1991

Compiling GEN-X knowledge bases into "C"

Paul Cuddihy

Rochester Institute of Technology

School of Computer Science and Information Technology

Compiling GEN-X Knowledge Bases into "C"

by

Paul Cuddihy

A thesis, submitted to
The Faculty of the Department of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Al Biles

Andy Crapo

Peter Anderson

Table of Contents

Abstract

GEN-X is a proprietary expert system shell developed by General Electric. It supports several types of rule-based module types which use sophisticated goal-selection techniques. A compiler for the GERULE modules has been written. The modules are translated to "C" code, which shows considerable improvements in speed and space usage over the traditional interpreted version of GEN-X. This work demonstrates that all of GEN-X could be compiled to "C", and that even complex rule-based systems can be compiled into efficient code--removing the speed handicap which has characterized many expert system shells. This opens up opportunities for the use of these intuitive rule-based systems for use with time-critical applications.

Section 1. Introduction and Background

I have written a compiler which translates knowledge bases consisting of GEN-X GERULE modules into efficient and portable "C" code. Compiling this subset of GEN-X is meant to show that knowledge bases with sophisticated inference engines can be compiled, and also to provide the theoretical basis for a useful tool. With a user-provided main program and question-asking mechanism, a knowledge base can be embedded in any system which supports the "C" calling interface.

GEN-X is an expert system shell which has been developed by General Electric Corporate Research and Development over the last five years, and continues to be enhanced. GERULE modules are one of the four module types employed by GEN-X. These tables consist of goals and rules, each having a value and cost which change as rules fire. As these values and costs change, the system can choose the optimum path though a knowledge base.

This cost minimization path selection is a very important feature of GEN-X, and sets GEN-X inference apart from that of other knowledge bases. Demonstrating the ability to compile GEN-X if-then tables implies that the rest of GEN-X can be effectively compiled, and that this powerful inference technique can be implemented in a manner suitable for applications requiring high speed computations.

The compiler has been implemented in "C", and presently runs on Sun workstations. It has been tested on a number of sample modules, and performs many times quicker than regular interpreted GEN-X. Full goal mode inference functionality is supported, along with a small group of theoretically significant backplane commands (these are used to ask questions and link modules). The compiler has effectively

demonstrated that GEN-X modules can be compiled into "C" resulting in significant improvements in performance.

1.1 The GEN-X Expert System Shell

GEN-X was developed by General Electric with the intention creating an AI environment which is easy enough to use that a broad array of expert system applications would be developed throughout the company. The target users are engineering, marketing, and manufacturing personnel-- not just computer scientists[A]. The system is designed for use on small personal computers, and has been licensed to DEC for marketing under the name "Decision Expert".

The complete GEN-X system 2.2 supports a number of types of inference. There are and/or trees, decision trees, spreadsheet-like tables, and if-then tables. A backplane level allows these modules to be connected to each other and to the operating system. In addition, there are many other tools for assisting in the modification of knowledge bases and for interacting with an end user. There is even an existing "C" interface which allows GEN-X knowledge bases to be invoked by other programs.

It quickly becomes apparent that any attempt to show that GEN-X can be compiled into "C" code and achieve greatly increased efficiency must first focus on a single inference type. The results of this smaller undertaking can then be used to predict the possible success (or failure) of writing a compiler for the entire system. After consulting with some of the developers of GEN-X, it was decided that if-then tables would be the best place to start. These tables were the first module type designed for GEN-X, and they very clearly demonstrate the basic concepts of GEN-X goal-oriented (backward chaining) inference.

1.2 The GERULE Module

This section offers a basic introduction to GERULE tables. A precise description of the mechanism and theory of these modules can be found in [M] and [L], respectively, and these topics will receive much more attention as this paper progresses. Before we dive into details, it would undoubtedly help to see an example of a GERULE module. (These are often referred to as "if-then tables", and I will use the terms interchangeably.) Figure 1 and Figure 2 give an example of an if-then table.

Figure 1 An example if-then table called "filter".

FACT	B	T-VAL	T-CST	1	2	3	4	5	6	7	8	9	11
filter				>F				>T		>T	>F	>F	
sm. tot. value	B	.5000	100	T									
hot big-ask%					>T	>T	>F	T				F	
small bid-ask%	B	.2000	100		T		F						
large bid-ask%	B	.2000	100			T	F						
good bid%close	B	.5000	100					T			F		
crazy volume	B	.5000	700							T	F	F	

Figure 1 shows how an if-then table might appear to the user of GEN-X. First, take a look at the columns of rules (on the right of the table). This table uses the markers T, F, >T, and >F. T and F, as you may guess, are "true" and "false". >T and >F are short for "implies true" and "implies false".

So, rule #1 says:

- if "sm. tot. value" is true, then "filter" is false.

rule #4 says:

- if "small bid-ask%" is false AND "large bid-ask%" is false, then "hot bid-ask%" is false.

To be slightly more precise, each column is a rule whose premises must all be matched for the conclusion to be declared true or false. When a fact has many rules associated with it, the first rule that fires will set the fact's value. Simply, up and down is AND, while left and right is OR. In addition, if a second rule were to fire for the same fact, the fact's value would not change. Once a value is set, it is set.

Keep in mind also that this is an open world system. That means that failing to prove true does NOT mean that the fact is false. As we'll see later, there are other values in addition to 'True' and 'False', and additional rule markers.

Next, look at the T-val and T-cst columns. These columns in the table in figure show *true values* and *true costs*. Values are the a priori probabilities that the facts will be true. Cost is an integer representing the cost of obtaining the value of the fact. There are also *false values* and *false costs*, and a value and cost for each rule (column).

Notice, in Figure 1 on page 3, that the goal "filter" has no cost or value displayed. If a goal depends on other goals (like "filter" depends on "sm. tot. value" by rule 1) then GEN-X will calculate the costs and values when inference begins. This forward inference step is called *validation*.

For the top and middle-level goals ("filter" is a top-level goal because it affects no other goals; "hot bid-ask%" is a middle-level goal; and the rest are bottom-level goals), the true-cost may be different from the false-cost, and the values likewise dif-

ferent. Each value depends on an algebraic combination of the costs and values of the rules which the goal depends on.

However, for bottom-level goals, some assumptions are made. For example, the true and false values always sum to one [M]. The probability that the answer is unknown is not taken into consideration. Additionally, the cost of finding out that the answer is true is presumed to equal the cost of finding out that it is false. Remember, these restrictions only apply to bottom-level goals.

Figure 2 Backplane commands for the example module "filter"

```
----- FACT "sm. tot. value" -----
-BACKPLANE TO-SET: ask"is(shares * value < 2500)"
----- FACT "good bid%close" -----
-BACKPLANE TO-SET: ask "is (bid / last > 1.05)"
----- FACT "sm. tot. value" -----
-BACKPLANE TO-SET: ask"is(shares * value < 2500)"
----- FACT "good bid%close" -----
-BACKPLANE TO-SET: ask "is (bid / last > 1.05)"
----- FACT "large bid-ask%" -----
-BACKPLANE TO-SET: ask" is ((bid - ask) / last > .95)"
----- FACT "small bid-ask%" -----
-BACKPLANE TO-SET: ask "is ((bid - ask) / last <.12)"
----- FACT "crazy volume" -----
BACKPLANE TO-SET: call czvolume
```

The next obvious question is: "How do we find the answers to the bottom-level goals?" That's where the 'B' comes in. The 'B' column in Figure 1 indicates that many of the facts have backplanes. These very simple backplanes are shown in Figure 2 . This module only used to-set backplanes, which you might have guessed tell

the system how to set the values of facts. The "ask" commands give the system a string to use when prompting the user for an answer, and the "call" command (at the bottom) tells the system to run inference on a different module called "czvolume" in order to find the answer.

When inference begins, the system first does *validation* (forward-chaining) and fills in the values and cost of all the rules and goals. Then, it uses a combination of cost and value (called the "weight") to decide which top-level goal should be chosen first. (Note that "filter is true" and "filter is false" are different goals, even though they are the same fact.) It is the weight factor attached to each goal and rule which allows the inference to pick the quickest route to the answer of any particular question.

Once the top-level goal is chosen, a similar process is used to find the best rule, and so on down to a bottom-level goal. After the bottom-level goal's value is known, the rules which contain it must have their costs and values re-computed, and so on back up to top-level goal. The system will then reconsider the 'cheapest' path to solve a top-level goal, and ask the next appropriate question. New goals and rules are picked until all the top-level goals are known.

There is a consistency factor *epsilon_infer* which can also figure into the goal selection process. This will be described after values, costs, and weights are more precisely defined.

1.3 GERULE Values, Costs, and Weights

In order to build an understanding of what the compiler must achieve, it is essential that values, costs, weights, and *epsilon infer* be precisely defined. In order to

achieve this goal, a number of tables from the GEN-X User Manual [2] have been reproduced.

Values

Values are the probability that a goal will be true. Bottom level goals are given values by the knowledge-base developer. Rules and all remaining goals' values are calculated by the system. Facts (goals) have t-values and f-values, representing the probability that the fact will be true and false, respectively. These always sum to 1.

Values propagate differently, depending on whether the value is for a rule (which uses AND to combine rule markers) or a goal (which uses OR to combine rules). An *and-value* equals the probability that all children will be true. An *or-value* is one minus the probability that all the children will be false. Table 1. illustrates this.

Table 1. TABLE D-7 [2] Propagation of 'value'

and:	$V_{and} =$	$v_1 * v_2 * v_3 \ldots v_n$
or:	$V_{or} =$	$1 - [(1 - v_1)(1 - v_2)(1 - v_3) \ldots (1 - v_n)]$
not:	$V_{not} =$	$1 - v_1$
Nand:	$V_{nand} =$	$1 - V_{and}$
Nor:	$V_{nor} =$	$1 - V_{or}$

Now that we have the basics, it is time to consider that there are values possible in addition to *true*, *false*, and *unknown* (*unknown* refers to the decimal values just described). A fact can also be *can't answer* if the user decides that the answer can not be obtained. In addition, a fact can be *can't infer* when a premise is violated

(rules are called 'can't fire'). The following tables sum up this information, and were invaluable reference material during the design of inference compilation techniques:

Table 2. TABLE D-4 [2] Effect of rule premises on rule status

All premises satisfied	fire
Any premise violated	can't fire
No premise violated AND at least one 'can't answer'	can't answer
One or more premises unknown AND all others satisfied	unknown

Table 3. TABLE D-5 [2] Effect of rule status on conclusion fact

Any rule fires	T or F
All rules are 'can't fire'	can't infer
All rules are 'can't answer'	can't answer
One or more 'can't answer' and the rest 'can't fire'	can't answer
Otherwise (none fired and at least one unknown)	unknown

Weights

Weights are calculated from values and cost (best explained later). The intent is that the lowest weight goal should be asked first. [4] and [2] describe the theory which can be simplified to say that it is optimal to search for the child which is most likely to single-handedly make or break the rule.

Therefore, the *and-weight* should be lower when the value is low. That is, we first test the goals which are most likely to be false, and single-handedly render the result *can't infer*. *Or-weights* work in the opposite direction. We search first for the child most likely to be true, and give an immediate answer of true to the parent. The weight of each child should then be tempered by the cost of finding its value. High

costs push the child goal to the end of the list to be tested. This is perhaps more easily demonstrated with the following formulas:

Table 4. TABLE D-6 [2] Propagation of 'weight'

and:	$w = c / (1 - v)$
or:	$w' = c / v$

where c is cost, and v is value.

