ANL-84-89

ANL--84-89

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

DE85 005792

**A SIGNAL FLOW MODEL FOR SEQUENTIAL LOGIC
BUILT FROM COMBINATIONAL LOGIC ELEMENTS
AND ITS IMPLEMENTATION IN PROLOG***

J. R. Gabriel and P. R. Roberts

Mathematics and Computer Science Division

September 1984

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

TABLE OF CONTENTS

Abstract.....	1
1. Introduction.....	1
2. Modeling Memory Devices	3
3. Prolog Implementation of the Signal Flow Model.....	9
References	11
Addendum: Prolog Program Containing modelfire, gus, and guo.....	13

LIST OF FIGURES

1. Circuit Layout for R-S Latch	3
2. Graph of Signal Flow Model for R-S Latch	4
3. Circuit Layout for R-S Flip-Flop Using R-S Latch	6
4. Graph of Signal Flow Model for R-S Flip-Flop	7
5. Circuit Layout for J-K Flip-Flop.....	8
6. Graph of Signal Flow Model for J-K Flip-Flop.....	8

A Signal Flow Model for Sequential Logic Built from Combinational Logic Elements and Its Implementation in Prolog

*John R. Gabriel
Philip R. Roberts*

Mathematics and Computer Science Division

Abstract

A signal flow model is described that uses memory devices from combinational logic. The implementation of this model in Prolog is also presented.

1. Introduction

Automated diagnosis of digital logic systems, as described by Gabriel [1983], uses (implicitly) a signal flow model of the system under test, and a small, special-purpose theorem prover asserting that "every component is functioning according to specification." The software directs measurements on the system; and when enough information has been accumulated to deny the assertion, the denial is printed to show which fault was found.

The automaton involves simulation of the logic, driven by a machine-readable description of the schematic diagram and a library of component behavior specifications. It performs the simulation by tracing signal flow graphs. Therefore, graphs containing loops cause non-termination of the process.

Most approaches to this problem have difficulty in modeling an R-S flip-flop built from NAND gates as primitives, unless continuous-state variables are used. The continuous ordinary differential equations describing motion of the system have a pair of stable states at large times. In a properly working R-S flip-flop, a pulse on the S line, for example, drives the system into the state at large time with $Q = 1$; a pulse on R drives it to $Q = 0$. However, if one builds an R-S from NAND gates such as were available in the early days of TTL logic, it may not always behave reliably. The reason is that if one follows the motion of the system microscopically as it crosses the boundary in state space between those states tending to $Q = 1$ and those tending to $Q = 0$ at large time, it must have sufficient "momentum" not to drift back across the boundary because of random fluctuation in the signals.

The necessary momentum is achieved in various ways. Modern 74LS TTL seems to have some small hysteresis; the various MOSFET technologies have intrinsic capacitances able to "remember" the previous state; and when vacuum tubes were still used to build R-S flip-flops, the capacitors nominally used to improve high-frequency gain around the feedback loop were also "evanescent" memory elements.

During the last few months, we tried a number of these "obvious" ways. Each proved to have flaws on close examination. Either they did not work, or they required significant symbolic algebra capability in the special-purpose prover.

Recently our attention was drawn to Petri nets as a useful modeling tool. Their "sequential" properties provide just the evanescent memory needed: the process of moving tokens to places, and then from places to transitions, seems to eliminate the difficulties encountered in other approaches. Unfortunately, net models are not quite appropriate to our needs, for the following reason. In the Petri model proper, if a single token arrives at a place, it goes to only one of the transitions able to receive tokens from the place. Thus a token models a non-sharable resource passed from place to place by decisions at transitions. In our model, on the other hand, a signal may be sent to any number of destinations, making a sharable resource.

Our model of signal transmission can be described as follows:

Gates are modeled as transitions, and places as named signals. A signal originates at one and only one transition (ignoring details such as tri-state bus drivers) and moves to a place attached to that transition. From that place it is distributed to one or more transitions taking inputs from that place, and defines signal values "known" at the inputs to the transitions. The signal is removed from the place.

When all the signals at inputs to a transition are known, the transition may become active (fire), generating an output signal (Petri token). When the transition fires, all tokens (signals) on the inputs are removed.

Each place always contains a token, and every token has a value. In our case the values are 1 and 0, representing high and low signals, respectively. Thus, all transitions are always enabled, and, in our model, all enabled transitions fire simultaneously.

