (setq B (cons trm (cdr J)))))))

(defun sum (X Y)
  (cond ((null X) Y)
        ((null Y) X)
        (t (do ((A X (cdr A)) (B Y) (C nil) (D nil) (J))
               ((null A) (primes D (append B C)))
             (cond ((not (embraces B (car A))) (setq C (cons (car A) C))
                    (setq J (resolvents (car A) B))
                    (setq D (append (car J) D))
                    (setq B (cdr J))))))))

(defun product (X Y)
  (cond ((equal X '(nil)) Y)
        ((equal Y '(nil)) X)
        (t (do ((A (pairs X Y) (cdr A)) (B nil) (trm))
               ((null A) (primes B nil))
             (setq trm (glb (caar A) (cadar A)))
             (cond ((not (equal trm 'no)) (setq B (cons trm B))))))))
```

```lisp
(defun setvariable (v)
  (prog (sym)
    (setq sym (intern (gensym 'v)))
    (set sym v)
    (putprop sym nil 'resolved)
    (putprop sym nil 'unresolved)
    (mapc '(lambda (x) (putprop x sym 'variable)) v)
    (setq VARSYMS (cons sym VARSYMS))
    (return sym)))

(defun getnode (ex1 x1 ex2 x2)
  (do ((N NODES (cdr N)) (n))
      ((null N) (setq n (intern (gensym 'n)))
                (set n nil)
                (putprop n nil 'fore)
                (putprop n nil 'back)
                (putprop n x1 ex1)
                (putprop n x2 ex2)
                (setq NODES (cons n NODES))
                n)
    (setq n (car N))
    (cond ((and (like x1 (get n ex1)) (like x2 (get n ex2))) (return n)))))

(defun getarc (trm dir1 d1 dir2 d2)
  (prog (b f z)
    (putprop 'z d1 dir1)
    (putprop 'z d2 dir2)
    (setq b (get 'z 'back))
    (setq f (get 'z 'fore))
    (setq z (intern (make_atom (catenate b f))))
    (return (do ((A (get z 'arcs) (cdr A)) (a))
                ((null A) (setq a (intern (gensym 'a)))
                          (set a trm)
                          (putprop a f 'fore)
                          (putprop a b 'back)
                          (putprop b (cons a (get b 'fore)) 'fore)
                          (putprop f (cons a (get f 'back)) 'back)
                          (putprop z (cons a (get z 'arcs)) 'arcs)
                          (mapc '(lambda (v) (putprop v (append (get v 'unresolved) (list a)) 'unresolved))
                                (varsyms trm))
                          (setq NEWARCS (cons a NEWARCS))
                          (setq ARCS (cons a ARCS))
                          a)
              (cond ((covers (eval (car A)) trm) (return (car A))))))))
```

58

```
(defun init (c)
  (prog (n)
    (setq n (getnode 'fex (list (car c)) 'bex (list (cadr c))))
    (getarc (car c) 'back 'BEND 'fore n)
    (getarc (cadr c) 'back n 'fore 'FEND)))

(defun initialize (V C)
  (setq VARSYMS nil)
  (mapc 'setvariable V)
  (setq NODES nil)
  (setq ARCS nil)
  (setq BEND nil)
  (putprop 'BEND nil 'bex)
  (putprop 'BEND '(nil) 'fex)
  (setq FEND nil)
  (putprop 'FEND '(nil) 'bex)
  (putprop 'FEND nil 'fex)
  (mapc 'init C))

(defun check (trm Q)
  (do ((A Q (cdr A)))
      ((null A) t)
    (cond ((covers (car A) trm) (cond ((covers trm (car A)) (return t))
                                      (t (return nil)))))))