Costs

Costs are the predicted "price" of finding the value of a goal. Again, bottom level goals are given costs by the knowledge-base developer. Rules and all remaining goals' costs are calculated by the system. The cost of proving a bottom-level fact true or false is presumed equal.

Costs propagation is simple to understand, but the formulas are ugly. The cost of a parent is derived by 1) arranging the children from lowest to highest weights (first to last to test), 2) multiplying the cost of each child by the probability that we'll ever make it that far down the list, and 3) adding up the total.

Table 5. TABLE D-8 [2] Propagation of 'cost'

Values of an operator's children must be arranged from lowest to highest weight. Their values are represented $v_1 ... v_n$, their costs as $c_1 ... c_n$:

and:	$g = c_1 + v_1c_2 + v_2(c_3 + v_3(c_4 ... + v_{n-2}(c_{n-1} + v_{n-1}c_n)...)))$
or:	$g = c_1 + (1 - v_1)(c_2 + (1 - v_2)(c_3 + ... + (1 - v_{n-2})(c_{n-1} + (1 - v_{n-1})c_n)...)))$
not:	$g = c_1$
Nand:	same as and
Nor:	same as or

Epsilon_infer

The most important factor in compiling if-then tables is the fact that values, costs, and weights propagate through the table each time a new value is obtained. Therefore, the prospective path thought the knowledge is constantly changing. This can occasionally leave the user with an apparently scatter-brained inference. If cost is very important, then this is ok, since the questions will be asked in the optimum order which could have been predicted. However, it is sometimes advantageous to have the system stick to one goal at a time unless the facts swing strongly away from a certain conclusion. For this reason, GEN-X has a persistence factor, *epsilon_infer.*

Epsilon_infer is a number which specifies how much the weight of a goal must change before it is discarded in favor of a goal which recently obtained a lower weight. 0 means always asked the best goal (possibly unfocused logic from a human perspective). A very large *epsilon_infer* means stick to the original path until you are sure you are wrong.

With this overview of GEN-X inference complete, it is now time to take a look at the work others have done in compiling knowledge bases, and consider approaches to compiling GEN-X if-then tables.

1.4 Others' Work on Compiling Knowledge Bases

Over the past few years, commercially available expert system shells have been developed for nearly all major computer platforms. By far, the most popular form of expressing the knowledge in these systems is the production rule [13] (The GERULE modules fall into this category). Generally, the major drawback of this kind of knowledge based system is performance [1] [13] . This poor performance can

be attributed to the fact that these systems are usually interpreted [1] [7] [13], and they typically spend an inordinate amount of time doing goal selection [1] [13]. By the latter, I mean to say both that selecting rules takes a long time, and that the algorithms used often do not do a great job of finding the best order to pursue rules and their answers.

Although a number of people have worked on compiled knowledge bases, the inference mechanisms used by the systems I have found were much simpler than those used by GEN-X. However, much has been learned by those who have written compilers, and this knowledge was used as a solid starting point for combining compilation with GEN-X's efficient rule selection--and thereby simultaneously attacking the two major drawbacks of rule-based expert systems.

The DTD Knowledge Base Compiler

Dr. Vrba and Juan Herrera, both of Perceptics Corporation, implemented a knowledge base compiler for imaging systems [13], the Decision Tree Designer (DTD). They put forth an argument that if-then rules provided a powerful interface, but that the rules should be compiled into a decision tree for better efficiency. Their compiler did this reduction of rules into trees, but it does not determine which tree was optimal, since it lacks a mechanism for considering the cost of performing tests to obtain the expected outcomes. To overcome this, the system asked the user to choose from several "optimized trees" created through syntactic manipulation of the rules.

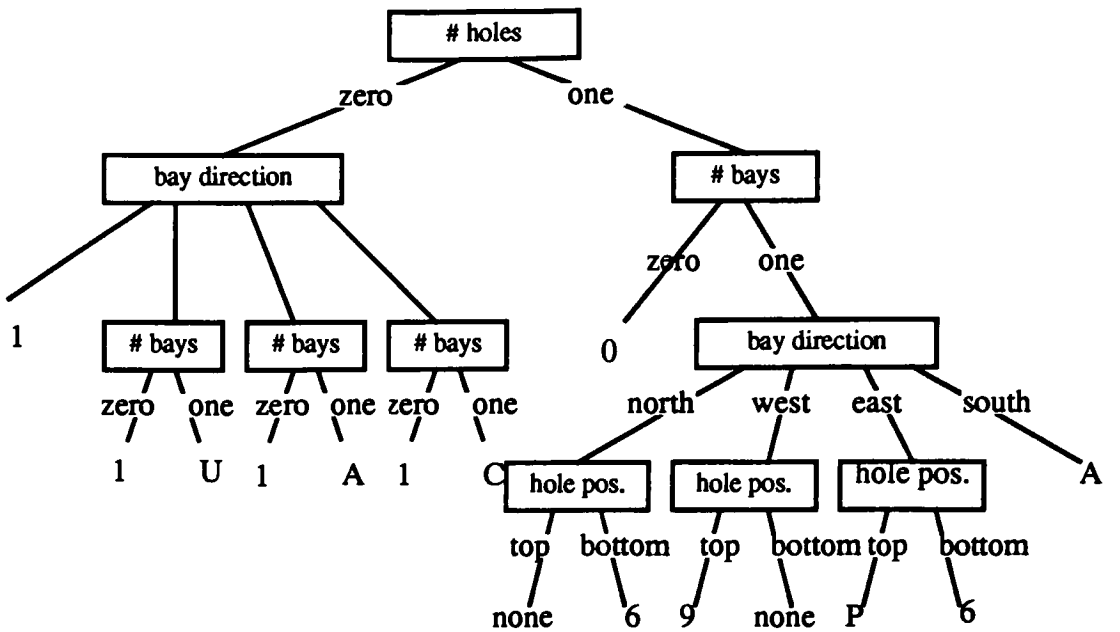
Figure 3 is offered by Vrba and Herrera as a sample of an optimized decision tree which was created from a set of if-then rules. This tree has many irreducible

forms, and each is optimal for a certain set of cost criteria. After creating many trees like the one shown, the system guides the user through the process determining the best tree, based on the cost of answering each possible question. The tree shown, for example, would be chosen if the knowledge base implementor knew that asking "number of holes" is the lowest-cost test, "hole position" was a high cost test, etc. (See [3] for a more complete discussion of this topic).

Once a decision tree is chosen, "C" code is generated. The "C" language was chosen for its portability, and the relative ease of incorporating it into existing systems. The compiler has been successfully incorporated into Perceptics Corporation's image processing products.

For comparison to GEN-X if-then tables, it is useful to note that, although costs are considered by this model, a priori probabilities for the outcomes are presumed equal. This also means that the answer to one question can not effect the probability of another answer in the tree, so the system can't be lead towards a low-cost path. This type of tree is, however, a powerful solution to a slightly simpler class of problems than one pursued by GEN-X.

Figure 3 One of the optimized forms of a decision tree for recognizing 1,6,9,0,A,C,P, and U.



INCFES, a Prolog-based Rule Compiler

Zhou Lizhu and Li Xiaoye, of Tsinghua University in Beijing, built a Prolog-based rule compiler, INCFES [11] . This is another compiler which reduces rules into trees, but the resulting trees are more robust than those produced by Vrba and Herrera's tool--and closer to the kind of trees needed for a GEN-X compiler.

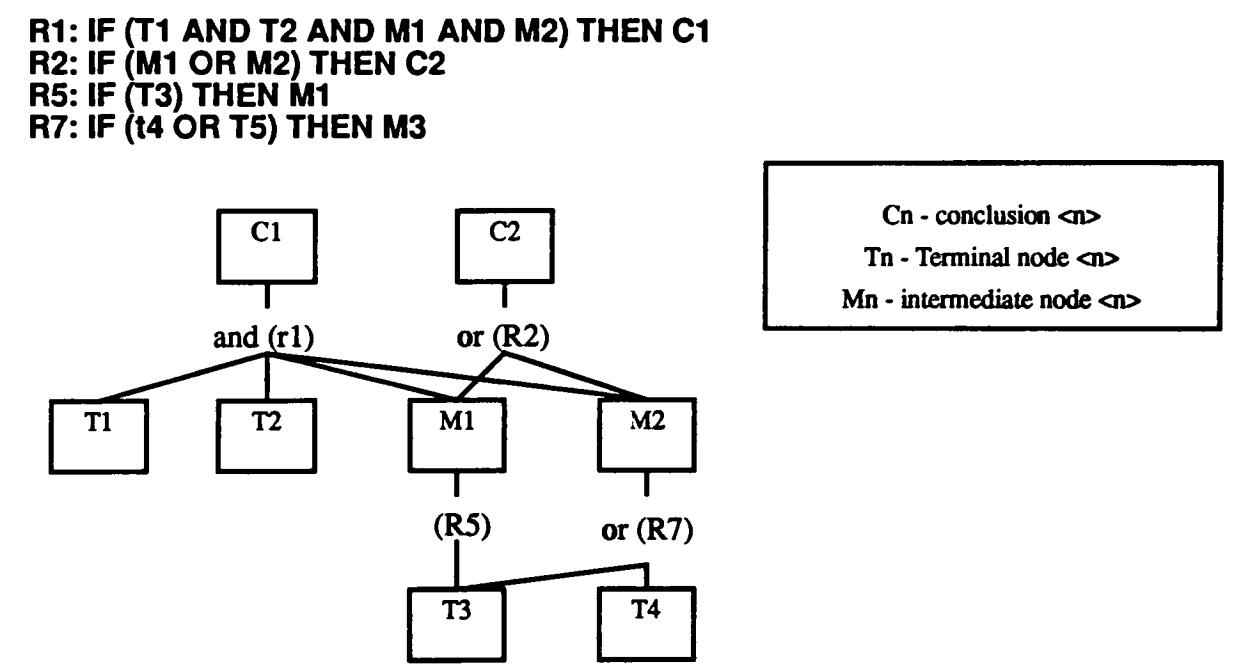
The decision trees produced by this compiler include AND and OR relationships, and allow multiple parents for a node. The trees are arranged with conclusions at the top and terminal nodes at the bottom. These bottom nodes indicated questions which can be asked by the system. The GEN-X if-then rules can be compiled into similar diagrams.

Unfortunately, Lizhu and Xiaoye's compiler does not appear to have any mechanism for choosing the best path through a decision tree based upon the cost of an-

swering each question. In other words, when the inference tries to find the value of "C1", it doesn't know which child to ask first, "T1", "M1", or "M2". In addition, if a child like "M2" is found to be true or false, the answer could effect that node's other parent, "C1". Such changes are not taken into consideration.

Finally, Lizhu and Xiaoye ran into problems with limitations on the size of Prolog's in-core database. They suggested that commercially available Prolog implementations are not sufficient for this style of compiler.

Figure 4 An INCFES inference network. A subset of Fig. 1 in [11].



TRC Rule Compiler

The expert system compiler TRC, written by Daniel Kary and Paul Juell, was designed to translate rule bases into "C" [9]. In designing TRC, the authors careful-

ly distinguished between two approaches. In the standard approach, a compiler generates knowledge base-specific data structures and generic inference engine code. By contrast, their approach was to combine the rules with the generic code to create almost completely unique code for each knowledge base which is compiled. This approach resulted in larger amounts of code produced, but better execution speed.

