# Qualitative Simulation of Swine Production

by

Matthew T. Mason

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
Massachusetts Institute of Technology
May, 1976

Signature of Author ....Matthew T. Mason....
Department of Electrical Engineering and Computer Science, May 7, 1976

Certified by ....Gerald Jay Sussman....
Thesis Supervisor

Accepted by ..........................................................
Chairman, Departmental Committee on Theses

# Qualitative Simulation of Swine Production

by

Matthew T. Mason

## Abstract

This thesis investigates techniques for simulating and analyzing models expressed as diagrams. The programming language used (ARSE) represents a departure from conventional simulation languages by providing powerful analytical tools, such as symbolic expression manipulation and explanation of results. Methods for defining domains and models are presented, and demonstrated on the domain of hog farms. The techniques demonstrated may be useful in a variety of modeling and simulation applications.

Thesis Supervisor: Gerald J. Sussman

Title: Assistant Professor of Electrical Engineering and Computer Science

## Table of Contents



## List of Figures

## *I Introduction*

A simulation of a particular system may be undertaken for either of two reasons; first, to attempt to predict the actual behavior of the system, and second, to investigate the structure of the system. In the first case, the output of the model given a particular input is the only item of interest. An example of this type of simulation is a simulation of a football game to determine a profitable bet.

More frequently, the input-output behavior is not the only interesting characteristic of the model. When one is dealing with poorly understood systems, the purpose of simulation may be to develop a better model of the system. In this case, the actual behavior of the model is not enough -- the investigator needs to know <u>why</u> the particular model does or doesn't work. Of course, in simple cases, the designer can infer the cause of failure from the observed input-output behavior of the model. However, this task can become insurmountable for more complex models.

The purpose of my thesis is to investigate a language (ARSE <Sussman 75>) which offers several advantages over conventional simulation languages. ARSE is designed for use with an interactive time-sharing environment. The user monitors the simulation while it is in progress and may interrupt at any time to investigate the reasons for the model's behavior and possibly to change the model or the input. If the user does make changes, ARSE saves all the work which isn't affected by the changes. Since ARSE remembers the reasons for every datum produced, it is able to explain the behavior of the model in terms of ARSE's reasoning processes. The user may use symbolic variables as input values, in which case ARSE will express the output as a function of the input variables.

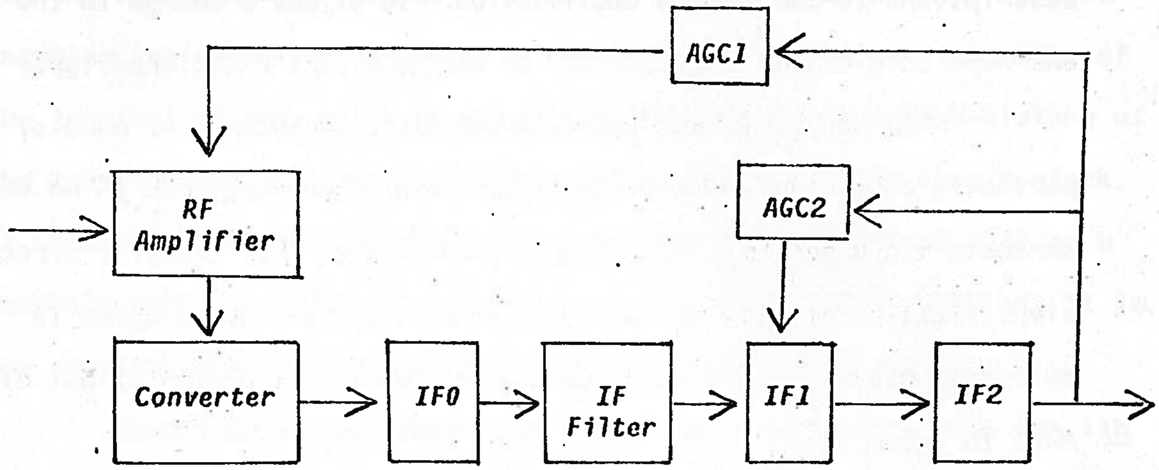
## *II Models and Simulation*

### II.A Model Descriptions

It is common, when designing a simulation model, to begin with a diagram of the system being modeled, where boxes in the diagram represent parts of the model, and lines represent paths of flow of material or information between the parts. The usefulness of a diagrammatic description is attested to by the fact that it is the preferred medium for humans. It is a concise and transparent means of describing the model. Diagram descriptions are declarative in nature. For example, in the world of electronic circuits, a symbol which represents a resistor is interpreted as saying, "this part is a resistor." A line means "this part is wired to this other part."

In order to discuss diagrams easily, I'll define some terms. By *part*, I mean a box of the diagram. Usually each part is an instantiation of a *part-type*, an entity whose behavior is well-known. As an example, "Q101" could be a *part* in a circuit diagram, with corresponding *part-type* "bipolar-junction-transistor". Lines in the diagram will sometimes be referred to as *connections*. A line is connected to a part at a *terminal* of the part. Hence a diagram of an assembly line would have a line from the "out" terminal of one assembly station to the "in" terminal of the assembly station which follows it. I'll refer to parts and lines collectively as the *elements* of the diagram. See figure 1 for an example of these conventions.

For the purposes of simulation, a diagram description must be transformed into a computational description. Most languages require that we encode the flow of material or information as <u>events</u> and, for each part




| Part | Part-type |
|---|---|
| RF Amplifier | Amplifier |
| IF0 | Amplifier |
| IF1 | Amplifier |
| IF2 | Amplifier |
| Converter | Converter |
| AGC1 | Automatic-gain-control |
| AGC2 | Automatic-gain-control |
| IF filter | Filter |

Figure 1 -- Diagram Example

in the diagram, write code to interpret and generate events. The declarative elements of the diagram description are transformed into imperative constructs, blocks of code to carry out the appropriate operations. This transformation does not preserve the full power of the diagram description. There are a multitude of operations which are fairly natural on a diagram, but which are extremely awkward with a computational description. For instance, with most computational descriptions, simulation time is monotone increasing. There is no similar constraint on diagrams, making it just as easy to find causes given effects as it is to find effects given causes. An additional advantage of diagrammatic descriptions is the ease of modification. To effect a change in the model, one need only change a connection, or change a part's declared type.

ARSE does, in fact, use diagrammatic descriptions of models, and is perfectly comfortable with reverse inference. However, ARSE in no way exhausts the benefits of the diagram description. For example, direct simplification of the diagram may be possible, a technique which is extremely useful in the domain of electronic circuit analysis, but of which ARSE is ignorant.

Obviously, if the model is to be completely specified, the part-types used in the diagram must be well defined. Knowing that a particular part is a resistor is not useful unless we know exactly what a resistor is. Hence, an important part of model construction with ARSE is the definition of part-types. Usually the part-type definitions are declarative in nature, too. Most definitions simply specify a mathematical relation between variables defined at the terminals of the part-type.