(defun pairnodes (m n ex1 ex2 end)
  (prog (z)
    (cond ((equal m end) (return n))
          ((equal n end) (return m))
          ((equal m n) (return m)))
    (setq z (intern (make_atom (catenate ex1 m n))))
    (cond ((null (plist z)) (setq z (intern (make_atom (catenate ex1 n m))))
          (cond ((null (plist z)) (set z nil)
                 (putprop z (sum (get m ex1) (get n ex1)) ex1)
                 (putprop z (product (get m ex2) (get n ex2)) ex2)
                 (putprop z t 'flag)
                 (cond ((null (get z ex2)) (set z 'bad)))))))
    (cond ((null (eval z)) (return z))
          (t (return (eval z))))))
```

59

```lisp
(defun complete (y p ex1 ex2 dir end)
  (prog (a1 a2 trm z)
    (setq a1 (car p))
    (setq a2 (cadr p))
    (setq trm (glb (eval a1) (eval a2)))
    (cond ((equal trm 'no) (return 'exit1)))
    (setq z (pairnodes (get a1 dir) (get a2 dir) ex1 ex2 end))
    (cond ((equal z 'bad) (return 'exit2))
          ((not (check trm (sum (get y ex2) (get z ex1)))) (return 'exit3))
          ((embraces (get z ex1) trm) (return 'exit4))
          ((and (get z 'flag)
                (null (get z dir))) (mapc '(lambda (p) (complete z p ex1 ex2 dir end))
                                          (pairs (get (get a1 dir) dir) (get (get a2 dir) dir))))
          ((null (get z dir)) (set z 'bad)
                              (return 'exit5)))))
  (putprop y (cons (list trm z) (get y dir)) dir)
  (return 'exit6)))

(defun merge (x dir1 dir2)
  (set z (getnode 'fex (get z 'fex) 'bex (get z 'bex)))
  (set (eval z) (cons z (eval (eval z))))
  (mapc '(lambda (p) (prog (n)
    (setq n (cadr p))
    (cond ((get n 'flag) (cond ((null (eval n)) (setq n (merge n dir1 dir2)))
                               (t (setq n (eval n)))))
          (getarc (car p) dir1 n dir2 (eval z)))))
    (get z dir1))
  (eval z))
```

```lisp
(defun ncovered (a Q dir1 dir2 ex done)
  (do ((A Q (cdr A)) (m (get a dir1)) (nextA (get (get a dir1) dir1)) (nextQ nil) (newd nil) (newm nil) (b) (n))
    ((null A) (cond ((or (null nextA) (null nextQ)) (return nil)))
      (setq done (append newd done))
      (do ((B nextA (cdr B)))
          ((null B) t)
        (cond ((not (ncovered (car B) nextQ dir1 dir2 ex done)) (return nil)))))
    (setq b (car A))
    (setq n (get b dir1))
    (cond ((and (not (member n newm))
                (null (assoc n done))
                (covers (eval b) (eval a))
                (or (not (covers (eval a) (eval b)))
                    (and (not (null nextA)) (null (get n dir1)))
                    (contains (get (get b dir2) ex)
                              (get (get a dir2) ex)))) (cond ((or (null (get n dir1))
                                                                   (equal m n)
                                                                   (member (cons m n) done)) (return t)))
           (setq nextQ (append (get n dir1) nextQ))
           (setq newd (cons (cons m n) newd))
           (setq newm (cons n newm))))))

(defun remove (a dir1 dir2)
  (prog (n1 n2 z)
    (setq n1 (get a dir1))
    (setq n2 (get a dir2))
    (putprop n1 (subtract (get n1 dir2) (list a)) dir2)
    (putprop n2 (subtract (get n2 dir1) (list a)) dir1)
    (setq z (intern (make_atom (catenate (get a 'back) (get a 'fore)))))
    (putprop z (subtract (get z 'arcs) (list a)) 'arcs)
    (mapc '(lambda (v) (putprop v (subtract (get v 'resolved) (list a)) 'resolved)
                        (putprop v (reverse (subtract (get v 'unresolved) (list a))) 'unresolved))
          (varsyms (eval a)))
    (cond ((equal a CURRENT) (setq CURRENT nil)))
    (setq NEWARCS (subtract NEWARCS (list a)))
    (setq ARCS (subtract ARCS (list a)))
    (set a 'bad)
    (cond ((and (includes (list n1) (get n1 dir2))
                (not (equal n1 n2))) (mapc '(lambda (z) (set z nil)) (eval n1))
           (setq NODES (subtract NODES (list n1)))
           (mapc '(lambda (b) (remove b dir1 dir2)) (get n1 dir1))))))
```

61