A welcomed side effect of this approach was "C" code that used less data space than expected, and which was very well organized--because each fact had its own function. Each fact called dependant facts' functions to try to prove or disprove themselves, and this process continued through the rule base. The builders of this system felt very strongly about using "C" for a target language because of its portability to all types and sizes of systems. In addition, they had a direct comparison of their system in "C" and LISP forms. The difference in performance was the difference between a useful tool and a scrapped project.

Although the TRC compiler offers a number of ideas for the implementation of a GEN-X if-then table compiler, the inference and goal selection techniques were once again much simpler than those used by GEN-X. Their simple backward chaining inference was missing that critical cost-minimizing feature.

Other compilers

In addition to the compilers mentioned so far, other compilers have been written to convert/compile (in some sense of the word) knowledge bases into lower level functional or procedural programming languages [12] [10] --or even to binary representations [6] . Although these types of compilations were only tangentially related

to the GEN-X compiler, a number of useful concepts were considered by the systems' authors and contributed to my work.

One fairly straight-forward point made by a few of the implementors of these systems is that entire knowledge bases can be loaded into memory before the compilation begins (as opposed to a line-by-line compilation like "C" compilers use). This allows the compiler to resolve all questions about how rules interact before any code is produced. The end result--of course--is better code.

Additionally, most systems which were successful showed increases of twenty to fifty times in speed over the interpreted versions of their knowledge bases. A performance jump in this range should, therefore be considered reasonably successful.

Also, author after author pointed out the need for modular and portable AI tools. Usually "C" was the target language of choice [8] [9] [13] because it is very well suited to achieve these goals.

To summarize the highlights of the characteristics of a compiler which have been "recommended" by authors of previous systems or of articles about rule based expert systems:

- read an entire knowledge base module into memory before generating code
- build dependency trees out of the rules and goals (like those used by INCFES)
- the code can be neatly arranged into function calls for each goal, and perhaps for each module
- writing specific code for each knowledge base (as opposed to writing data structures and generic inference code) will lead to more code, but faster code
- "C" is the language of choice
- the final code must be easily integrated into other systems [5] (i.e. like lex and yacc code can be included into "C" programs)

Most importantly, this simple model will have to be expanded on to deal with the if-then table's more advanced goal selection techniques. GEN-X inference, and the compilation techniques which are needed will be discussed next.

1.5 Glossary

Bottom-level goal - a goal with no children goals (i.e. it can't be proved by a rule)

Backplane - commands used to obtain values for bottom-level goals, and to link modules together. Only a small subset of this is supported by the compiler.

Cost - the predicted price of obtaining a fact's value.

Epsilon-infer - the persistence factor used by GERULE modules.

Fact - piece of information represented by a row in an if-then table.

Fire - give a goal a value other than UNKNOWN.

Foreplane - the rules and facts shown in the GERULE table, such as Figure 1

GERULE - if-then table Goal - a fact or rule value which the system is trying to obtain or infer.

If-then table - GERULE module, a GEN-X module type consisting of facts and rules (see Figure 1 on page 3).

Rule - logical construct represented by a column of markers in an if-then table.

Top-level goal - a goal which no parent goals (i.e. it can't be used to prove other goals.

Unknown - the state of a fact with a decimal value between zero and one

Validation - the first step in goal mode inference during which the costs and values for non bottom-level goals are calculated from the goals' children.

Value - the a-priori probability that a fact will be true.

Weight - a combination of cost and value which determines the order in which information should be asked and inferred.

Section 2. A Theoretical Model for the GEN-X Compiler

GEN-X was originally designed for small machines such as IBM PCs, and a compiler must, therefore, run in this kind of environment if it is to reach the majority of GEN-X's user base. This says a lot about the kind of restraints which will be put on the code the compiler will generate. The resulting code can be large (overlaying techniques for PCs abound), but the memory usage must be as small as possible. Speed will be important.

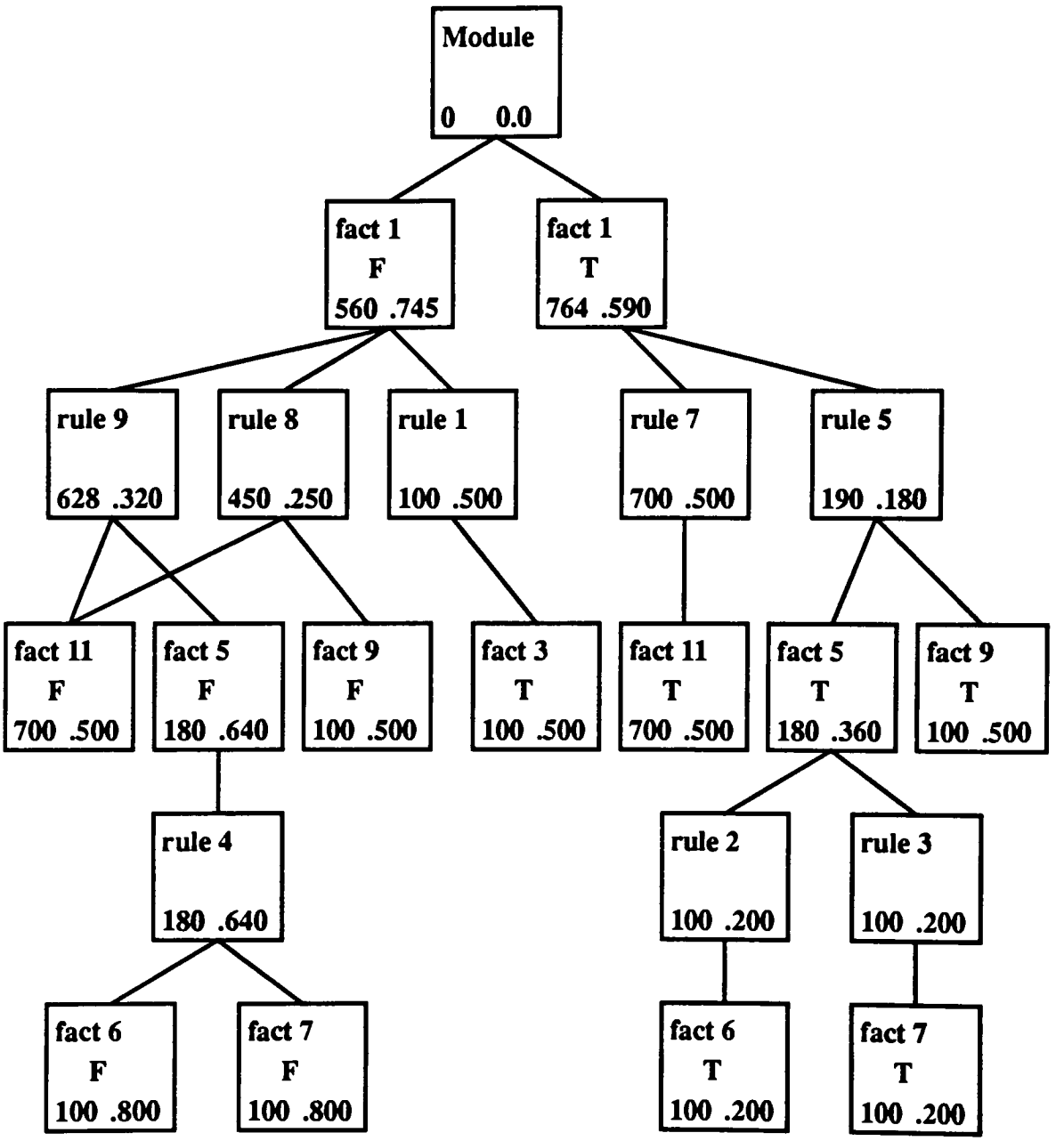
2.1 A GERULE Decision Tree

Before discussing how to compile knowledge bases, let's remember the major goals. First, the compiled code must be fast. Second, the code should fall within the constraints of smaller machines such as PC's. This means lots of code can be generated but data space must be used very sparingly (Most smaller machines can overlay code, but not data).

The work of previous compiler designers which was discussed earlier strongly suggested that the best way to integrate all of the information on costs, values, weights, etc. is to formulate a decision tree. Since the constraints of this system favor using lots of code to achieve high speed, the decision trees designed for GEN-X modules won't really exist as data structures in the compiler's output code. Instead, they will be data structures build inside the compiler, and hard coded in the module-specific output code. Thus, everything which can be known beforehand is compiled into "C" statements, leaving as few data structures as possible in the output code.

After much trial and error, I arrived at the kind of tree shown in Figure 5 . This tree represents the data in module "filter":

Figure 5 Node tree for the example "filter" (Figure 1 on page 3).



This tree shows the relationship between rules, facts, and the module. Note that facts' children are OR'ed together, while rules' children are AND'ed together.

The costs (lower left) and values (lower right) represent the state of the module after the "validation" step where the costs and values of all non-leaf nodes are filled in, but before and backward-chaining inference had begun.

The parent-child relationships work as follows: The top node is for the whole module. Its children are the top-level goals. In this case, the top-level goals are "filter" being true or being false. The node "filter" 'T' can be proved by rules7 or 5, so those nodes are the children. This pattern continues for the whole table. Compare this tree to the module display in Figure 1 on page 3, and observe how the nodes represent the goals and rules of the "filter" module.

It is particularly important to notice that for goal nodes (i.e. those which represent a fact's *true* or *false* value), the value in the lower corner is the degree to which the fact's value can be expected to match the node's 'T' or 'F' marker. For example, if fact 5 fires and becomes True, the "fact 5 'T'" node will be set to 1.0 and the "fact 5 'F'" node will become 0.0.

A reasonable exercise at this point is to list the attributes of a node which are static throughout inference, and those attributes which are dynamic. It seems that only value and cost are dynamic. This means that our implementation of each knowledge base could use very little data space (i.e. most of the data will be hard-coded and use code space). This conforms the constraint mentioned earlier: use all the code space you want, but save on the data space!

2.2 Siblings

There is one relationship between nodes which is not expressed by the tree, namely siblings. It is clear that when fact 1, for example, is found to be true, all

nodes representing fact 1 values (true nodes, false nodes, etc.) should be set to their proper values.

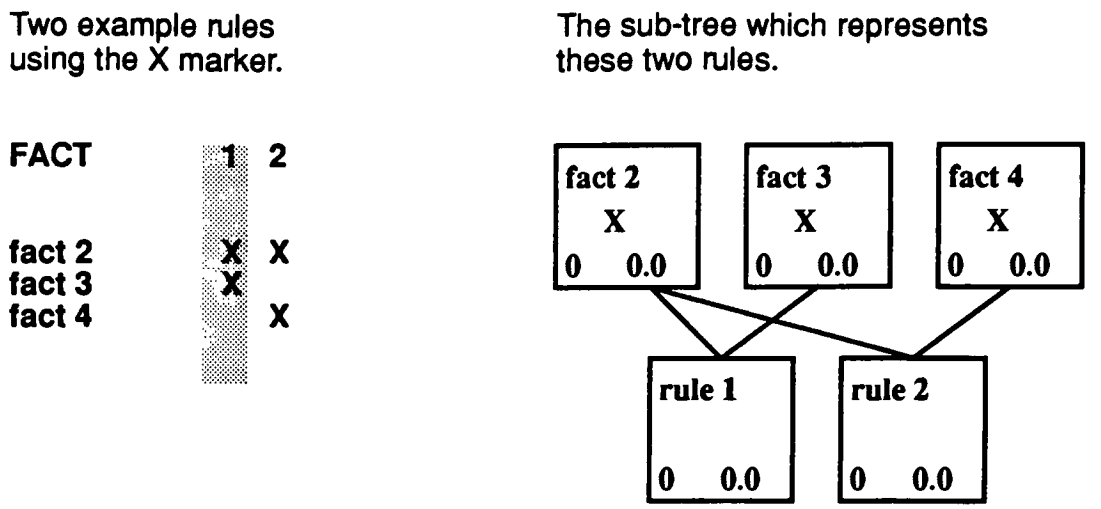
Sibling information doesn't change, so the compiler simply uses a little memory keeping track of them, then hard codes the information into the output code. (Exact implementation details will be saved for later.) Keep in mind that this relationship does exist, and that it doesn't explicitly appear in the module tree diagrams.