The system is conceptually driven by a two-phase clock. In one phase, tokens or signals move from places to transitions; in the next phase, those transitions able to fire do so, and move their outputs to places. This alternation appears to provide just enough "evanescent memory" to allow synthesis of memory devices from combinational logic.

Some interesting properties follow from the model presented. Let us define the "state" of the system at some time as being given by the bindings between places and signals at the end of the "fire transition" phase, the signals themselves being drawn from some finite alphabet. Exactly one signal occupies a place, because the next phase will move all signals from places to transitions, and there is only one signal source connected to a place. This guarantees that the number of states in the system is finite, but possibly large.

The output from a transition depends only on its collected inputs; there is no wider context. If one of the inputs is a control signal, then the system state after that transition fires depends only on the previous state and the control signal. This makes the signal/data flow model a finite-state machine.

If the motion of the system is considered as a function of time, and the possible control signals are a finite set, then eventually, since the number of possible sets of states and control values bound to transitions is finite, some combination of control signals and state must be encountered a second time. From then on, the motion will be repeated. This shows that the system has two kinds of states at very large times: those undergoing periodic oscillation, and those where the state does not change with time. The first class we call *periodic*, the second *stable*.

2. Modeling Memory Devices

The R-S latch (Fig. 1) is a memory device with two input signals, r and s , and two output signals, q and $q0$. $q0$ is the inverted output signal and q is the noninverted output signal. That is, if q is producing a high signal, $q0$ should be producing a low signal; and if q is producing a low signal, then $q0$ should be producing a high signal. No other combination of output signals is allowed.

The r input signal is the "reset" input, and the s input signal is the "set" input. If the r input signal is brought low, the q output will be driven high. If the s input signal is brought low, the $q0$ output will be driven high. The R-S latch is at rest when both input signals are high. In this situation, the q output will rest with whatever signal it is currently producing. If both of the input signals are brought low, both of the output signals will be driven high, so this input situation is not allowed.

The R-S latch is built with two NAND gates. One NAND gate produces the q output signal from the r input signal and the $q0$ output signal. The other NAND gate produces the $q0$ output signal from the s input signal and the q output signal. The R-S latch uses the outputs as feedback to "remember."

Figure 1 - Circuit Layout for R-S Latch

In our model (hereafter referred to as the *signal flow model*), we use much of the Petri net terminology. The signal flow model of any circuit is a set of places that represent the lines of the circuit and a set of transitions that represent the devices of the circuit. Each place will contain exactly one valued token before and after every firing cycle. A firing cycle is conceptually a simultaneous firing of every transition in the model. Each transition has a list of input places, a list of output places, and a device type that describes the type of logic device the transition is emulating.

A valued token is a sharable resource. If a place is an input to several transitions, each transition gets a copy of the valued token from that place. When a transition fires, a valued token is placed on every output place of the transition. A transition may have multiple outputs, but a place may be the output of only one transition. The value of a token is determined by the device type of the transition producing it and the values of the inputs to that transition.

We use a circle to represent each place in the graph of the signal flow model and conventional drawings of devices to represent each transition. In the circle for each place we will put the name of the place.

A state in our model is a term "state" with two arguments. The first argument is the cycle number or time t ; the second argument is a list of ordered pairs, one for each place.

In the model there is a function δ which, when given a state of the model, will produce the next state. A state is said to be stable if $\delta [\text{state}(t, X)] = \text{state}(t+1, X)$; it is called periodic if $\delta^r [\text{state}(t, X)] = \text{state}(t+r, X)$ for some $r > 1$.

Figure 2 - Graph of Signal Flow Model for R-S Latch

The signal flow model for an R-S latch (Fig. 2) has four places (two input lines and two output lines) and two transitions (NAND gates). The initial state of the R-S latch model is

$\text{state}(0, [(r, 1), (s, 1), (q, 1), (q0, 0)])$.

We assume that the values of the places representing the input lines r and s are being generated from some outside source. If the inputs are held high, this state is stable, i.e.,

$\delta[\text{state}(0, [(r, 1), (s, 1), (q, 1), (q0, 0)])] = \text{state}(1, [(r, 1), (s, 1), (q, 1), (q0, 0)])$.