```
(defun prune (dir1 dir2 ex)
  (do ((X NEWARCS (cdr X)) (n) (N))
      ((null X) (do ((Y N (cdr Y)) (A))
                    ((null Y) t)
        (setq A (get (car Y) dir1))
        (do ((Z A (cdr Z)) (a) (new (intersect2 A NEWARCS)) (done (list (cons (car Y) (car Y)))))
            ((or (null Z) (null (cdr A))) 'exit)
          (setq a (car Z))
          (cond ((ncovered a (subtract A (list a)) dir1 dir2 ex done) (remove a dir1 dir2)
                 (setq A (subtract A (list a)))
                 (setq new (subtract new (list a)))
                 (cond ((null new) (return t)))))))
      (setq n (get (car X) dir2))
      (cond ((not (member n N)) (setq N (cons n N))))))
(defun nresolve (var a1 a2)
  (prog (trm b f)
    (setq trm (resolve var (eval a1) (eval a2)))
    (cond ((equal trm 'no) (return 'exit1)))
    (setq b (pairnodes (get a1 'back) (get a2 'back) 'bex 'fex 'BEND))
    (cond ((equal b 'bad) (return 'exit2)))
    (setq f (pairnodes (get a1 'fore) (get a2 'fore) 'fex 'bex 'FEND))
    (cond ((equal f 'bad) (return 'exit3))
    (cond ((not (check trm (sum (get b 'bex) (get f 'fex))) (return 'exit4)))
    (cond ((and (get b 'flag)
                (null (get b 'back))) (mapc '(lambda (p) (complete b p 'bex 'fex 'back 'BEND))
                                            (pairs (get a1 'back) (get a2 'back) (get (get a2 'back) 'back)))
                (set b 'bad)
                (cond ((null (get b 'back)) (return 'exit5)))))
    (cond ((and (get f 'flag)
                (null (get f 'fore))) (mapc '(lambda (p) (complete f p 'fex 'bex 'fore 'FEND))
                                            (pairs (get a1 'fore) (get a2 'fore) (get (get a2 'fore) 'fore)))
                (set f 'bad)
                (cond ((null (get f 'fore)) (return 'exit6)))))
    (setq NEWARCS nil)
    (cond ((and (get b 'flag) (null (eval b))) (setq b (merge b 'back 'fore))))
    (cond ((and (get f 'flag) (null (eval f))) (setq f (merge f 'fore 'back))))
    (getarc trm 'back b 'fore f)
    (prune 'back 'fore 'fex)
    (prune 'fore 'back 'bex)
    (return 'exit7)))
```

62

```lisp
(defun generate (v)
  (cond ((null (get v 'unresolved)) nil)
        (t (do ()
               ((null (get v 'unresolved)) t)
               (setq CURRENT (car (get v 'unresolved)))
               (putprop v (cdr (get v 'unresolved)) 'unresolved)
               (do ((X (get v 'resolved) (cdr X)))
                   ((null X) (putprop v (cons CURRENT (get v 'resolved)) 'resolved))
                   (cond ((member (car X) (get v 'resolved)) (nresolve v CURRENT (car X))
                          (cond ((null CURRENT) (return t)))))))))