## II.B The Representation of Time

The greatest difficulty in writing the hog-farm simulation was the representation of time. In fact, ARSE's ability to pass symbolic information was compromised somewhat in this area. In the original application, electronic circuit analysis, the explicit treatment of time is not necessary.

ARSE attempts to find a value, numerical or symbolic, for each terminal in the model description diagram. The assumption is that each terminal has one correct value. This is at odds with the usual interpretation of diagrams, where the value of a terminal varies with time.

Conventional languages handle this easily, using a- simulation clock and changing the values of terminals as required. If we collect the value of the terminal at each point in simulation time, we obtain a time-history of the terminal, which I call a <u>schedule</u>. Since ARSE does not have a clock, it works directly with schedules, i.e. the value of a terminal will be a complete schedule. This is certainly not the only way to do it, but it is the easiest to work with that I could come up with.

Schedules are represented as one-dimensional arrays, with the i<u>th</u> entry representing the value at the i<u>th</u> time interval. An entire schedule may be represented by a symbolic expression, but an individual entry may not. Certainly this compromises the symbolic manipulation in general but it doesn't seem to hamper the effectiveness of the simulation.

The effectiveness of ARSE's ability to handle symbolic expressions depends upon the operators defined. Symbolic expressions are completely unwieldy unless ARSE is able to simplify them. Consider the effect if no simplification is performed. Each part in the chain between the input and output in question will be given a symbolic input value, and will express its output as a function of its input. As the symbolic expression propagates through the chain, each part will add at least one level of functional composition. The end result will be an expression which is incomprehensible.

Hence, it is important that the set of mathematical operations used be amenable to simplification. It is also important that each operator have an inverse, so that ARSE can perform reverse inference. Unfortunately, these goals were only partially satisfied, restricting somewhat ARSE's effectiveness. This is only partly due to the design. The use of schedules requires the use of some operations which slightly inhibit simplification, and which, more importantly, do not have inverses. For example, the operator which converts weekly schedules to monthly schedules does not have an inverse.

Feedback loops are another source of difficulty. With conventional languages, feedback loops cause no difficulty because of the use of the simulation time clock. The use of schedules in ARSE, however, leads to an impasse when feedback loops arise. Consider the feedback loop in figure 2. This diagram has two parts, defined by functions *f1, f2* and *f3*. Part *A* has three terminals, *A1, A2* and *A3*. Part B has three terminals labeled *B1, B2* and *B3*. *IN* and *OUT* are external terminals. *IN* has acquired a value, a schedule which I have abbreviated by *SC3*. Whether *SC3* is an actual numerical schedule or a symbolic expression doesn't matter. ARSE will propagate *SC3* as the value for terminal *A1*, but then cannot proceed immediately. It needs values for both *A1* and *A2* in order to find *A3*. Likewise, it needs a value for *B1* before it can find *B2* or *B3*. The solution is to give one of the terminals in the feedback loop a symbolic value, as in figure 2.2. *B1* has been given a symbolic value, x. Now ARSE deduces that the value of *B2* is *f3(x)*. This value is propagated as the value of *A2*. Now we find that the value of *A3* is *f1(SC3, f3(x))*. Now it is possible to solve for *x*, as long as the functions *f1* and *f3* are well-behaved.

There are other approaches to the feedback problem. It is possible to embed a closed-form solution in the definition of a part-type which contains the entire feedback loop, as in figure 2.4. ARSE also provides a mechanism (see section V) for making assumptions and backtracking which can be used to solve feedback loops.

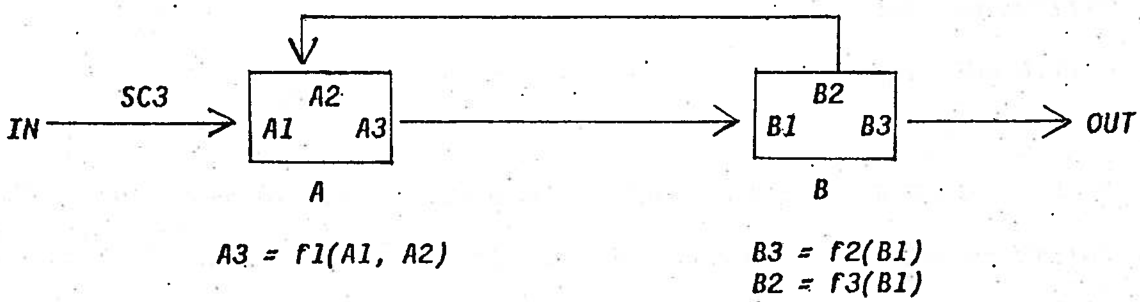



Figure 2.1

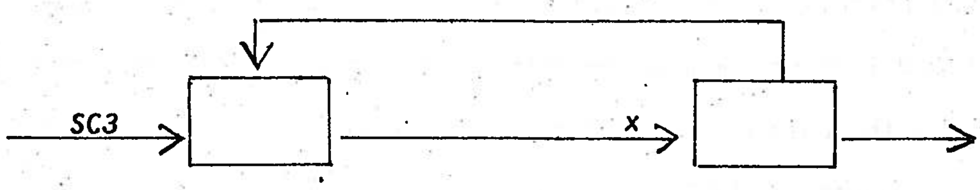



Figure 2.2

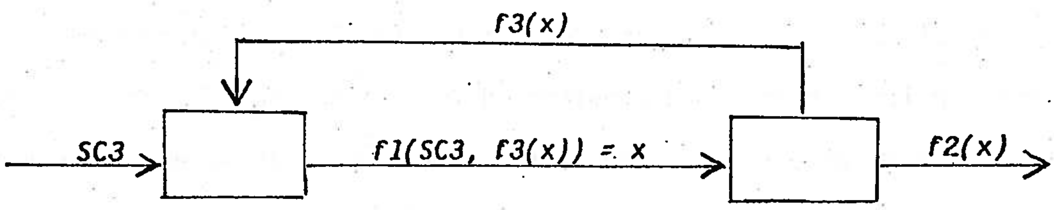



Figure 2.3

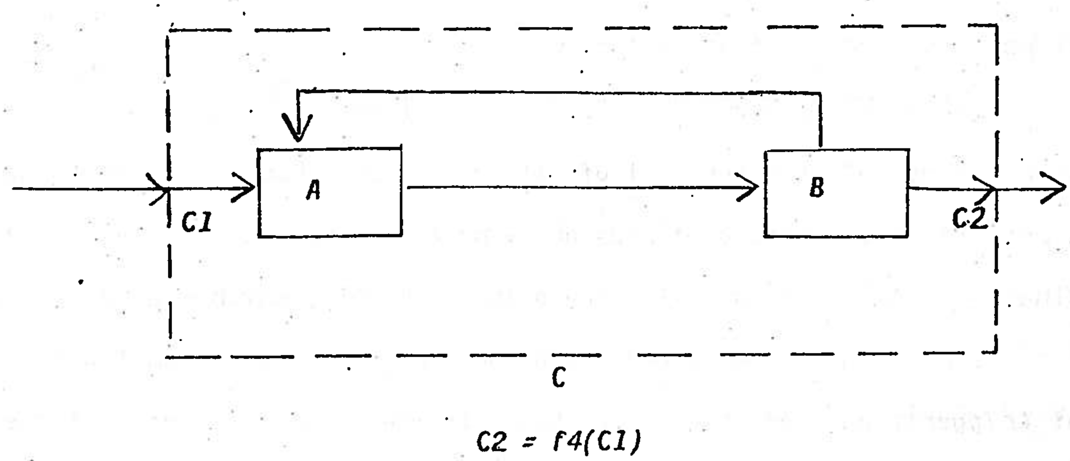



Figure 2.4

Figure 2 -- Feedback Loop

## *III Using ARSE*

### III.A ARSE Introduction

ARSE is a system developed by Gerald J. Sussman and Richard M. Stallman to support an electonic circuit analysis program. It is written in MACLISP. <Moon 74> ARSE consists mainly of a pattern-matching data base, a priority queue control structure, and some stuff for defining part-types and loading model descriptions. Every result that ARSE obtains is put into the data base as a ***fact*** of the form