2.3 X markers

The other rule marker supported by the compiler is the mutual exclusion marker, 'X'. This mark is used to indicate that if one fact is known, the others in the rule must have the opposite value. Since 'X' markers are not taken into account by GEN-X goal selection, they are handled slightly differently when the module tree is built.

These 'X' rules make up their own small trees, with the 'X' marker nodes as parents, and the rule(s) as children. Inference can never make the jump to such a separate sub-tree to select a goal (and rightly so), but when a fact obtains a value in the main tree, that fact will be able to find its sibling(s) in the sub-tree(s). As values propagate through the sub-tree, those nodes will find their siblings back in the main tree, and set the values accordingly. Figure 6 shows a few 'X' rules, and their resulting sub-tree.

Figure 6 Transforming "X" rules into node sub-trees



Note that there is one way which an 'X' marker could affect goal selection. That is if a fact has only 'X' markers, and the GEN-X profile option *ask_mx_only* is set to *true*. The compiler does not presently support modules with *ask_mx_only* set to *true*. This can be justified by the fact that this is a fairly special case, and that theoretical problems are neither created nor left unsolved. A later version of the compiler could simple attach these 'X' marker sub-trees to the main tree and make a few adjustments to inference, and things would work correctly. At this point, such work would only clutter the inference theory and code, and create more work with an extremely small "bang for the buck" ratio.

2.4 Initial Value Shortcut

GEN-X has a "global fact base" (gfb) which holds values for global facts. It can be represented simply as a list of values and type information. When a particular module is run, the global fact base could contain values other than the initial values used when the knowledge base was created. In this case, some of the costs and val-

ues shown in Figure 5 would have to be transferred from the gfb, and the rest re-computed before backward-chaining inference started.

However, many prospective applications for this compiler would entail running the same module(s) (with the initial values intact) in rapid succession, resetting and recomputing the initial state of the tree each time. In these cases, it would be nice to have all the values shown in the tree in memory, in order to remove the overhead of re-computing the middle and top-level values and costs each time. This short cut uses very little memory, and has been incorporated into the model.

2.5 Weights

What about weights? These very important values can be calculated from cost and value, so they need not be stored. However, we must somehow store the relationship between weights of children of each node. This will prevent us from having to recalculate those long formulas each time we need to decide which child to try next.

The following solution has been used. Each node has a pointer to a list of children. This list will be sorted each time values and costs are propagated through the tree. The weights are never actually stored, but their relationship is evident in the results of the sorting. This results in the computation of weights only when needed, and minimum use of data space.

2.6 '*' Markers

The '*' (or star) marker is used by GEN-X to express the relationship where if one fact is known to be *true* or *false*, then a conclusion can be drawn about another fact. Just like a rule with the markers 'T' and '>T' says "if the first fact is *true*, then

the second is *true*," a rule with '*' and '>T' markers says "if the first fact is *true* or *false*, then the second is *true*."

A '*' marker is represented in the tree by a '*' fact node. This node always has a value of.9999, regardless of the costs and values of any children. Costs are computed as usual. This approach matches GEN-X 2.2, and fits into the node tree nicely.

2.7 Backplane

The backplane commands allowed by this compiler will be limited to "ask", "call", and "return". Since "call" changes modules and transfers information between modules and the global fact base during inference, and "return" exits from a module at any point, these commands cover the major theoretical challenges to compilation.

Other commands in the backplane allow for very late binding, and require interpreting. Version 3.0 of GEN-X will include 'partially compiled backplanes', written by Brian Murren of the GEN-X development team. This compiles the backplane to the extent that it can be compiled, and leaves little to be studied in this area.

2.8 "C" code

The code generated by this compiler is rather repetitive. Future implementors of similar compilers may be tempted to try to output "good" code, but this compiler simply churns out routine after routine which is almost identical, but has some numbers changed. The idea is to move as far away as possible from the notion of interpreted inference, and as close as possible to hard-coding everything. Of course the line must be--and has been--drawn somewhere, but the code generated is well with in the pre-defined constraints.

Code is divided into two types. First there is a general utility file which is included in any compiled knowledge base. Routines in this file will do the calculating of cost, value, and weights for a given node and its children. These routines were hand coded, and never change.

Each if-then table will then have its own files containing data structure definitions, and table-specific inference code. The details will be described in the next section. In general, there will be top level routines, and then many groups of similar routines. For example, our "filter" example has a top-level routine *filter*() which executes the table and returns an answer (a floating point value of the last top-level goal inferred).

Examples of "groups of similar routines" include "filter_inferN()", where N is an id number of a node. Each function performs the piece of the backward chaining which pertains to a particular node in the tree. In most cases, this means calling the inference functions of the appropriate children nodes until an answer is found or the node's weight changes beyond *epsilon_infer*. Leaf node functions will have to execute a backplane command to get an answer (the "filter_infer" functions will be described in detail later).

This approach to inference uses the function calling stack provided by "C" as the inference stack, and results in a very clean output code design. The only serious 'cheating' which occurs happens when there is a sudden need to start goal selection over from the top-level goal. This occurs most commonly when a goal fires, and it is included in more than one rule--a node with multiple parents (this could drastically change weights farther up in the tree, so GEN-X simply starts the goal selection process from scratch). In such a case, *epsilon_infer* is set to zero, and all inference func-

tions return because they think that a it's time to select a new path starting somewhere above them. The top-level function is then the only one who needs to be aware of this trick. It resets *epsilon_infer* to its real value between calls to its childrens' inference functions.

The only return values of functions on the inference stack are *true* or *false*, *true* if a backplane command has flagged the end of inference for this module, and *false* otherwise. This checking for a return flag each time reeks of interpreting a little more strongly than is desired, but it has been shown that backplanes are always going to be demanding interpretation.

This section has simply mapped out the strategy used by the compiler. Details will be given later.

2.9 Compiler Design Strategy

The first step in writing the compiler was writing a program to parse the GEN-X data files, and arrange the data into a representation of the global fact base, and of trees like Figure 5 . This data was then printed out and used to draw the trees and design output code as it should appear for the modules "filter" and "czvolume" ("czvolume" has some 'X' markers, and exercises the backplane as it is called from, and returns to "filter").

Next, 'output code' was written by hand for the two modules. Each function was written in conformance to a strict set of rules which the compiler could later use. Of course, these rules were continually being revised during the hand-compilation process. Once the hand-written code seemed to work, it was profiled using standard unix profile tools. At this point, the most CPU-hogging functions were trimmed or

corrected. This continued until the CPU time was spent mostly on calculating new costs, weights, and values (the bulk of the actual work).

When these iterations were complete, the compiler was written. It simply mimics what has been done by hand.

This approach seems to have worked very well. The compiler has not needed any major design changes, presumably because the hand compiled modules provided a thorough and well-tested specification. Careful inspection of the compiler will expose the seam between where original parsing-and-printing program and the finished compiler were joined. This seam shows the division between the first and second of the three major steps of compilation:

- reading the table from data files
- building a module tree
- writing the output code

Section 3. Compiler Description

The GEN-X compiler, "gc", compiles a GEN-X version 2.2 if-then table into "C" code which can be easily integrated into other applications. The emphasis is on proof-of-concept, but the compiler is quite usable. It supports backward ("goal") inference on complex if-then tables, with support for simple backplane commands.

The theory of this inference was summarized earlier in this paper, and the fine points (such as tie-breaking and order of goal validation) are rather confusing when described out of context. Instead, these points will be discussed as the output of the compiler is described, and compilation will be considered successful when it matches the results of GEN-X version 2.2.

3.1 Functions Performed

"Gc" compiles into "C" if-then tables containing any and all rule markers ('T', 'F', '>T', '>F', 'X' and '*'). The backplane commands "call", "return", and "ask" are supported. When a fact has no to-set backplane, the system will search for a user-provided implementation of the default "ask" backplane function. Top-level use of the compiler "gc" will be described first.

Using the Compiler

First, a knowledge base should be created using GEN-X 2.2. Once again, all modules should be if-then tables, and backplanes should be limited to "ask", "call", and "return".

The syntax for using the compiler is:

gc kb_path module_names	- compile gfb and all modules listed
gc -g kb_path module_names	- compile gfb for use with module_names
gc -m kb_path module_names	- compile modules listed

The compiler is normally used with no options. The options are meant to be used to update the target code after a minor change in the knowledge base. Any modules which reference each other through "call" commands must be compiled together.

The compiler will create the following files (where 'kbase' is the name of your knowledge base, and it contains modules 'module1', 'module2', etc.): kbase.c, kbase.h, module1.c, module1.h, module2.c, module2.h...

The following functions are included in the resulting code, and are meant to be called from the user's program:

kbase_reset()	- resets all values to their starting state.
module1_reset(), module2_reset(),...	- reset just a module to it's initial values.
float module1(), module2(),...	- run goal inference, and return the value of the last top-level goal which was set.

Note that the inference functions return the value of the last top-level goal set. This is fine conceptually, but the user could run into problems by creating modules with lots of top-level goals. It would then be unclear which fact the returned value referred to. The module called from the user's should, therefore, only have one top level goal.

Somewhere in the future, it would be trivial to implement some macros which would return the current values of facts. This would allow the user to get more detailed information about the results of inference.

User-provided "ask.h" file

The user is expected to include in the final program the files util.c and util.h, and an additional header file "ask.h". This file may be empty, but it is designed to allow the user to provide macros for use inside "ask" backplanes.

In regular GEN-X, ask backplanes send a prompt to the screen. In the compiled version, it is reasonable to want to try to answer questions as efficiently as possible. Therefore, if a node has a backplane "ask" command, the contents of the question are considered by the compiler to be a function or macro call. With the "ask.h" file, the user can provide macros for this situation. The macro should evaluate to 1.0, 0.0, or the constant CANT_ANSWER, found in "util.h". This should reflect the value of the *true* value of a fact.

If a leaf node has no backplane, then the default backplane will be inserted by the compiler (just like GEN-X 2.2). This is an 'ask("<fact name>")' command, and will be compiled to the following function, which should be provided by the user:

*float module1_ask(string) char *string;*

This function accepts a string which includes the name of the fact being asked (imbedded in a question format). The function should return1.0, 0.0, or the constant CANT_ANSWER, found in "util.h".

The ask function could actually ask the user for a value, but it will more commonly compute an answer from it's internal data and return it the compiled knowledge base. In such a case, the <string> parameter may very well be a somewhat cryptic message intended for quick interpretation by the host software, instead of an

obvious natural language question like "Is the solenoid robot's left index finger functional?".

Figure 7 shows the unlikely case where the user is simply asked a question. This is meant to provide a simple road map of how the function should work.