(defun split (m n ex1 dir1 dir2 end)
  (prog (P)
    (setq P nil)
    (do ((A (get m dir1) (cdr A)) (a) (q))
        ((null A) t)
        (setq a (car A))
        (setq Q (intersect1 (cons (get n dir1) (mapcar '(lambda (v) (append (get v 'unresolved) (get v 'resolved)))
                                                       (varsyms (eval a))))))
        (setq P (append (mapcar '(lambda (b) (list a b)) Q) P)))
    (do ()
        ((null P) 'exit)
        (prog (a b nextm nextn z newm newb newr)
          (setq a (caar P))
          (setq b (cadar P))
          (setq P (cdr P))
          (setq nextm (get a dir1))
          (setq nextn (get b dir1))
          (cond ((or (null (get nextn dir1))
                     (equal nextm nextn)
                     (equal (eval a) 'bad)
                     (equal (eval b) 'bad)
                     (not (covers (eval a) (eval b)))) (return t)))
          (setq z (intern (make_atom (catenate dir1 nextm nextn))))
          (setq newm (get z 'node))
          (cond ((equal newm nextn) (return t))
                (newm (setq newb (getarc (eval b) dir1 newm dir2 n))
                      (remove b dir1 dir2)
                      (setq P (subst newb b P))
                      (return t)))
          (setq newr (sum (get nextm ex2) (get nextn ex2)))
          (cond ((like newr (get nextn ex2)) (putprop z nextn 'node)
                 (cond ((or (member nextm (eval nextn))
                            (member (list nextm nextn) WORK)) (return t))
                       (setq WORK (cons (list nextm nextn) WORK))
                       (return t)))))))
```

63

```lisp
(do ((C (get nextn dir1) (cdr C)) (c) (newC nil) (old))
    ((null C) (cond ((null newC) (remove b dir2 dir1)
                     (cond ((null (get nextn dir2)) (mapc '(lambda (d) (remove d dir1 dir2))
                                                          (get nextn dir1))))

                    (return t)))

     (setq newm (getnode ex1 (get nextn ex1) ex2 newx))
     (setq old (get newm dir1))
     (setq newb (getarc (eval b) dir1 newm dir2 n))
     (mapc '(lambda (d) (getarc (eval d) dir1 (get d dir1) dir2 newm)
            newC)

     (remove b dir1 dir2)
     (setq P (subst newb b P))
     (putprop z newm 'ncde)
     (cond ((null old) (setq WORK (cons (list end newm) (cons (list newm newm) WORK)))
                       (setq WORK (union2 (list (list nextm newm)) WORK))
                       (return t))

            ((not (or (member nextm (eval newm))
                      (member (list nextm newm) WORK))) (setq WORK (cons (list nextm newm) WORK))))
     (cond ((includes old (get newm dir1)) (return t)))
     (mapc '(lambda (q) (setq WORK (cons (list q newm) WORK))) (eval newm))
     (set newm nil))

(setq c (car C))
(cond ((and (check (eval c) (sum (get (get c dir1) ex1) newx))
            (or (null (get (get c dir1) ex1))
                (not (embraces newx (eval c)))) (setq newC (cons c newC)))))))))

(defun update (ex1 ex2 dir1 dir2 end)
  (mapc '(lambda (n) (set n nil)) NODES)
  (setq WORK (cons (list end end) (append (mapcar '(lambda (n) (list end n)) NODES)
                                          (mapcar '(lambda (n) (list n n)) NODES))))

  (do ((n)) (n))
    ((null WORK) 'exit)
    (setq n (caar WORK))
    (setq n (cadar WORK))
    (setq WORK (cdr WORK))
    (set n (cons n (eval n)))
    (split n n ex1 ex2 dir1 dir2 end)))

(defun pcgraph (V C)
  (initialize V C)
  (do ()
      ((not (member t (mapcar 'generate VARSYMS))) NODES))
    (update 'box 'fex 'back 'fore 'FEND)
    (update 'fex 'box 'fore 'back 'BEND)
  NODES)

(defun expand (N)
  (mapcar '(lambda (n) (list (cons n (cons (eval n) (plist n))) (ascii 15) (ascii 12))) N))
```

64