(<pattern> FACT <fact-number>)

Whenever such an item is obtained, ARSE also prints it at the user's terminal to keep him informed of the simulation's progress. As in diagram descriptions, models are constructed from ***parts***, each of which is an instance of some pre-defined ***part-type***. Most facts assert a value for some terminal of a particular part. For example, if we have defined a part-type called ***pen***, and have specified that a part of our model called ***sow-pen*** is of part-type ***pen***, a typical fact would be

((= (HOGS-IN SOW-PEN) #SCHEDULE-263) FACT F201)

where "hogs-in" is a terminal of all ***pens***. #SCHEDULE-263 is an array which represents the schedule of hogs arriving at the sow-pen. Unless instructed otherwise, ARSE will not list the actual schedule, which may be lengthy.

Each part-type is defined by one or more ***laws***. Each law has a set of ***triggers***, each of which corresponds to one of the terminals of the part-type. When a fact asserts a value for a terminal specified by one of the triggers, the law is triggered and may assert a value for any other terminals of the part in question. In order to explain how laws are written, it is first necessary to describe the pattern-matching.

Two patterns match if and only if they look exactly the same, with the following exceptions. Any variable preceded by "!," is to be evaluated before matching is attempted. Hence "(foo bar)" matches "(foo !,x)" if and only if the value of x is *bar*. A variable preceded by "!>" may match anything, and the variable is bound to whatever it matches. So "(foo bar)" will always match "(foo !>x)", with the side effect of binding x to *bar*. The prefix "!?" means the same thing as "!,", sort of. Its other properties are unimportant in this context.

The form of a law is

```
(law <law-name> ( -variables- )
     ( -triggers- )
     <body> )
```

As an example of a law, consider *hog-sell*:

```
(law hog-sell ((mrkt* market) hogs price income cwt)
     ((= (price/cwt !?mrkt*) !>price)                      ;triggers
      (= (average-cwt !?mrkt*) !>cwt)
      (= (hogs-in !?mrkt*) !>hogs)
      (= (income !?mrkt*) !>income))
     (equation 'income                                      ;body
               '(sn&* hogs (&* price cwt)) ))
```

In this case, the terminals are *price/cwt*, meaning dollars per hundred pounds of hog, *average-cwt*, meaning average weight of hogs, *hogs-in*, and *income*.

The following remarks should make hog-sell fairly clear. The convention for variable declarations is that the first variable in the list, in this case *mrkt**, will be bound to the part's name when the law is triggered. The declaration of *mrkt** also specifies that it can only match names of *markets*. Note that each trigger, if successfully matched, causes a variable to be bound to the value of a terminal. The function *sn&** represents multiplication of an array by a scalar. The function *&** represents multiplication of two scalars (see appendix 1 for a complete list of the arithmetic operators.) So, the body of hog-sell indicates that *income* is equal to the product of *hogs*, *price*, and *cwt*. The function *equation* enforces this relation in the following way:

(1) if all but one of the terminals have values, equation will solve for the unknown and assert its value in the data base.

(2) if more than one of the terminals is unknown, equation will simply return. Note that the law will be triggered again if one of the unknown terminals becomes known.

(3) if all terminals have values, equation will make sure that the equality relation is true. If it isn't, equation will cause a *contradiction* (contradictions are discussed in section V). If one or more of the terminal values is <u>symbolic</u>, equation will attempt to find values for the symbolic variables used.

## III.B A Simple Example

Let's suppose that a simulation of a hog farm is in progress, and that ARSE has reported deducing the following facts:

```
((= (HOGS-OUT BOAR-PEN) #SCHEDULE-263) FACT F200)
((= (HOGS-IN BOAR-MARKET) #SCHEDULE-263) FACT F201)
((= (INCOME BOAR-MARKET) #SCHEDULE-66) FACT F202)
```

In this case, *boar-pen* and *boar-market* are two of the *parts* of the model being simulated, each of part-type *market*. The *values* are again one-dimensional arrays in this case, abbreviated for readability. So fact F202 means that the income generated at the boar-market is given in #schedule-66. If we want to know more about fact F202 the appropriate question (with ARSE's response) is:

```
(what '(income boar-market))

((13 760.0) (26 760.0) (39 760.0) (52 760.0) (65 760.0)) (F202)
```

This means that $760 was generated in each of months 13, 26, 39, etc. If we want to know how this fact was determined, we can query ARSE:

```
(why 'f202)

ANTECEDENTS OF F202 (= (INCOME BOAR-MARKET) #SCHEDULE-66)
F67 (= (PRICE/CWT BOAR-MARKET) 38.0)
F68 (= (AVERAGE-CWT BOAR-MARKET) 4.0)
F201 (= (HOGS-IN BOAR-MARKET) #SCHEDULE-263)
F66 (IS-A BOAR-MARKET MARKET)
D19 HOG-SELL
OK?
```

ARSE responds with a list of the facts needed to deduce F202, along with the name of the *law* used to make the deduction, in this case the law named "hog-sell", which we examined in section III.A. The body of hog-sell consisted of an equation of four variables. The facts F67, F68, and F201 provided values for three of those variables, and the fourth was asserted by calling the function *equation*. Since *hog-sell* only applies to *markets*, the deduction also depends upon fact F66, which asserts that *boar-market* is a *market*.

### III.C "Wiring Up" a Model

Once each possible part-type has been defined, models may be constructed from them. There are two basic operations involved. First, each part must be declared, specifying its name and part-type. Second, the connections between the parts of the model must be specified. A list of the desired connections is passed to the function *connect*. It asserts the connections in the data base in a form that a special law called *propagate* recognizes. In the previous example, "(hogs-out boar-pen)" was connected to "(hogs-in boar-market)" in this fashion. The effect of this connection may be examined by typing

```
(why 'f201)

ANTECEDENTS OF F201 (= (HOGS-IN BOAR-MARKET) #SCHEDULE-263)
F112 (OUTPUT (HOGS-IN BOAR-MARKET) N109)
F200 (= (HOGS-OUT BOAR-PEN) #SCHEDULE-263)
F111 (INPUT (HOGS-OUT BOAR-PEN) N109)
D29 PROPAGATE
OK?
```

Apparently, the connection was effected by creating a node, ***N109***, and making "(hogs-out boar-pen)" an input to the node and "(hogs-in boar-market)" an output of the node. The law ***propagate*** is triggered whenever an input or output value is determined and asserts that the sum of the inputs is equal to the sum of the outputs. In simple cases, with only one input and one output, this has the effect of identifying the two terminals with each other. The process of constructing a model out of predefined part-types is referred to as "wiring up" because of the obvious analogy to wiring an electronic circuit.