Figure 7 A sample user-provided ask() function

```
#include "util.h"

 float module_ask(string)
/*
** A sample ask function which simply prints the question to
** stdout, accepts either 't', 'f', or 'c', and then returns
** a value.
*/

char *string;                                   /* the questions */
{
  char yn;
  float value;
  fprintf(stdout, "\n %s? (y/n/c) ",string);    /* print question */
  fflush(stdout);                               /* Note: a typical application would be */
                                                /* able to interpret and answer the */
                                                /* question without user interaction. */
                                                /* Otherwise, regular GEN-X could */
                                                /* probably be used. */
  do
  {
        yn = getc(stdin);                       /* get a one-character answer */
        if (yn == 'y')
              value = 1.0;
        else if (yn == 'n')
              value = 0.0;
        else if (yn == 'c')
              value = CANT_ANSWER;
        else if (yn == '\n')
              yn = '\n';
        else fprintf(stdout, "\nTry again (y/n/c):");

  } while (yn != 'y' && yn != 'n' && yn != 'c');

  return(value);
}
```

3.2 Functional Restrictions

In addition to the restrictions that all modules must be if-then tables with backplanes containing only "ask", "call", and "return" commands, there are a few more subtle restrictions. First, the module inference is always goal (backward chaining) mode. This is the mode used by the overwhelming majority of modules, and the major theoretical challenge of the compiler.

There are also default environment settings:

- •default_init_value 0.5000
- •default_init_cost 100
- •disable_ca false
- •ask_mx_only false

At this point, the default initial value and initial cost are hard coded into the compiler. This means that if the user changes the defaults when developing the modules in GEN-X and compiles, the compiler could change some of the initial costs and values. Since 0.5000 and 100 are the 'default defaults', this problem is moderately obscure. Nevertheless, this is a prime target for change if this compiler were to be upgraded from a proof-of-concept to a product.

The *disable_ca* variable tells GEN-X not to allow the user to enter "Can't Answer" when asked a question. In a compiled knowledge base, the user provides the ask function, and is free to enforce this rule if it is so desired. The compiler, however, always allows the ask function to return *can't answer.*

The variable *ask_mx_only* was discussed in the 'X markers' section of this paper.

Given the user-provided ask() mechanism described earlier, there is not a mechanism for passing the rule explanations, fact explanations, or fact long names to the user. These values are simply strings, so such a mechanism could be added at some later date. This functionality is trivial, but has not been added because the real intent of having compiled GEN-X is to be able to imbed a knowledge base deep into another software package to allow that package to do high-speed AI calculations. Such a system would have no need for the additional strings which help explain the question to a human user. If a system were actually asking a user to answer the questions raised by the knowledge base, it would clearly not be in need of high-speed calculations, and could more easily use regular interpreted GEN-X.

One final clarification: the fact that the compiler supports only if-then tables and simple backplanes effectively excludes the use of variables except those fact variables created automatically during the building of an if-then table.

3.3 Input Files

The input files for the "gc" compiler are the output files of GEN-X 2.2, translated into ASCII format by the GEN-X utility, *cvtkb*. The following files are used: module1,... - module definition files kbasea.fdf - fact definition file (global fact base)

The files kbasea.ndf and kbasea.rtf are also generated by GEN-X 2.2, but their information on nodes and indexing information are not needed.

Module definition file

The module definition file holds all information about facts and their costs, values, backplanes, rule markers, etc. The files typically have the same name as the

module, and reside in a directory with the same name as the knowledge base. Excerpts from the file for the module "filter" are shown in Figure 8 .

Figure 8 Excerpts from the data file "filter"

```
Module: ift Version: 5 Flags: 36867 7/9/0/0/0 * Sequence: 14 ""Epsilon: 8.000000e+02
Variables:@@
MaxFactSlot 11
Slot: 1 GFB: 1 Cost: -1
  Expl:@@ To:@@ Post:@@
Slot: 3 GFB: 2 Cost: -1
  Expl:@@ To:@ask'"is(shares * value < 2500)'" @ Post:@@
.
.
EndFacts
MaxRuleSlot 9
Slot: 1
        Expl:@@
        Fact: 1 Marker: >f
        Fact: 3 Marker: t
  EndMarkers
Slot: 2
        Expl:@@
        Fact: 5 Marker: >t
        Fact: 6 Marker: t
  EndMarkers
.
.
EndRules
```

The top of the file has information about module type, version, flags, sequence and epsilon. The only information important to the compiler is the verification that the type is "ift", and the version is "5" (this is the GEN-X data file format version). Epsilon value is used for epsilon_infer.

The variables field is ignored (i.e. not implemented).

The MaxFactSlot line tells how many rows are in the table. It is followed by information about each row. Each row has information on position ("Slot:"), global fact

base index ("GFB"), cost, explanation, and backplanes ("To:" and "Post:"). The text fields are delimited by '@' characters. The cost field often contains -1, which indicates that the default should be used. The EndFacts line indicates that the end of the fact information has been reached.

The MaxRuleSlot line tells how many columns are in the table. It, in turn, is followed by information about each column. This information consists of column number ("Slot:"), explanation, and a list of each marker's row ("Fact:") and marker type. The EndMarkers line indicates the end of information for each rule slot. The EndRules line then announces the end of the file.

Compare Figure 7 with Figure 1, and you will see that this ascii data file's information is quite readable and readily understandable.

Fact Definition File

The fact definition file is also fairly simple. It is made up of a list of lines such as:

```
0  0  0  26  -1.0000  " "              "
1  1  6  0   0.5000   "filter"         "
2  2  5  0   -1.0000  "sm. tot. value" ""
```

The first line can be ignored, and the rest of the lines consist of

1) an unused integer,
2) a global fact base index,
3) an unused integer,
4) an unused integer,
5) initial value (-1 indicates the default),
6) the fact's short name, and
7) the fact's (unused) long name.

The global fact base index is the value seen in the module file. This is the fact's unique number, its identity. The 'unused integers' are used by GEN-X 2.2 for housekeeping and searching.

3.4 Compiler Specifications

The "gc" compiler is approximately 8500 lines of "C" code. Its structure is fairly simple and flat (compared to the programs it generates, which will demand much more discussion). Compiler basically reads in the fact definition file (GFB), sets up a simple array, and passes that array to compiling functions which output one source file and one header file.

The compiler then reads in module files one at a time, builds a tree data structure, and passes that tree to a compiling function. The compiling function acts as a switch yard, sending the tree to a series of lower-level functions. Each of these functions knows how to output a particular set of output functions (which will be described later).

Since the data flow is relatively simple, it is fair to say that the complexity of the compiler resides in its data structures, and in the design of the "C" program which the compiler writes. Once these are understood, it will become clear that the compiler consists simply of a parsing file (yacc was used), tree-building routines, and outputting routines which are made up of fairly straight-forward loops and printing statements.

The bulk of the attention will therefore be given to the compiler's data structures, the resulting code's data structures, and the resulting code's flow of control. This section will explain the compiler data structures.

The compiler's data structures

The global fact base is represented simply by an array of 'facts'. Each 'fact' contains a short name, and initial value. Since there is no explicit limit on the number of facts in a knowledge base, this array is allocated dynamically.

The module's data structure is more complex. It consists of a two- dimensional array of rule markers (the grid of 't', 'f', '>t', etc.), an epsilon infer, an array of goals, and a rule tree. The grid of rule markers and epsilon infer are very easily created, given the module file.

The array of goals is also fairly straight-forward. The array holds a gfb number, initial cost, explanation (not currently used), to-set backplane text, and post-set backplane. These values are read straight from the module definition file, with goal (n) referring to slot (n+1). Figure 9 shows the first few goals of the module "filter". These goals correspond to the fact slots in Figure 8 .

Figure 9 Example goal data structures (Compare to Figure 8 on page 35)

```
(0)   gfb: 1                                  /* Fact slot 1 has gfb pointer 1 */
      cost: 100                               /* The default cost is 100 */
      expl:
      to set:
      post set:

(1)   gfb: 0                                  /* Fact slot 2 is empty */
      cost: 100                               /* The default cost is 100 */
      expl:
      to set:
      post set:

(2)   gfb: 2                                  /* Fact slot 1 has gfb pointer 1 */
      cost: 100                               /* The default cost is 100 */
      expl:
      to set: is(shares * value < 2500)       /* This is backplane text */
      post set:
```

Finally, the rule tree is a representation of the tree shown in Figure 5 . There is a node for each kind of marker in each row of the if-then table (Figure 1), and a node for each column. For example, row 3, "sm. tot. value", has only a 'T' node, because there is no 'F' or '>F' marker in the row. That goal's false value is of no concern to the module. On the other hand, row 5, "hot bid-ask%", has a 'T' node and a 'F' node, because of all the markers in its row. 'X' and '*' markers would also have their own nodes.

The parent-child relationships between these nodes (as explained earlier) are easily determined by sight, and require the compiler to do just a little bit of looping up and down rows and columns. The actual links are stored in a pointer array. For example, if the top node in the tree (node 0) has a child pointer of 10, then the children could be found in the pointer array starting at position 10. Any values found from 10 on up until a zero is encountered would constitute that node's list of children.

This pointer array allows for easy sorting of children for lowest to highest weights. The beginning of this array is used in a similar fashion, to hold lists of siblings. This sibling information will be used, for example, to generate code that allows a row's 'T' node to find the 'F' node and set it. Obviously, the two will have to be closely related.

Compiler Flow of Control

Figure 10 shows a call tree for the compiler. This tree has been pruned to show only the major modules of the compiler, and not the routine low-level calls. Since the flow of control moves in a rather straight line from top to bottom and then loops

through the 'ift_load' and 'ift_compile' phase for each module, this chart shows flow of control rather clearly.

The functions shown in Figure 10 have the following prefixes:

- •gc - general functions
- •far - fact array (GFB) functions
- •ift - if-then table (module) functions
- •rtr - rule tree functions (the tree part of the module)
- •bpl - backplane functions

Figure 10 Code modules in the compiler, and flow of control

```
main
  gc_args                          - read command line
  gc_cvtkb                         - make data files ascii
  far_load                         - load fact array (GFB)
        yyparse
  far_compile                      - control gfb code output
        far_compile_h                    - write header file
        far_compile_c                    - write source file
              far_c_reset                        - write reset function


>> LOOP FOR EACH MODULE <<<
  ift_load                         - load module
        yyparse
  ift_compile                      - control module code output
        ift_build_tree                   - build tree
              rtr_build_tree
        ift_fill_tree                    - compute values for tree
              rtr_fill_tree
        ift_compile_h                    - write module header file
              ift_h_top                        - top of file
              ift_h_defines                    - # defines
              ift_h_vars                       - variables
                    ift_h_pt                         - pointer array
                    ift_h_node                       - node array
```

```
ift_compile_c                          - write module source file
    ift_c_top                              - top of file
    ift_c_tlevel                           - top level functions
    ift_c_infer                            - infer functions
        ift_c_infer0                           - for module node
        ift_c_linfer                           - for leaf nodes
            bpl_compile_to                         - do to-set backplanes
        ift_c_rinfer                           - rule header nodes
    ift_c_set                              - set functions
    ift_c_fire_goal                        - fire_goal functions
    ift_c_fire_rule                        - fire_rule functions
    ift_c_fire                             - fire functions (backplanes)
        bpl_compile_post                       - post-set backplanes
    ift_c_update_last                      - update_last functions
    ift_c_update_from_gfb                  - update_from_gfb functions
    ift_c_update_to_gfb                    - update_to_gfb functions
    ift_c_validate                         - write validation inference
        ift_c_val_trav                         - val_trav functions
            ift_c_vt_children                      - nodes with children
            ift_c_vt_nochild                       - noes with no children
    ift_c_x_trav                           - x_trav functions
    ift_c_reset                            - module reset function
```

In summary, the compilers flow of control and data structures are not terribly complicated. Once the tree structure (Figure 5) was decided upon, the compiler could be implemented in any of a number of ways without affecting its operation too seriously. The real challenge in building this compiler, was designing the output code. This will be explained next.