If the r input is brought low the following state results:

$\text{state}(1, [(r, 0), (s, 1), (q, 1), (q0, 0)])$.

where $\text{state}(1, \dots)$ is stable. If the s input is brought low the sequence

$\text{state}(0, [(r, 1), (s, 1), (q, 1), (q0, 0)])$
 $\text{state}(1, [(r, 1), (s, 0), (q, 1), (q0, 0)])$
 $\text{state}(2, [(r, 1), (s, 0), (q, 1), (q0, 1)])$

state(3,[(r,1),(s,0),(q,0),(q0,1)])

is caused by successive application of δ , and state(3,...) is stable while the inputs are maintained.

Let us note here that none of these circuits are closed systems and when we say that a model is stable, we assume that the inputs will be held constant from the input source.

Supposing that the q output signal is initially low, bringing the r input low will produce the sequence

state(0,[(r,1),(s,1),(q,0),(q0,1)])
state(1,[(r,0),(s,1),(q,0),(q0,1)])
state(2,[(r,0),(s,1),(q,1),(q0,1)])
state(3,[(r,0),(s,1),(q,1),(q0,0)])

and state(3,...) is stable. Making the s input low will result in the following stable state:

state(1,[(r,1),(s,0),(q,0),(q0,1)]).

Notice that at time $t = 2$ in two of the above examples, both places q and $q0$ are producing high signals. In our original description of the R-S latch, we said that this condition should not be allowed. However, this is a condition that does arise with devices that operate on continuous signals. In a physical R-S latch, there is a fraction of a second when, as the signals are changing, both output lines will have a high signal. However, the signals will stabilize with one output high and the other output low. The time the circuit takes to stabilize is known as the *propagation delay*. This situation raises certain issues about timing. Do the input signals have to be maintained until the outputs stabilize? If so, how long must the input signals be maintained?

The answer to the first question is yes. This is known among circuit designers as the *hold time*. The answer to the second question is not so easy. It turns out, as we shall see later, that the hold time is different for every circuit. In the R-S latch, the input signals must be held until the circuit is stable (at $t = 3$). Problems arise when one begins to build systems of these circuits and the propagation delays begin to accumulate. One solution to this problem is the use of a system clock to synchronize the operation of the circuits.

An R-S flip-flop (Fig. 3) is a memory device that exhibits properties similar to an R-S latch but uses a clock input for synchronization. It can be constructed using an R-S latch and two additional NAND gates. The inputs to the R-S flip-flop are inhibited except when the clock is pulsed. When the clock is pulsed, the r and s inputs are used to produce the output signals at q and $q0$ just as in the R-S latch. In the R-S latch, the normal resting state for the input signals is high. The normal resting state for the inputs to an R-S flip-flop is low. The reason for this can be seen by examining the circuit diagram for the R-S flip-flop. Low inputs to the R-S flip-flop will produce high inputs for the R-S latch used to build the flip-flop.

A high signal on the r input will drive the $q0$ output high, and a high signal on the s will drive the q output high. High signals on both inputs when the clock is pulsed will produce high signals on both outputs. Again this condition is not allowed.

Figure 3 - Circuit Layout for R-S Flip-Flop Using R-S Latch

The signal flow model for the R-S flip-flop has seven places and four transitions (Fig. 4). Again, we assume input signals are being provided by an external source. The initial state of the R-S flip-flop model is

$state(0, [(r,0), (s,0), (clock,0), (s1,1), (r1,1), (q,1), (q0,0)])$.

Until the clock input is turned on, $s1$, $r1$, q , and $q0$ will not change -- no matter what signals are given to r and s . When the clock is on, r and s will control the values of the tokens on $s1$, $r1$, q , and $q0$. Suppose, for example, that the clock input is high, the r input is high, and the s signal is low. The following sequence of states will result:

$state(1, [(r,1), (s,0), (clock,1), (s1,1), (r1,1), (q,1), (q0,0)])$
 $state(2, [(r,1), (s,0), (clock,1), (s1,0), (r1,1), (q,1), (q0,0)])$
 $state(3, [(r,1), (s,0), (clock,1), (s1,0), (r1,1), (q,1), (q0,1)])$
 $state(4, [(r,1), (s,0), (clock,1), (s1,0), (r1,1), (q,0), (q0,1)])$

and $state(4,...)$ is stable.

If, instead, the s input is high and the r input is low while the clock is on,

$state(1, [(r,0), (s,1), (clock,1), (s1,1), (r1,1), (q,1), (q0,0)])$
 $state(2, [(r,0), (s,1), (clock,1), (s1,1), (r1,0), (q,1), (q0,0)])$

would be the resulting sequence of states with $state(2,...)$ stable.