Usually, a model is wired up and the simulation begun by calling *plan*. An example of the use of *plan* is given in figure 3 and figure 4. In figure 3 the model is named *toyfarm* and a 3-year simulation is specified. The parts are declared and some of their terminals are given initial values. The first part declared is *br-pen*, which is an instance of part-type *breeding-pen*. This line will cause the facts F42 and F43 to be asserted.

```
((= (DELAY BR-PEN) 3.0) FACT F42)
((= (FEED-RATE BR-PEN) 31.5) FACT F43)
```

The declarations of the rest of the parts are handled similarly. Figure 4 lists the connections required to wire up the model. Figure 5 represents the same model in graphical form.

```
(plan toyfarm 3                          ; analyze first 3 years

   (parts (breeding-pen br-pen     (delay 3.0)   (feed-rate 31.5))
          (farrowing-house fw-house              (feed-rate 91.0))

          (pen gestation-pen       (delay 13.0)  (feed-rate 31.5))
          (pen non-breeder-pen     (delay 3.0)   (feed-rate 31.5))
          (pen finish-house        (delay 6.0)   (feed-rate 64.8))

          (market finish-market         (price-cwt 52.40) (average-cwt 2.2))
          (market non-breeder-market    (price-cwt 45.00) (average-cwt 3.0))

          (source sow-source   (price 100.0)))

   (connect  . . .

          ))
```

Figure 3 -- Sample Part Declarations

## *IV Hilltop*

### IV.A Notes on Confinement Hog Production

Before presenting a full-fledged model, it seems best to outline the basic operation of confinement hog farming, with particular attention to Hilltop farm. Confinement hog production is a method whereby all hogs, except the breeding herd, are kept inside specially designed houses. The breeding herd and boars are kept in outdoor pens. The breeding herd is divided into groups, typically three in number. Breeding is accomplished by placing a group of sows in the breeding pen with the boars for three weeks. Sows which do not conceive will be sold at market. It takes a few weeks to tell whether or not a sow has conceived.

Sows which conceive successfully are placed in the gestation pen

```
(plan toyfarm 3                              ; analyze first 3 years

   (parts   . . . )

   (connect (((hogs-conceived-out br-pen))              ; node1
             ((hogs-in gestation-pen)))

            (((non-breeders-out br-pen))                ; node2
             ((hogs-in non-breeder-pen)))

            (((hogs-out gestation-pen))                 ; node3
             ((hogs-in fw-house)))

            (((hogs-out non-breeder-pen))               ; node4
             ((hogs-in non-breeder-market)))

            (((hogs-out fw-house)                       ; node5
              (hogs-out sow-spurce))
             ((hogs-in br-pen)))

            (((pigs-out fw-house))                      ; node8
             ((hogs-in finish-house)))

            (((hogs-out finish-house))                  ; node10
             ((hogs-in finish-market)))

            (((feed-in br-pen)                          ; node11
              (feed-in gestation-pen)
              (feed-in fw-house)
              (feed-in finish-house)
              (feed-in non-breeder-pen))
             ((feed-out mill1)))

            (((feed-cost mill1)                         ; node12
              (cost sow-source))
             ((expense bank1)))

            (((income finish-market)                    ; node13
              (income non-breeder-market))
             ((income bank1)))))
```