Section 4. Compiler Output Code

The knowledge base "sample," containing the modules "filter" and "czvolume", was used as the input as the compiler was designed. The output files of the finished compiler are listed in the appendices, and each file will be described in detail.

4.1 "Util" files

The first code we need for the compiled if-then module is the code that will be needed by any module. This code is not generated by the compiler, but it must be linked with any compiled knowledge base. This represents a small portion of the compiled module--well under 450 lines of code.

The file "util.h" defines some 'convenience' macros, and a type definition for the compiled version of the trees' nodes. This contains only cost and value, since they are the only things which change. In order to obtain increased efficiency, the node also remembers its previous value. That way, if an inference routine computes it again and it is the same, the routine doesn't have to continue along the tree, because all the dependent nodes won't need to be re-computed.

Figure 11 shows a list of routines in util.c.

Figure 11 Routines in the general utility file "util.c"

```
                                    /* sort nodes children by weight */
util_and_sort_children              /* - "and" nodes (rules) */
util_or_sort_children               /* - "or" nodes (goals) */

                                     /* calculate a node's weight */
cmp_or_weight
cmp_and_weight

                                    /* calculate a node's value */
util_sor_value                      /* - star '*' nodes */
util_or_value
util_and_value
                                    /* calculate a node's cost */
util_and_cost
util_or_cost
                                    /* update cost and value */
util_or_update
util_sor_update
util_and_update
```

Each of the functions simply knows how to read the pointer arrays, gather up the list of a node's children, and perform the calculations as explained in the 'compiler' theory section of this paper. As you can see, these basic calculations are the only ones which share code. The rest of the module inference is hard coded.

The files "util.h" and "util.c" are listed in Appendix C and Appendix D The file "util.h" has a small companion file "general.h" which contains some basic and obvious macro definitions.

4.2 Output Knowledge Base Files

The knowledge base output files consist of one header file and one source file. The header file (Appendix E) contains macro definitions for short name, long name (unused), and initial value for every fact in the global fact base. These macros can then be used to hard code the unchangeable attributes of facts into the compiled

code. Also in the header file, the array of floats, 'gfb' is declared. It is initialized with the initial values of all facts. A variable *G_gfb_changed* is declared, and will be used to increase efficiency by allowing modules to skip the update-from-gfb step if the global fact base has never been changed.

The source file for the knowledge base contains only one function. This *kbase_reset*() function sets all of the gfb values back to their initial values, and calls the reset functions for each module in the knowledge base. Appendix F shows the source file generated for the knowledge base "sample".

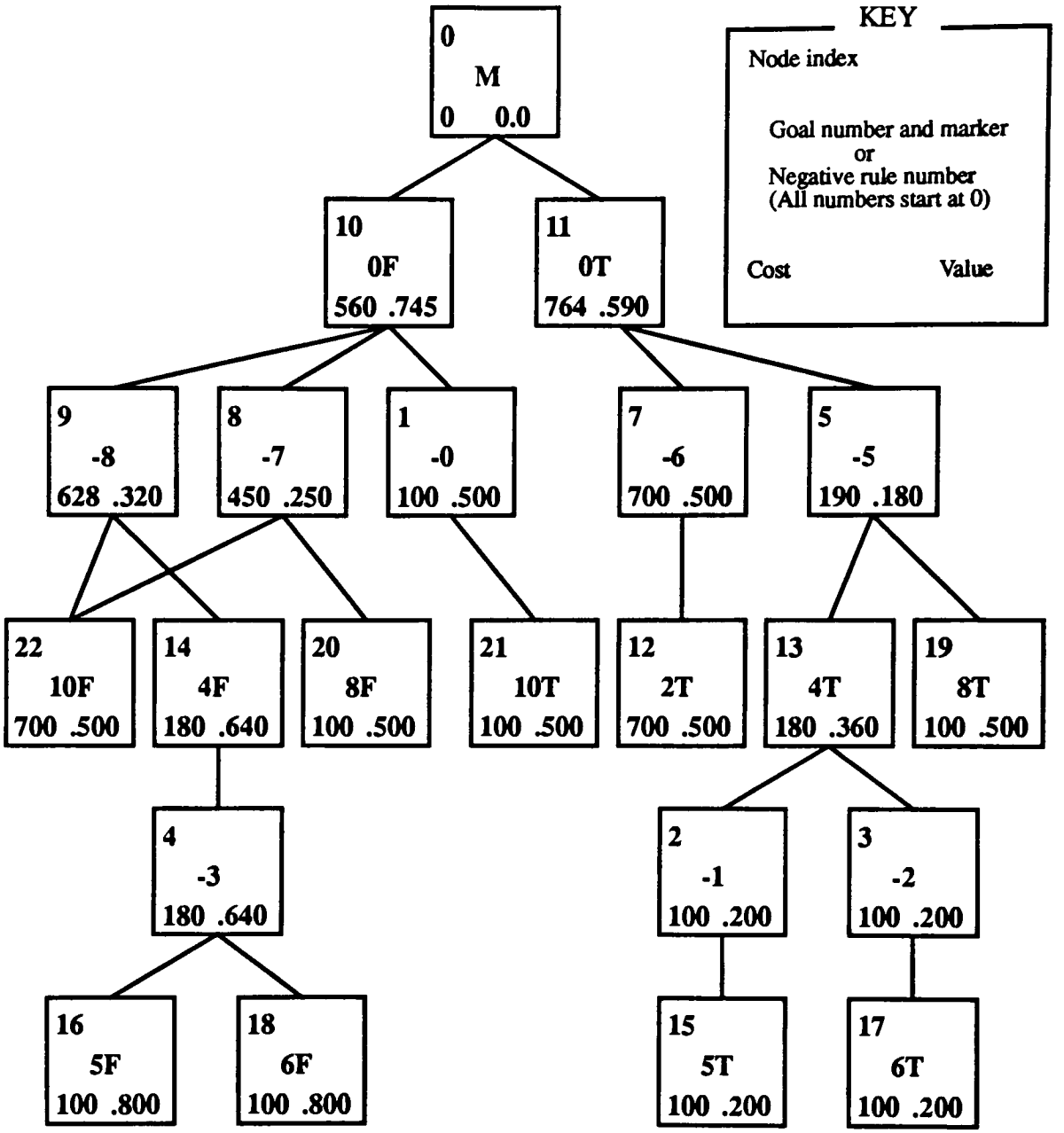
4.3 Output Module Files

Like the gfb files, the module files consist of one header file and one source file for each module. The bulk of the important concepts supporting the compiler are found in the module source file output. In order to understand the source file, the raw form of the module tree must first be presented.

Raw module trees

Figure 12 shows the actual tree generated inside the compiler for module "filter." Compare this to Figure 5 on page 20, and notice that rules and facts are indexed starting at 0, and that rule nodes are given negative numbers. The node indices in the top left corners show the nodes' position in the node array, "module_node." Note that sibling nodes (those representing different markers for the same fact) are adjacent in this array.

Figure 12 Node tree as generated by the compiler for "filter"



Once you understand how this tree relates to the simplified tree in Figure 5 on page 20, and how that tree was derived from the module "filter," this tree can be considered the starting point for the output code generation. The fact, rule, and node numbers in this tree will appear in the output code. Keep this figure handy if you plan on studying the output code in the appendices closely.

The module "czvolume" and its tree are also listed (Appendix A and Appendix B). These are useful in that they show 'X' rules, and provide a module to be called by the sample module "filter".

Module Header File

The module header file has the duty of implementing the module tree (Figure 5, again) using as little memory as possible. (Remember, the tree contains values it would hold after the validation phase if the gfb hadn't been modified.)

The file starts by declaring macros for all the parts of the tree which don't change. For each node:

- _ICOST initial cost (integer)
- _IVAL initial value (float from 0 to 1)
- _GFB global fact base index (integer)
- _BOOL node marker (t, f, x, s, or none)
- _CHILD child node index

The '_pt' array is then declared. It contains:

- pointer 0: N, a pointer to the position of the first child list in this pointer array.
- pointer 1-N: Sibling information. Nodes are put into the node array with siblings all next to each other. The first node in row 1 of the table has its index stored in pointer 1, and so on. All subsequent nodes are siblings until you reach the node listed in pointer 2, or the next pointer with a non-zero value.
- pointer N: Zero.
- pointer N+1... Child lists. The start of each list is pointed to by some node's _CHILD macro. The list is terminated by a zero. It is kept sorted from lowest to highest weight.

Finally, the node array is allocated and initialized. It contains cost, initial value, and value as of the last time the node was validated (an efficiency measure explained earlier).

The output files "filter.h" is listed in Appendix G

Module Source File

The module source file generated by the compiler is divided up into groups of similar functions. Each group of functions performs a similar role, with the individual function in the group servicing a particular goal or rule in the module.

Top-level functions

Each module output file contains *module*() and *module_call*() (in this context of function names, the word 'module' will be replaced by the name of the module being compiled). The function *module*() simply calls *module_validate*() and *module_infer0*(), to validate the module and infer the value of the top node. The top node in the tree's children are the top level goals of the module. After this is complete, the *module*() function returns the value of the top node. This is the main entry function to be called by a user who has incorporated a module into their "C" application.

The next function is *module_call*() which will be used by other module's backplanes. This function is like *module*(), except that it updates all the values from the gfb before inference, and writes them back when inference is done. This behavior accurately mimics the backplane "call" command.

The source code in Appendix H contains examples of these functions. Keep this appendix handy as the rest of the source functions are described. (Appendix J shows how the functions look for the module "czvolume")

Debugging macros

At the top of the module output file is a group of debugging functions. These are intended for debugging the compiler (not the module, which can be debugged in regular GEN-X before compilation). These statements provide information on goal setting and fact or rule firings, and roughly correspond to the information given by a module in "step" mode. This information will be used to verify that the compiled code mimics actual GEN-X.

"Infer" functions

At the top level of the backward chaining inference is the function *module_infer0*(). It loops until one of the top-level goals in the table has been fired. In each iteration of the loop, it sorts its children by weight, and resets *G_module_epsilon* to the correct value (in case one of the nodes pulled that set-epsilon-to-zero trick mentioned earlier). The lowest weight child is then picked off the list (the first one), and it's infer function is called. If the function returns true, that means that a node has signalled the end of inference, so *module_infer0*() returns true.

Rule header nodes have infer functions which are slightly different. In these cases, epsilon infer isn't changed, and children aren't sorted. The first was only a trick of infer0, and the latter will be reserved for functions which fire or validate goals. Instead, the weight of the node has to be monitored so the goal will be abandoned if the weight changes more than epsilon infer.

These rule node infer functions are set up as follows: An *old_weight* variable saves the initial weight. As long as the node hasn't fired and the weight hasn't

changed too much, the infer function of the lowest-weight child is called. Next, the new weight difference is calculated and the loop repeats.

Regular goal nodes' infer functions work the same way, except that weights are calculated differently because they are 'or' nodes instead of 'and' nodes.