The timing for the signal flow model of the R-S flip-flop is such that the device will stabilize on cycle $t = 4$. The clock can be turned off before the firing that produces $state(4,...)$ but only after the firing that produces $state(2,...)$. For example, going from

$state(2, [(r,1), (s,0), (clock,1), (s1,0), (r1,1), (q,1), (q0,0)])$

in the first example with no input for the clock,

`state(3,[(r,1),(s,0),(clock,0),(s1,0),(r1,1),(q,1),(q0,1)])`

and

`state(4,[(r,1),(s,0),(clock,0),(s1,1),(r1,1),(q,0),(q0,1)])`

would result. In the model for the R-S flip-flop, the minimum hold time would be two cycles; i.e., the inputs can be turned off after two firing cycles. The firing cycle that produced state(1,...) is essential, and the firing cycle that produced state(2,...) is essential.

Figure 4 - Graph of Signal Flow Model for R-S Flip-Flop

There is still a problem with the R-S flip-flop. If both the r and the s input signals are high, both of the outputs will be driven high -- the condition we do not allow. A memory device called the J-K flip-flop (Fig. 5) handles the high input on both inputs in a different fashion. If both inputs are high, the device "toggles." That is, the output signals are interchanged. If the q output is high and the $q0$ output is low, the q output will go low and the $q0$ output will become high (and vice versa). This is a desirable situation allowing integrated circuit designers the freedom to design with memory devices that do not have any indeterminate states (states where both outputs are high).

Figure 5 - Circuit Layout for J-K Flip-Flop

The model (Fig. 6) for the J-K flip-flop exhibits some interesting characteristics for the case when both inputs are high. The inputs must be held for a certain amount of time to produce the toggle, but they must not be held too long.

Figure 6 - Graph of Signal Flow Model for J-K Flip-Flop

Beginning with

```
state(0,[(r,0),(s,0),(s1,1),(r1,1),(q,1),(q0,0),(j,1),(k,1),(clock,1)]),
```

and leaving the *j*, *k*, and *clock* inputs high, the following sequence of states is produced:

```
state(1,[(r,1),(s,0),(s1,1),(r1,1),(q,1),(q0,0),(j,1),(k,1),(clock,1)])
state(2,[(r,1),(s,0),(s1,0),(r1,1),(q,1),(q0,0),(j,1),(k,1),(clock,1)])
state(3,[(r,1),(s,0),(s1,0),(r1,1),(q,1),(q0,1),(j,1),(k,1),(clock,1)])
state(4,[(r,1),(s,1),(s1,0),(r1,1),(q,0),(q0,1),(j,1),(k,1),(clock,1)])
state(5,[(r,0),(s,1),(s1,0),(r1,0),(q,0),(q0,1),(j,1),(k,1),(clock,1)])
state(6,[(r,0),(s,1),(s1,1),(r1,0),(q,1),(q0,1),(j,1),(k,1),(clock,1)])
state(7,[(r,1),(s,1),(s1,1),(r1,0),(q,1),(q0,0),(j,1),(k,1),(clock,1)])
state(8,[(r,1),(s,0),(s1,1),(r1,0),(q,1),(q0,0),(j,1),(k,1),(clock,1)])
state(9,[(r,1),(s,0),(s1,0),(r1,1),(q,1),(q0,1),(j,1),(k,1),(clock,1)])
```

and the system has completed one oscillation (state 3 and state 9 are the same). The outputs will now toggle at $t=4, 7, 10$, etc. To produce the first toggle, the inputs must remain high until after the $t=2$ firing but must be turned off before the $t=5$ firing. Turning the clock off will again sufficiently disable any input from the *j* or *k* lines. If the clock is turned off after the $t=1$ firing, the outputs will not toggle. If the clock is turned off after the $t=2$ firing, the outputs will alternate together between high and low signals, a bad situation. If the clock is turned off after the $t=5$ or the $t=6$ firing, the outputs will toggle back to their original states; but if it is turned off after the $t=7$ firing, the situation with both outputs alternating together between high and low will occur. So, the designer must be careful to issue clock signals at the correct time and for the correct duration and also warn any users about undesirable behavior.