Figure 4 -- Sample Connections

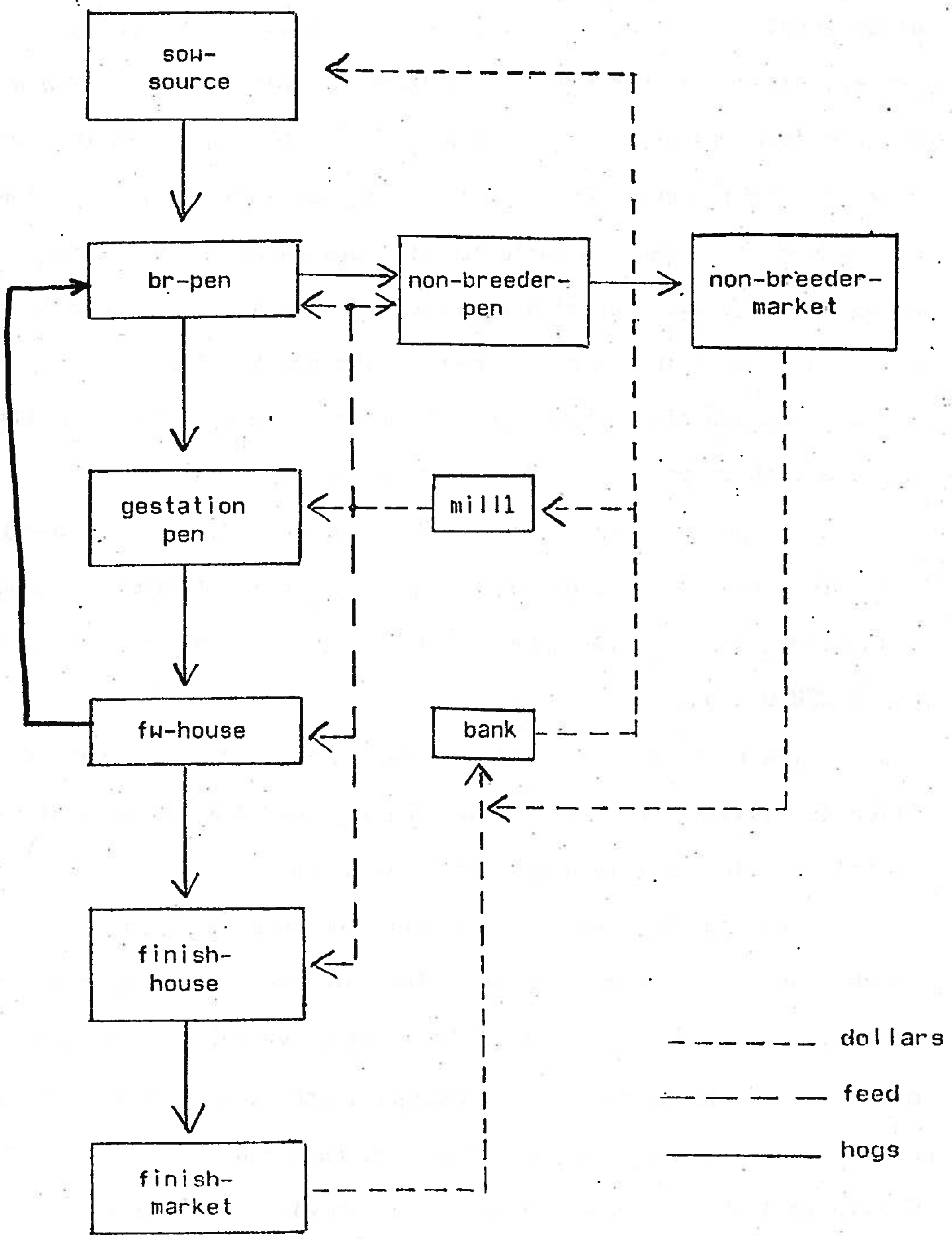



Figure 5 -- Graph of Toyfarm

for a period of thirteen weeks. Then they are placed in the farrowing house. To farrow means to give birth to pigs. Farrowing usually takes place about four weeks after the sows arrive in the farrowing house. An average litter can range from 8 to 10 pigs. The sows nurse the pigs for another four weeks, bringing the sows' total time spent in the farrowing house to eight weeks. From the farrowing house they return to the breeding pen. Hence the complete cycle for the sows takes about 24 weeks. The groups of sows are bred at staggered intervals of eight weeks to ensure efficient use of the farrowing house. The number of sows in a group is dictated by the size of the farrowing house. In the case of Hilltop Farm, there are three groups of 30 sows each.

The pigs are sent from the farrowing house to the grower house, where they remain for eight weeks, and then to the finishing house, where they remain another eleven weeks. They are then sent to market, weighing about 220 pounds.

The boars are kept for one year and replaced from outside the farm. After two litters the sows in the breeding herd are replaced with gilts (female virgin pigs) from the finishing house.

The hogs are fed mostly on milo and corn. Hilltop has a feed mill which mixes and grinds the grain. The main sources of expense are capitalization, feed, livestock replacement, veterinary expenses, equipment maintenance, market and hauling expenses, and labor. Income is generated by sale of slaughter hogs, non-breeders, replaced boars, and replaced sows. Capitalization expenses were never incorporated into the model.

## IV.B The Hilltop Model

The most common data type used in the model is the ***schedule***. Schedules are coded as one-dimensional lisp arrays. All schedules are weekly except for dollar schedules, e.g. "(balance bank1)", which are monthly. Exceptions to this rule will have names clearly indicating so, e.g. "(mnthly-pigs-present hilltop)".

Part-types are defined for pens, markets, sources (where gilts and boars are purchased), mills, veterinarians, banks, and laborers. In addition, there are some laws written for part-type ***hog-farm***, which requires a little explanation. Actually, when ***plan*** is called to wire up the model and start the simulation, the first thing it does is assert

((IS-A HILLTOP HOG-FARM) FACT F30)

So any laws written for part-type *hog-farm* will be run on Hilltop. These laws were written for obtaining global information, like the total number of hogs present at any given time.

Appendix 2 lists the laws used to define all the part-types. Appendix 3 is a listing of the wiring of Hilltop.

## IV.C Part-types

Appendix 2 is a listing of the part-type definitions used in the simulation of Hilltop Farm. Although these definitions are pretty clear, a brief explanation of the part-types is given below.

*Pen* is the most common part-type in the simulation of Hilltop Farm. The law *pen-delay* says that the input schedule is delayed appropriately to obtain the output schedule. *Pen-present* computes the number of hogs present in the pen, and *pen-feed* computes the feed required.

*Breeding-pen* is the same as pen, except that it uses the conception-rate to determine how many sows bred successfully.

*Farrowing-house* uses a global variable, farrow-time, to determine how long the sows are delayed. Another global variable, farrow-delay, determines how long after a sow's arrival farrowing takes place. A third global variable, lactation-time, determines how long after farrowing the pigs and sows remain. There is no good reason why I used global variables for these quantities, but it does demonstrate the flexibility of ARSE. These variables will appear in results symbolically if their values have not been determined when the laws are triggered.

*Market* is defined by laws *hog-sell* and *market-cost*. The first of these was discussed in detail in section III.A. The second is used to compute market and hauling costs, on the assumption that they are proportional to the number of hogs marketed. *Source* is the part-type which handles the purchasing of hogs.

The first three laws for part-type *hog-farm*, namely *br-input*, *groups*, and *sow-mrkt-input*, compute the breeding schedules and sow-replacement schedules. I doubt that close examination of these laws will be very instructive, but they are included for completeness. *Monthly-hogs* determines the total number of hogs present. This information is used in particular by the veterinarian part-type.

The two part-types *vet* and *worker*, require one law each to compute the costs of a veterinarian and worker respectively.

There are two laws for *miller*. The first, *grain-purchase* determines the cost of feed. The second, *mill-charge* finds the cost of running the mill.

The last part-type is *banker*, defined with just two laws. There are a lot of possibilities in this area, because it is relatively simple to write laws for part-types which correspond to well-understood financial entities. For example, different parts could have been used to divide income and loss into appropriate tax categories, or to divide it according to the interests of the partners, or both. Likewise laws might have been written to determine exposure to price risks, which might then be laid off in the commodity futures market.

## IV.D Wiring Up Hilltop

In appendix 3 we find one new feature of ***plan***. Every item under ***assumptions*** of the form "(<x> <y>)" will cause an assertion of the form "(= <x> <y>)". This feature is used to initialize some of the variables when the simulation begins. The initializations of "(hogs-out boar-source)" and "(hours Will)" demonstrate two ways to give ARSE schedules; by passing a lisp association list, or by passing a function which generates an association list. Each pair "(<i> <x>)" in the association list causes an assignment of <x> to array[<i>]. Unspecified array entries are zeroed.

## IV.E Analysis of Hilltop

If the simulation is run to completion without interruption, the results will reflect (to whatever extent made possible by the model) the operation of Hilltop Farm if the assumed prices and production are borne out. However, it is much more interesting to explore the model's performance when symbolic data are given for uncertain information. For example, we can variablize the price of feed as follows:

```
(what '(feed-price mary))
0.05 (F76)
T

(change '(feed-price mary) 'x)
((= (FEED-PRICE MARY) X) FACT F197)
NIL

(run)
DONE

(what '(balance beth))
(SS&+ #SCHEDULE-35 (SN&* #SCHEDULE-36 (&* -1.0 X))) (F263)
T
```

First, the price of feed was checked to verify the initialization. The price of feed was then changed to the variable *x*. When the cash balance is checked, we find that ARSE has solved for it <u>symbolically</u>. A single run of the simulation has given us more information than a table compiled from dozens of simulations performed while varying the price of feed.