Leaf nodes have very different infer functions, depending on whether there are "ask" (the default), or "call" statements in the backplane. Those with "ask" do the following: First, execute the code in the "ask" parameter string, setting the goal node to either the return value or one minus the return value (for 'F' nodes). Then, set all the siblings accordingly (with the node's "set" function). Finally, call the backplane function (the goal's "fire" function) and either return TRUE immediately if it wants to end inference, or call the goal's "fire_goal" function. This function will fire appropriate parent rules, and propagate results back up the tree.

A default "ask" statement works the same way, except that it calls the module_ask() function, since there is no "ask" parameter to execute.

If the leaf node is a "call" node, then it must first save all the 'last' values (to save unnecessary re-validations upon return), and write all the current values to the global fact base. Then the value of the node will be set to the return value of the called module's "call" function (or 1 minus the return value if it is a false node). Upon returning, the node's "set" function is called just like before, as is the goal's "fire" function. If the backplane doesn't call for an end to inference, all the nodes are updated from the global fact base before the goal's "fire_goal" function propagates the values up the tree.

"Set" functions

As was just mentioned, the "set" functions set the values of a node's siblings, depending upon its (new) value. A set function is generated for every node which has a valid global fact base pointer (i.e. not rules) as long as it is not an 'X' node, which works differently.

These functions are simple. All they do is set opposite siblings' values to opposite boolean values. For example, if we find out that goal five is true, than any '5F' nodes should be set to false (because the goal is not false). If a goal's value is "can't answer", then all siblings are set to "can't answer".

"Fire_goal" functions

The "fire_goal" functions are more complicated. There is one "fire_goal" function for each row in the table which contains markers. As described earlier, these functions are called after a value has been found, and are used to propagate the new value back up the module's tree.

If a node is a top level goal, then the module node (node zero) is simply set to the proper value, and the function ends. Otherwise, the goal's parent rules must be fired. Otherwise, parent rules and goals are affected.

In order to simulate GEN-X, the function fires the goal's parent rules in a seemingly peculiar order. The order is obtained by starting with the lowest numbered parent goal, and firing all the rules which connect the two goals, highest numbered rules first. Then continuing up through the other parent goals's rules, firing any rules which remain.

The "fire_goal" function fires the rules with "fire_rule" functions. These functions try to fire a given rule, and return TRUE only if the rule has actually fired. If a parent rule has fired, then the rule's parent goal nodes have their costs and values recalculated with by *util_or_update*(), highest nodes first. If these nodes have fired, then their siblings are set with "set" functions. Finally, the parent goals' "fire_goal" function is called. This propagates any changes up the modules tree.

If a goal has an 'X' node, there is an extra trick or two. As the very first step, an attempt is made to fire any 'X' rules with their "x_trav" functions. Then the normal "fire_rule" functions are called. Before doing the normal updating and goal firing, and if and 'X' rule has fired, all the changed 'X' nodes' parent rules must be fired, highest to lowest. The "fire_goal" function then continues as usual. These actions allow the compiled code to mimic the way GEN-X propagates 'X' rule firings before regular goal and rule firings.

The "fire_goal" function is complicated enough to warrant a look at an example in the Appendices. The function *czvolume_fire_goal3*() is a good example of an 'X' goal "fire_goal" function. Any of the filter module's "fire_goal" functions are good examples of regular functions.

"Fire_rule" functions

The "fire_rule" functions are generated for every rule in the table. This translates to any node in the tree which has no valid global fact base pointer, except node 0 which is ignored here.

These functions are called by "fire_goal" functions, and are designed to recalculate rule values. They simply call *util_and_update*() for the node and return TRUE. If the rule had already fired, the function does nothing, and returns FALSE.

"Fire" functions

These functions contain the post-set backplanes. The only legal backplane commands here are "call" and "return". A "call" works just like the to-set "call" already described, and returns FALSE. A "return" simply returns TRUE, indicating that inference for this module should end. This trick cascades up to through the "infer" functions as described earlier. Finally, if there is no backplane, the "fire" function just returns FALSE.

"Update_last" function

Each module has an "update_last" function which sets all of the nodes' "last" fields to their "value" fields. This function is called before temporarily leaving the module. Upon later return to the module, these values will allow the validation step to be performed with great efficiency.

"Update_from_gfb" and "update_to_gfb" functions

The "update_from_gfb" function updates all TRUE or FALSE goal values from the global fact base. As an efficiency measure, the function keeps track of whether or not any values were actually changed, so that unnecessary module validation doesn't take place. The "update_to_gfb" function works the same way, but in the opposite direction.

"Validate" and "val_trav" functions

Validation is the process in which forward inference is done to make sure all costs and values are propagated through the tree. This process occurs when inference is started (or continued) on a module.

The top-level function generated to run this procedure is the "validate" function. It keeps calling the top row node's "val_trav" function until no more changes have taken place in the module's tree. "Val_trav" functions call each other in a recursive fashion very similar to the way "infer" functions work. Calls to "val_trav" functions start with the top node, and propagate down to the leaf nodes, where the forward-type inference starts as the functions return and pop off the system stack.

At key times when the tree is known to be consistent (validated), the values are saved in the nodes' "last" fields. These "last" values save a lot of time during the next validation step. As long as a goal and its children don't contain a node that has become TRUE or FALSE since the last validation pass, the values and costs don't have to be recomputed.

A "val_trav" function is generated for each occupied row in the module. The functions start by arbitrarily picking a node in the goal. If that node had fired before the last validation, simply return FALSE and abandon this part of the tree. If the node has recently fired, manually set the siblings to the correct values and return TRUE.

If the node hasn't fired, the child goals' "val_trav" functions are called in groups, from highest to lowest. Each group consists of the children of the next high-

est rule node which is a child of the original node. Only the "val_trav" functions which have not yet been called are included in the group.

After each group of "val_trav" functions is called, the rule node whose children have just been validated has its values and costs updated with *util_and_update*(). The node's "last" field is also updated, indicating that the node has been validated. These two steps only occurs if one of the node's children's "val_trav" functions has returned TRUE. Otherwise, nothing has changed since the last validation.

If a goal has no children goals (i.e. it is a leaf node), then the "val_trav" function is short and simple. If needed, the "last" fields of all nodes in the goal are updated and TRUE is returned. Otherwise the "val_trav" function simply returns FALSE.

In the case that a goal has 'X' markers, the 'X' node's rule's "x_trav" functions are called, highest to lowest, at the very beginning of the "val_trav" function. The rest of the function is unaffected.

"X_trav" functions

The "x_trav" function is generated for any rule node who has parent goals with 'X' markers. It is called by the "val_trav" and "fire_goal" functions. This function fires 'X' rules.

The function simply checks to see if one parent node has fired and the other hasn't. If this is so, it sets the un-fired node to the opposite value (the 'X' rule is mutual exclusion) and sets siblings with the "set" functions.

"Reset" function

Finally, each module has a reset function. This sets all the child pointers, costs, values, and last back to there original state. This way, the module can be efficiently restored to its initial state.

Section 5. Results

The compiler has been implemented in "C" on a Sun workstations. The compiled code runs an order of magnitude or two faster than interpreted GEN-X, and amount of code generated for each module seems reasonable. Indications are that this success would be repeated on other types and sizes of machines.

5.1 Speed

The compiler generates code at a speed which appears to be about as quickly as it can announce which file it is compiling. Therefore, no further investigation has been made into the efficiency of the compiler itself.

The efficiency of the resulting "C" code has undergone some basic testing on Sun workstations. The compiled code was compared to its counterpart in interpreted GEN-X, with the GEN-X function call interface. The compiled modules were given an "ask" function which allowed the program to read answers out of a data file. The test consisted of running the module "filter" and getting the value of the last top-level goal fired. The data files gave answers in such a way as to insure that the module "filter" called the "czvolume" module each time through. This was then repeated fifty times. The average results are shown in Figure 13 .

Figure 13 _Results of performance benchmark

Regular GEN-X "sample" "filter" run 50 times:
217.6 real 197.1 user 6.7 sys

Compiled "sample" "filter" run 50 times:
4.3 real 2.0 user 0.3 sys

Improvement
51x real 99x user 22x sys

These results are meant to give general feel for the performance boost gained by compilation. It should be noted that the interpreted GEN-X spent much of its time reloading the module (because there is no other easy way to reset all the values to their original states).

On smaller machines, regular GEN-X spends much of its time just loading modules from disk and doing overlays. The compiled version will have to do no loading from disk, and very clean overlaying. At worst, a new overlay will have to be called in to memory for each module called. The time comparison between these will fall markedly in favor of the compiled version, regardless of other factors (most of which also favor the compiled version).

5.2 Size of executables

Another important to be considered is the size of the executables. The compiled knowledge base "sample" was about 65 K. This includes both modules, and the general knowledge base information. Unfortunately, there is no GEN-X program containing only the GERULE inference code with which to compare this. The program used for banchmarking was about 780 K, but that included all module types, and module editing functionality. A stand-alone version of the GEN-X enduser, which contains none of the editing features, is even larger at 1277 K

It is clear, however, that the compiled executables are of very reasonable size. Some interesting future work may include attempts reduce the size of the compiler output, trying to find an optimum balance between size and speed; and then experiments to find out how big a knowledge base must be before the compiled output code

is bigger than GEN-X 2.2. Further work could then seek creative memory handling schemes to allow for very large compiled knowledge bases to be used.

5.3 Implementation of compiled modules

Judging from the test programs written so far, the compiled modules are easily incorporated in to "C" programs. After including the header files for the knowledge base and module, and implementing an ask() function, the top-level module functions can be used in a very straight-forward manner.

Since both the compiler and the modules it produces are compiled into "C" and designed with smaller machines in mind, the compiler should be useful across many platforms. GEN-X is currently in use on PCs, Sun, Hewlett Packard, and Apollo workstations, and the compiler output could undoubtedly be ported to virtually any other machine supporting "C".

Future work on the compiler in this area should probably include changing the compiled knowledge bases to expect a single ask function (instead of one for each module), or perhaps a user-written header file to be included by each module, in which the different ask() functions could be #defined'ed to one function (this would allow room for one the module which needs a different ask() function which is bound to pop up sooner or later!). In addition, it would be useful implement macros to return the current value of specific facts after running inference, as discussed in "3.1 Functions Performed" on page 29.

5.4 Implications of the GERULE table compilation

The GEN-X expert system shell has been used within General Electric for applications ranging from diagnostics and trouble-shooting, to maintenance, certain kinds of engineering design, and problems in the financial domain. The compiler written for GERULE tables strongly implies that the rest of GEN-X could be compiled, and its use could be expanded to applications which require very quick response time.

Potentially new applications include the monitoring of systems in situations which could reap benefits from response within a few seconds (i.e. not "real-time" but somewhere closer than expert systems typically achieve). Not surprisingly, the idea for this compiler was originally conceived in response to an opportunity where a quick appraisal of a system was valuable, and regular GEN-X was estimated to take at least twenty minutes to do the job. It is estimated that the compiler could emit code which could meet the challenge in around twenty-five seconds.

As was discussed earlier, some compilers have been written for simple inference engines, but extending this idea to a robust expert system shell with theoretically advanced goal-selection techniques--such as GEN-X--is a significant step. It is becoming clear that expert system shells need not be perceived as "wonderful tools--but too slow for many applications." Instead, it is technically feasible to expand the role of these systems into more time-critical domains.