3. Prolog Implementation of the Signal Flow Model

Prolog may be thought of as a database of facts, a means of asking whether some fact is in the database (a "simple question"), and a means to define rules that express general questions in terms of simple questions (and other rules). A question may contain "variables." If a fact making the question "true" is in the database, then the question is answered true, with the variables returned set to the values from the database that makes the answer true. A question or rule answered "true" is said to "succeed."

Thus, if the database contains the facts

```
doctor(joe).
doctor(bob).
```

then the question

```
? doctor(X).
```

may succeed twice in a search of the database, once with $X=joe$, and once with $X=bob$.

Adding facts about residence to the database, we might say

```
local(bob).
```

If we now want to find individuals who are doctors and local, we write the rule

```
localdoctor(X) :-  
    doctor(X),  
    local(X).
```

which in English is

X is a local doctor, if X is local and X is a doctor.

The question

```
? localdoctor(X).
```

will cause a search of the database. When `doctor(joe)` is found true, a second search will be made for `local(joe)`. This will fail. There is no fact "`local(joe)`". The Prolog processor will then retry to find another instance of `doctor(X)`, starting at `doctor(joe)` where it branched to test `local(joe)` previously. It will find `doctor(bob)`, proceed once more to search for `local(bob)`, and succeed; and so the rule `localdoctor(X)` will succeed, instantiating (i.e., setting) X to "bob".

With this much background (see Clocksin and Mellish [1981] for a more detailed discussion), we will now describe the signal flow model simulator.

A device is defined by a set of predicates that name the device and describe its input and output behavior. For example, a NAND gate would be described by

```
device(nand,[1,1],[0]).  
device(nand,[_,_],[1]).
```

If both inputs to a NAND gate are 1, the output from the NAND gate is 0. Any other input (denoted in Prolog by the "_" for "match any") will produce an output of 1. Remember, we use 1 for a high signal and 0 for a low signal. A three-input AND gate would be described by

```
device(and3,[1,1,1],[1]).  
device(and3,[_,_,_],[0]).
```

Each device type must have a description. Any type of device can be described as long as its performance can be described solely in terms of input signals and output signals. The general form for the device description is

```
device(dev-type,input-list,output-list).
```

where *dev-type* is the name of the type of the device. This name is important because it will be used in the circuit specification to identify a particular device in the system. The *input-list* is a Prolog list specifying the input signals. The ordering may be important because the circuit specifications will name the place (line) providing the input signal. The *output-list* is also a Prolog list. It designates the output that corresponds to the inputs. For most devices, only one output is specified, but some may specify multiple outputs.

A circuit is described by using three types of clauses. The *dev* clauses define the particular devices in the circuit by providing a device number, a device type, a list of input places (lines), and a list of output places. The *dev_list*

clause gives a list of devices in the circuit (this is useful if you want to remove particular devices and determine the consequences of their removal). The *state* clause has been used extensively throughout this paper. The initial state of every device must be included.

A sample circuit description is the description of the R-S flip-flop:

```
/* R-S FF */
dev(1,nand,[r,clock],[s1]).
dev(2,nand,[clock,s],[r1]).
dev(3,nand,[r1,q0],[q]).
dev(4,nand,[s1,q],[q0]).
dev(5,high,[],[clock]).
dev(6,high,[],[r]).
dev(7,low,[],[s]).
dev_list([1,2,3,4,5,6,7]).
state(0,[(r,1), (clock,1), (s,1), (s1,1), (r1,1), (q0,1), (q,0)]).
/* end of R-S FF */
```

The high and low devices are used to produce signals, presumably from an outside source. Any time during the execution of the model, devices can be changed by asserting new facts about them. Note that you would only want to do this with devices that were used to produce signals from the outside.

The core of the Prolog implementation of the model is the predicate *modelfire*. This predicate fires all the devices in the device list and produces a new state from the outputs of those devices. The predicate *gus* will produce new states until a stable state is reached. If the system never reaches a stable state, the predicate terminates because of lack of storage space on the machine. The predicate *guo* will generate new states until an old state is regenerated, at which time the system will begin oscillating and generation is stopped.

Note that a new state is generated solely by the outputs of the transitions. All tokens are absorbed at a given firing, and the only tokens that now describe the new state are the tokens that are produced by firing the devices.

References

Clocksin, W. F., and Mellish, C. S. 1981. *Programming in Prolog*. Springer-Verlag, New York.