If, on the other hand, we would like to examine the cash balance for a few different values of the price of feed, we can use the functions *vartell* and *varchange*.

```
(what '(balance beth))
(SS&+ #SCHEDULE-35 (SN&* #SCHEDULE-35 (&* -1.0 X))) (F263)
T

(vartell 'x .05)
((VALUE X 0.05) FACT F264)
NIL

(what '(balance beth))
((0 -5376.73)     (1 -5841.9485)     (2 -9916.8855)     (3 -10580.311)
 (4 -15197.246)   (5 -16402.6696)    (6 -21315.093)     (7 -25645.431)
 (8 -7543.8319)   (9 -11874.1705)    (10 6227.429)      (11 1897.0901)
 (12 21354.0893)  (13 16276.2505)    (14 35733.2495)    (15 31402.910)
 (16 50859.91)    (17 46529.571)     (18 64631.171)     (19 60300.832)
 (20 78402.43)    (21 74072.092)     (22 92173.693)     (23 87843.355)
 (24 107300.355)  (25 102970.018)    (26 121679.519)    (27 117349.18)
 (28 136806.18)   (29 132475.843)    (30 150577.443)    (31 146247.10)
 (32 164348.707)  (33 160018.367)    (34 178119.966))
(F264 F263)
T

(varchange 'x .08)
((VALUE X 0.08) FACT F266)
NIL

((0 -5520.13)     (1 -6123.0785)     (2 -10443.4755)    (3 -11346.691)
 (4 -16503.926)   (5 -18243.9797)    (6 -25303.7426)    (7 -31848.327)
 (8 -16733.876)   (9 -23278.4607)    (10 -8164.009)     (11 -14708.59)
 (12 1761.25683)  (13 -5530.8281)    (14 10939.0234)    (15 4394.4385)
 (16 20864.289)   (17 14319.704)     (18 29434.1562)    (19 22889.571)
 (20 38004.022)   (21 31459.4384)    (22 46573.891)     (23 40029.307)
 (24 56499.16)    (25 49954.576)     (26 65676.927)     (27 59132.344)
 (28 75602.197)   (29 69057.613)     (30 84172.064)     (31 77627.482)
 (32 92741.933)   (33 86197.35)      (34 101311.8))
(F266 F263)
T
```

Note that in the above dialogue, the only new facts are the new values for x. The symbolic expression for "(balance beth)" is still around, but when we ask for the value of "(balance beth)", ARSE substitutes the current value of x into the expression. None of the work of the original simulation was lost when we changed the price of feed, so ARSE can handle changes like this one with no perceptible delay.

*V Other Features of ARSE*

While the most important aspects of simulation with ARSE are included in the Hilltop simulation, some additional abilities deserve mention. In particular, *contradictions*, *choices*, and *monitors* will be described. In addition, a more general form of a law, the *rule*, will be discussed briefly.

When *equation* is called to enforce some relation, it is certainly possible that all of the variables of the relation will already have numerical values. In that case, *equation* will attempt to verify the relation. If the relation does not hold, a *contradiction* will occur. In the simplest case, this just advises the user that something is wrong and waits for him to fix it. This mechanism can be exploited by writing *monitors* to keep an eye on variables of interest. The syntax of monitors is almost exactly the same as for laws. Here is an example of a monitor which might have been written for hog farm simulations.

```
(monitor pen-capacity ((pen* pen) cap present)
          ((= (capacity !?pen*) !>cap)
           (= (hogs-present !?pen*) !>present))
          (cond ((> present cap) (contradiction))))
```

Although I have simplified a bit to avoid unpleasant details, the idea is simple. *Pen-capacity* will not assert values for "(capacity boar-pen)" or "(hogs-present !?pen*)" but it will make sure that the population is always less than or equal to the capacity of the pen. If this condition is violated, a contradiction will be generated. The only difference in implementation between *monitors* and *laws* is that *monitors* are run at a higher priority, since they may affect the value of other work.

One very powerful feature ARSE provides is that of *choices*.

Suppose that the simulation is unable to directly deduce the value of some particular terminal. In that case, ARSE can choose a default value for the terminal and proceed with the simulation. If a contradiction occurs, ARSE will back up and choose a different value for the unknown terminal. In ARSE's original domain, electronic circuit analysis, this mechanism was used to select one of several available transistor models, choosing the one most appropriate to the conditions surrounding each transistor. When a contradiction occurs, ARSE uses its explanation machinery to decide which choices the contradiction depends on, resulting in a very efficient search of the space of choices.

Finally, I'd like to mention *rules*. Actually, laws are sort of a special case of rules. The triggers of a rule need not have any part-type in common, making it possible to use rules where laws are awkward or impossible to write. However, this additional freedom is accompanied by a lot of subtle problems in writing rules, and their indiscriminate use is considered unwise. The Hilltop simulation was written using only one rule, *propagate*.

## *VI Conclusion*

I have touched on some of the problems encountered, any of which are suitable for further work. The representation of time is a long-standing problem in the field of artificial intelligence. Undoubtably better representations of time and associated mathematical operations could be devised for use in ARSE.

Understanding diagrams is an interesting topic in itself. A possible extension of ARSE would be to allow hierarchical diagram structures. Allen Brown has designed a circuit debugging program which uses hierarchical diagrams. <Brown 75>

ARSE's analytical power is certainly worthy of interest and, hopefully, emulation. As it stands, ARSE is not suitable for simulation of exceedingly complex models. Since it seems likely that ARSE's analytical power would prove useful in simulations of any size, it is worth some effort to either extend ARSE to handle more difficult models or to redesign existing simulation languages, endowing them with ARSE's analytical power.

*Appendix 1 -- List of ARSE Operators*

In general, the "&" prefix signals number arguments are required, the "s&" and "ss&" prefixes means schedule arguments are required, and the "sn&" prefix means a schedule and a number are required. Of course, any argument may be another symbolic expression, rather than a schedule or a number.

In the examples, schedules are represented by lists. A three week schedule which is always one would be represented as "(1.0 1.0 1.0)".

Functions on numbers:

| | |
|---|---|
| &+ | sum of arbitrarily many numbers |
| &- | difference of two numbers |
| &* | product of arbitrarily many numbers |
| &/ | quotient of two numbers |

Functions on schedules:

| | |
|---|---|
| ss&+ | sum of arbitrarily many schedules |
| ss&- | difference of two schedules |
| ss&* | product of arbitrarily many schedules |
| ss&/ | quotient of two schedules |
| s&dif | derivative of weekly schedule |
| s&mdif | derivative of monthly schedule |
| s&int | integral of weekly schedule |
| s&mint | integral of monthly schedule |
| s&wk-mn | converts weekly schedule to monthly schedule |

Examples:

```
(ss&+ '(0.0 3.0 4.0) '(1.0 1.0 2.0)) ==> (1.0 4.0 6.0)

(s&dif '(4.0 3.0 2.0)) ==> (4.0 -1.0 -1.0)

(s&int '(4.0 -1.0 -1.0)) ==> (4.0 3.0 2.0)
```

Functions on a schedule and a number:

| | |
|---|---|
| sn&* | product of schedule and number |
| sn&/ | quotient of schedule and number |
| sn&delay | schedule delay |

Examples:

```
(sn&* '(0.0 3.0 2.0) 2.0) ==> (0.0 6.0 4.0)

(sn&delay '(4.0 3.0 2.0) 2.0) ==> (0.0 0.0 4.0 3.0 2.0)
```

*Appendix 2 -- Definition of Hog Farm Domain*

```
(law pen-delay ((pen* pen) in out delay)
     ((= (hogs-in !?pen*) !>in)
      (= (hogs-out !?pen*) !>out)
      (= (delay !?pen*) !>delay))
     (equation 'out (sn&delay in delay) ))

(law pen-present ((pen* pen) in present out)
     ((= (hogs-in !?pen*) !>in)
      (= (hogs-out !?pen*) !>out)
      (= (hogs-present !?pen*) !>present))
     (equation 'present '(s&int (ss&- in out)) ))

(law pen-feed ((pen* pen) present feed rate)
     ((= (hogs-present !?pen*) !>present)
      (= (feed-in !?pen*) !>feed)
      (= (feed-rate !?pen*) !>rate))
     (equation 'feed '(sn&* present rate) ))

(law conceive ((br* breeding-pen) in conceived delay conc-rate)
     ((= (hogs-in !?br*) !>in)
      (= (hogs-conceived-out !?br*) !>conceived)
      (= (delay !?br*) !>delay)
      (= (conception-rate !?br*) !>conc-rate))
     (equation 'conceived '(sn&delay (sn&* in conc-rate) delay) ))

(law non-breed ((br* breeding-pen) in non delay conc-rate)
     ((= (hogs-in !?br*) !>in)
      (= (non-breeders-out !?br*) !>non)
      (= (delay !?br*) !>delay)
      (= (conception-rate !?br*) !>conc-rate))
     (equation 'non '(sn&delay (sn&* in (&- 1.0 conc-rate)) delay) ))

(law br-cum ((br* breeding-pen) in present con-out non-out)
     ((= (hogs-in !?br*) !>in)
      (= (hogs-present !?br*) !>present)
      (= (hogs-conceived-out !?br*) !>con-out)
      (= (non-breeders-out !?br*) !>non-out))
     (equation 'present '(s&int (ss&- in (ss&+ con-out non-out))) ))

(law br-feed ((br* breeding-pen) present feed rate)
     ((= (hogs-present !?br*) !>present)
      (= (feed-in !?br*) !>feed)
      (= (feed-rate !?br*) !>rate))
     (equation 'feed '(sn&* present rate) ))
```

```
; farrow-time is the total time the sows spend in the farrowing house.
; farrow-delay is the time from the sows arrival to farrow.
; lactation-time is the time from farrow to the pigs & sows departure.

(law farrow-delay ((fw-h* farrowing-house) in out)
     ((= (hogs-in !?fw-h*) !>in)
      (= (hogs-out !?fw-h*) !>out))
     (equation 'out '(sn&delay in farrow-time) ))

(law farrow ((fw-h* farrowing-house) in pigs)
     ((= (hogs-in !?fw-h*) !>in)
      (= (pigs-in !?fw-h*) !>pigs))
     (equation 'pigs '(sn&* (sn&delay in farrow-delay) litter-size) ))

(law farrow-cum-sow ((fw-h* farrowing-house) in present out)
     ((= (hogs-in !?fw-h*) !>in)
      (= (hogs-out !?fw-h*) !>out)
      (= (hogs-present !?fw-h*) !>present))
     (equation 'present '(s&int (ss&- in out)) ))

(law farrow-growers ((fw-h* farrowing-house) in out)
     ((= (pigs-in !?fw-h*) !>in)
      (= (pigs-out !?fw-h*) !>out))
     (equation 'out '(sn&delay in lactation-time) ))

(law farrow-cum-pigs ((fw-h* farrowing-house) present in out)
     ((= (pigs-in !?fw-h*) !>in)
      (= (pigs-out !?fw-h*) !>out)
      (= (pigs-present !?fw-h*) !>present))
     (equation 'present '(s&int (ss&- in out)) ))

(law farrow-feed ((fw-h* farrowing-house) present feed rate)
     ((= (hogs-present !?fw-h*) !>present)
      (= (feed-in !?fw-h*) !>feed)
      (= (feed-rate !?fw-h*) !>rate))
     (equation 'feed '(sn&* present rate) ))

(law hog-sell ((mrkt* market) hogs price income cwt)
     ((= (price/cwt !?mrkt*) !>price)
      (= (average-cwt !?mrkt*) !>cwt)
      (= (hogs-in !?mrkt*) !>hogs)
      (= (income !?mrkt*) !>income))
     (equation 'income '(s&wk-mn (sn&* hogs (&* price cwt))) ))

(law market-cost ((mrkt* market) hogs rate cost)
     ((= (hogs-in !?mrkt*) !>hogs)
      (= (haul-rate !?mrkt*) !>rate)
      (= (cost !?mrkt*) !>cost))
     (equation 'cost '(s&wk-mn (sn&* hogs rate)) ))
```

```
(law hog-buy ((src* source) price hogs expense)
     ((= (price !?src*) !>price)
      (= (hogs-out !?src*) !>hogs)
      (= (expense !?src*) !>expense))
     (equation 'expense '(s&wk-mn (sn&* hogs price)) ))

(law br-input ((f* hog-farm) s-delay s-p-g h-i-br)
     ((= (stagger-delay !?f*) !>s-delay)
      (= (sows-per-group !?f*) !>s-p-g)
      (= (hogs-in-br !?f*) !>h-i-br))
     (and (numberp s-delay)
          (equation 'h-i-br
                    "(sn&* ,(assoc-array (do ((i 0.0 (+$ i s-delay))
                                              (l nil))
                                             ((greaterp i lastweek) l)
                                             (push "(,i 1.0) l)))
                          s-p-g) )))

(law groups ((f* hog-farm) l-delay s-delay g)
     ((= (loop-delay !?f*) !>l-delay)
      (= (stagger-delay !?f*) !>s-delay)
      (= (groups !?f*) !>g))
     (equation 'g '(&// l-delay s-delay) ))

(law sow-mrkt-input ((f* hog-farm) conc-rate s-p-g
                     h-i-mrt s-delay l-delay l-s g)
     ((= (conception-rate !?f*) !>conc-rate)
      (= (sows-per-group !?f*) !>s-p-g)
      (= (hogs-in-sow-market !?f*) !>h-i-mrt)
      (= (stagger-delay !?f*) !>s-delay)
      (= (loop-delay !?f*) !>l-delay)
      (= (litters-per-sow !?f*) !>l-s)
      (= (groups !?f*) !>g))
     (and (numberp s-delay) (numberp l-delay) (numberp l-s) (numberp g)
          (equation 'h-i-mrt
                    "(sn&* ,(assoc-array
                              (do ((i (*$ l-s l-delay)
                                      (+$ i (*$ l-s l-delay)))
                                   (l nil))
                                  ((greaterp i lastweek) l)
                                  (do ((j i (+$ j s-delay)))
                                      ((= j (+$ i (*$ s-delay g))))
                                      (push "(,j 1.0) l))))
                          (&* s-p-g conc-rate)) )))

(law monthly-hogs ((f* hog-farm) wkly mnthly)
     ((= (wkly-hogs-present !?f*) !>wkly)
      (= (hogs-present !?f*) !>mnthly))
     (equation 'mnthly '(s&wk-mn wkly) ))
```

```
(law vet-cost ((v* vet) hogs rate cost)
     ((= (hogs-present !?v*) !>hogs)
      (= (rate !?v*) !>rate)
      (= (cost !?v*) !>cost))
     (equation 'cost '(sn&* hogs rate) ))

(law labor-cost ((man* worker) hrs wg cost)
     ((= (hours !?man*) !>hrs)
      (= (wage !?man*) !>wg)
      (= (cost !?man*) !>cost))
     (equation 'cost '(sn&* hrs wg) ))

(law grain-purchase ((mill* miller) out cost price)
     ((= (feed-out !?mill*) !>out)
      (= (feed-cost !?mill*) !>cost)
      (= (feed-price !?mill*) !>price))
     (equation 'cost '(s&wk-mn (sn&* out price)) ))

(law mill-charge ((mill* miller) out rate cost)
     ((= (feed-out !?mill*) !>out)
      (= (mix&grind-rate !?mill*) !>rate)
      (= (mix&grind-cost !?mill*) !>cost))
     (equation 'cost '(s&wk-mn (sn&* out rate)) ))

(law profit ((bank* banker) inc exp prof)
     ((= (income !?bank*) !>inc)
      (= (expense !?bank*) !>exp)
      (= (profit !?bank*) !>prof))
     (equation 'prof '(ss&- inc exp) ))

(law cash ((bank* banker) prof balance)
     ((= (profit !?bank*) !>prof)
      (= (balance !?bank*) !>balance))
     (equation 'balance '(s&mint prof) ))
```