Gabriel, J. R. 1983. *Algorithms for Automated Diagnosis of Faults in Physical Plants*. Argonne National Laboratory Report ANL-83-70.

Addendum: Prolog Program Containing modelfire, gus, and guo

```
/* device descriptions */
device(nand,[1,1],[0]).
device(nand,[_], [1]).

device(nor,[0,0],[1]).
device(nor,[_], [0]).

device(inverter,[1],[0]).
device(inverter,[0],[1]).

device(and,[1,1],[1]).
device(and,[_], [0]).

device(and3,[1,1,1],[1]).
device(and3,[_], [0]).

device(or,[0,0],[0]).
device(or,[_], [1]).

device(low,[],[0]).
device(high,[],[1]).

/*
 * gus generates new states until a stable state is reached
 */
gus :-
    stable.
gus :-
    modelfire,
    gus.

/*
 * guo produces new states until the system finds a previous state
 */
guo :-
    oscillating.
guo :-
    modelfire,
    guo.

/*
 * stable determines if f(u) = u
 */
stable :-
    state(T,S), !,
    T2 is T-1,
    state(T2,S).
```

```
/*
 * oscillating determines if f**r(u) = u
 */
oscillating :-
    state(T,S),
    state(T2,S),
    T == T2.

/*
 * modelfire finds the next state of the model
 */
modelfire :-
    dev_list(Devices),
    fire_all_devices(Devices,Outputs),
    create_new_state(Outputs).

/*
 * fire_all_devices fires all the transitions for the network collecting
 * the outputs for the new state
 */
fire_all_devices([],[]).
fire_all_devices([Device|Devices],Outputs) :-
    fire_dev(Device,Outputs1),
    fire_all_devices(Devices,Outputs2),
    merge(Outputs1,Outputs2,Outputs).

/*
 * commands issued to it
 */
create_new_state(Outputs) :-
    state(T,S),
    T2 is T+1,
    asserta((state(T2,Outputs))).

/*
 * fire_device fires Device_Number producing the new outputs
 */
fire_dev(Device_Number,Outputs) :-
    dev(Device_Number,Type,Inputs,Output_Places),
    state(T,State), !,
    get_inputs(Inputs,State,Input_Values),
    device(Type,Input_Values,Output_Values),
    make_outputs(Output_Places,Output_Values,Outputs).
fire_dev(,_).

/*
 * get_inputs from state returning a list of the values of the inputs
 */
get_inputs([],_).
get_inputs([Input|Inputs],State,[Ival|Ivals]) :-
    search_for(Input,State,Ival),
    get_inputs(Inputs,State,Ivals).
```

```
/*
 * search_for finds the input value for an input in state
 */
search_for(Input,[ ],0).
search_for(Input,[(Input,Ival)|_],Ival).
search_for(Input,[X|Places],Ival) :-
    search_for(Input,Places,Ival).

/*
 * make_outputs produces the output values for the new state
 */
make_outputs([ ],[ ],[ ]).
make_outputs([Output|Outputs],[Output_Val|Output_Vals],
    [(Output,Output_Val)|Rest_of_Output]) :-
    make_outputs(Outputs,Output_Vals,Rest_of_Output).

/*
 * merge list of places with outputs
 */
merge([X|Xs],Ys,[X|Zs]) :- merge(Ys,Xs,Zs).
merge(Xs,[Y|Ys],[Y|Zs]) :- merge(Xs,Ys,Zs).
merge(Xs,[ ],Xs).
merge([ ],Ys,Ys).
```


Distribution for ANL-84-89

Internal:

J. R. Gabriel (10)
K. L. Kliewer
A. B. Krisciunas
P. C. Messina
D. M. Pahis
T. M. Woods (2)
G. W. Pieper

ANL Contract File
ANL Patent Dept.
ANL Libraries
TIS Files (6)

External:

DOE-TIC, for distribution per UC-32 (170)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. Bona, U. of Chicago
T. L. Brown, University of Illinois, Urbana
J. C. Browne, U. of Texas, Austin
S. Gerhart, Wang Institute, Tynsboro, Mass.
G. Golub, Stanford University
W. C. Lynch, Xerox Corp., Palo Alto
M. F. Wheeler, Rice U.
D. Austin, BES-DOE
G. Michael, LLL
C. E. Oliver, BES-DOE
P. R. Roberts, University of Illinois, Urbana (10)