*Appendix 3 -- Definition of Hilltop Model*

```
(plan hilltop 3                               ; analyze first 3 years

   (parts (breeding-pen br-pen     (delay 3.0)    (feed-rate 31.5))
          (farrowing-house fw-house               (feed-rate 91.0))

          (pen gestation-pen       (delay 13.0)   (feed-rate 31.5))
          (pen non-breeder-pen     (delay 3.0)    (feed-rate 31.5))
          (pen grower-house        (delay 4.0)    (feed-rate 62.0))
          (pen finish-house        (delay 6.0)    (feed-rate 64.8))
          (pen gilt-pen            (delay 6.0)    (feed-rate 64.8))
          (pen boar-pen            (delay 52.0)   (feed-rate 50.0))

          (market finish-market        (price/cwt 52.40) (average-cwt 2.2)
                                              (haul-rate 1.5))
          (market sow-market           (price/cwt 41.00) (average-cwt 3.7)
                                              (haul-rate 1.5))
          (market non-breeder-market   (price/cwt 45.00) (average-cwt 3.0)
                                              (haul-rate 1.5))
          (market boar-market          (price/cwt 38.00) (average-cwt 4.0)
                                              (haul-rate 1.5))

          (source sow-source  (price 100.0))
          (source boar-source (price 300.0))

          (banker Beth)
          (miller Mary        (feed-price .05)  (mix&grind-rate .0035))
          (worker Will        (wage 3.00))
          (vet Vaughn         (rate .15)))

   (connect (((hogs-conceived-out br-pen))
             ((hogs-in gestation-pen)))
            (((non-breeders-out br-pen))
             ((hogs-in non-breeder-pen)))
            (((hogs-out gestation-pen))
             ((hogs-in fw-house)))
            (((hogs-out non-breeder-pen))
             ((hogs-in non-breeder-market)))
            (((hogs-out fw-house)(hogs-out sow-source)(hogs-out gilt-pen))
             ((hogs-in br-pen) (hogs-in sow-market)))
            (((hogs-out boar-source))
             ((hogs-in boar-pen)))
            (((hogs-out boar-pen))
             ((hogs-in boar-market)))
            (((pigs-out fw-house))
             ((hogs-in grower-house)))
            (((hogs-out grower-house))
             ((hogs-in finish-house)))
            (((hogs-out finish-house))
             ((hogs-in finish-market) (hogs-in gilt-pen)))
            (((hogs-present br-pen)       (hogs-present gestation-pen)
```

```
        (hogs-present fw-house)     (hogs-present grower-house)
        (hogs-present finish-house)
        (hogs-present gilt-pen)     (hogs-present boar-pen)
        (hogs-present non-breeder-pen))
       ((wkly-hogs-present hilltop)))
      (((hogs-present hilltop))
       ((hogs-present Vaughn)))
      (((hogs-in-br hilltop))
       ((hogs-in br-pen)))
      (((hogs-in-sow-market hilltop))
       ((hogs-in sow-market)))
      (((conception-rate hilltop))
       ((conception-rate br-pen)))
      (((feed-in br-pen)          (feed-in gestation-pen)
        (feed-in fw-house)        (feed-in grower-house)
        (feed-in finish-house)
        (feed-in gilt-pen)        (feed-in boar-pen)
        (feed-in non-breeder-pen))
       ((feed-out mary)))
      (((feed-cost mary)             (expense sow-source)
        (expense boar-source)        (cost Vaughn)
        (mix&grind-cost Mary)        (cost sow-market)
        (cost boar-market)           (cost finish-market)
        (cost non-breeder-market)    (cost Will))
       ((expense Beth)))
      (((income finish-market)
        (income non-breeder-market)
        (income sow-market)
        (income boar-market))
       ((income Beth))))

(assumptions ((conception-rate hilltop) 0.90)
             (litter-size 8.0)

             ((sows-per-group hilltop) 30.0)        ; breeding plan
             ((stagger-delay hilltop) 8.0)
             ((loop-delay hilltop) 24.0)
             ((litters-per-sow hilltop) 2.0)

             (farrow-time 8)                    ; delay times for farrowing
             (farrow-delay 3)
             (lactation-time 5)

                      ; Will works 200 hours a month
             ((hours Will) (do ((i 0 (1+ i)) (l nil))
                               ((= i 65) l)
                               (setq l (cons "(,i 200.0) l))))

             ((hogs-out boar-source) ((0.0 5.0)(52.0 5.0)
                                      (104.0 5.0)))

             ((hogs-in gilt-pen) zeroarray) ))
```

*Bibliography*


<Brown 75>

Brown, A. L. 1975. *Qualitative Knowledge, Causal Reasoning, and the Localization of Failures.* Doctoral Thesis, Massachusetts Institute of Technology. Cambridge.

<Hills 72>

Hills, R. 1972. *SIMULA 67, an Introduction.* Surrey, England: Robin Hills Ltd.

<Kiviat 73>

Kiviat, P. J., Villanueva, R., and Markowitz, H. M. 1973. *Simscript II.5 Programming Language.* Los Angeles, California: Consolidated Analysis Centers Inc.

<Moon 74>

Moon, D. A. 1974. *MACLISP reference manual.* MIT Project MAC. Cambridge.

<Sussman 75>

Sussman, G. J., and Stallman, R. M. Heuristic techniques in computer aided circuit analysis. *IEEE Transactions on Circuits and Systems* November, 1975, pp. 857-865.

Forrester Jay W.
Industrial Dynamics