One new domain toward which compiled GEN-X would be well suited is the monitoring of complex equipment as it operates, and detecting faults as they begin to manifest themselves. This requires more speed than interpretted GEN-X, but isn't

truly "real-time." It is, however, much more valuable than waiting for an equipment failure, and then using an expert system to diagnose the dead machine.

5.5 Compiler/Interpreter pairs

On a much broader scope, it is worthwhile to comment on an apparent trend in computer languages toward systems which have a compiler and an interpreter for each language. This can be seen in as diverse languages as GEN-X (if you will look at it as a language), "C", and Smalltalk.

GEN-X having been thoroughly discussed, lets start with Smalltalk. This language offers a theoretically pristine object-oriented development environment. However, it is interpreted, and therefore can be slow. In the last few years, a number of university professors have suggested compilation, and even tried writing compilers or describing changes in the language which would enable it to run faster.

On the opposite end of the spectrum is "C", which is can be compiled into very efficient code, but even its strongest supporters will admit that it can be a bit unruly at times. In recent years, Saber Software Inc. has rather successfully marketed an interpreted "C" (and "C++") development and debugging environment (which I highly recommend...) which is now used at such places as AT&T, DEC, Kodak, Sun, M.I.T., and General Electric R&D. This offers programmers a powerful debugger with all the advantages interpreting affords, at the price of speed. Of course, when the program is completed, it can be compiled.

In these two examples, people were coming to the same conclusion from opposite directions: It can be highly advantageous to develop a software system in an interpreted environment, and compile it when it is completed. Interpreting allows for

user-friendly environments with a quick turn-around time during development and a wide array of possibilities during debugging. Compiling usually offers much better execution speed. This seems to hold true virtually all types of software systems.

As compilers are written for rule-based expert system shells, these systems will be freed from the stigma associated with slower-running interpreted tools, and used to tackle a more time-critical problems.

Bibliography

[1] Bocca, Joge B. and Freytag, Johann Christoph, "Rules for Implementing Very Large Knowledge Base Systems", SIGMOD Record, Vol. 18, No. 3, September 1989.

[2] Clark, Russell and Mangano, Tom and Boone, Kate, "GEN-X Version 2.0 Volume 1--User Manual", General Electric Corporate Research and Development, 1988.

[3] Cockett, J.R.B., "Decision Expression Optimization", Fundamental Informaticae, Vol. X, 1987, pp. 93ff.

[4] Carpi, A. W. and Side, I. L. and Simmons, M. K. and Vivier, B. J., "Use of Probability and Cost to Control Search in an Expert Diagnostic System", General Electric Corporate Research and Development, January 1987.

[5] Narrow, Barbara, "Artificial Intelligence Penetrates Corporate Environment", Info World, Vol. 12, Issue 32, August 6, 1990.

[6] Denny, Richard, "A Compiler and Loader for C-STROBE Knowledge Bases", SPIE, Vol. 937, June 1988.

[7] Ghallab, M. and Phillippe, H., "A Compiler for Real-Time Knowledge-Based Systems", IEEE International Workshop on Artificial Intelligence for Industrial Applications, June 1988.

[8] Gross, Daniel, "'C' as in 'Commercial'", Computerworld, S8, November 23, 1987.

[9] Kary, Daniel D. and Juell, Paul L., "TRC: An Expert System Compiler", SIGPLAN Notices, Vol. 21, No. 5, May 1986.

[10] Kobayashi, Yasuhiro and Mitsuta, Tooru and Wada, Yutaka, "A Knowledge Compilation Method Through Conversion of Symbolic Rules and Facts into Functions", Journal of Information Processing, Vol. 11, No.3, 1988.

[11] Lizhu, Zhou and Li, Xiaoye, "A Prolog-based Rule Compiler for Building Expert Systems", Department of Computer Science and Techniques, Tsinghua University, Beijing, China, Second IEEE Conference on Computers and Applications, January 1987.

[12] Michael, Joel A. and Ropvick, Allen A., "Knoledge Compiler for an Expert Physiology Tutor", First ESO/SMI Expert System Conference, 1982.

[13] Vrba, Joseph A. and Herrera, Juan A., "Expert Systems for Imaging", ESD: The Electronic System Design Magazine, July 1987.

Appendix A Module "czvolume" (called by filter, has 'X' markers)

FACT	B	T-VAL	T-CST	1	2	3	4	5	6	7	8	9	11
volume crazy						>T	>T	>T	>T				
volume normal	B	.8000	100	X	X				T				
volume high	B	.1000	100	X			T	F					
volume low	B	.1000	100		X	T							
bid%vol high	B	.2000	100			T							
bid%vol low	B	.2000	100				T						
ask%vol high	B	.2000	100					T					
ask%vol low	B	.2000	100						T				

----- FACT "ask%vol high" -----

-BACKPLANE TO-SET: ask "is(asksize / volume >= .25)"

----- FACT "volume high" -----

-BACKPLANE TO-SET: ask "is (volume >= 75000)"

----- FACT "volume normal" -----

-BACKPLANE TO-SET: ask "is (volume > 25000 OR volume < 75000)"

----- FACT "bid%vol high" -----

-BACKPLANE TO-SET: ask "is (bidsize / volume > .25)"

----- FACT "volume low" -----

-BACKPLANE TO-SET: ask "is (volume <=25000)"

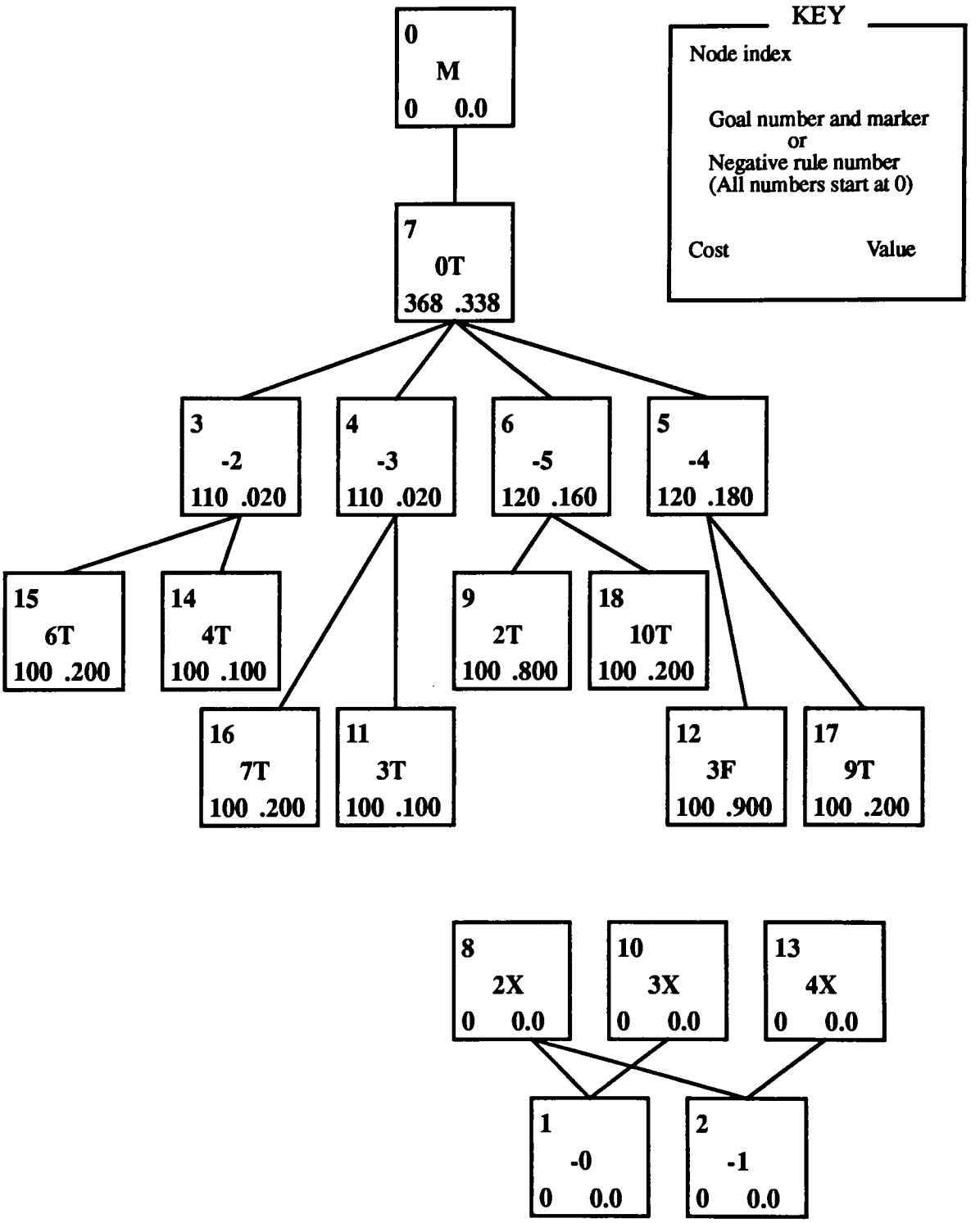
----- FACT "bid%vol low" -----

-BACKPLANE TO-SET: ask "is (bidsize / volume < .125)"

----- FACT "ask%vol low" -----

-BACKPLANE TO-SET: ask "is(asksize / volume < .12)"

Appendix B Module tree for "czvolume"



Appendix C. Utility file "util.h" and "general.h"

```
#ifndef UTIL_H
#define UTIL_H 1

typedef struct node_st
{
    short cost;
    float value;
    float last;
} NODE;

#define MAX_RULES 23
#define MAX_GOALS 23

#define GFB_UNKNOWN 2
#define GFB_TRUE    1
#define GFB_FALSE   0

/* rule and premise values */

#define CANT_INFER   0.0
#define CANT_ANSWER  -1.0
#define CANT_FIRE    0.0

#define fired(x)     (x == 1.0 || \
                      x == 0.0 || \
                      x == CANT_ANSWER)

#define torf(x)     (x == 1.0 || \
                     x == 0.0 )

#define util_and_weight(x) (x.value == 1.0?(99999):(x.cost / (1.0 - x.value)))
#define util_or_weight(x)  (x.value == 0.0?(99999):(x.cost / x.value))

long cmp_or_weight();
long cmp_and_weight();
#endif
```

```
#ifndef D_GENERAL
#define D_GENERAL

#include <stdio.h>
#include <assert.h>

#define FALSE   0
#define TRUE    1
#define NON_TF  2
#define DONE    3

#define SUCCESS 0
#define FAILURE 1

#define allocate(x)       malloc(x)
#define deallocate(x)     if ((x) != NULL) free(x)


#endif
```

Appendix D. Utility file "util.c"

```
#include "general.h"
#include "util.h"


util_and_sort_children(pnode, child)
NODE *pnode;
short *child;
/*=========================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Sort children from lowest to highest weights.
RETURN:     FALSE - already sorted
            TRUE  - sort complete
=========================================================================*/
{
    short i,j;
    short temp = -13;
    long diff;

    if (*(child + 1) == 0 || *child == 0)
        return(FALSE);

    for (i=0; *(child + i + 1) != 0; i++)
        for (j=i+1; *(child + j) != 0; j++)
        {
            if ((diff = cmp_and_weight(&pnode[*(child + i)],
                    &pnode[*(child + j)])) > 0)
            {
                temp = *(child + i);
                *(child + i) = *(child + j);
                *(child + j) = temp;
            }
            else if (diff == 0 &&
                     *(child + i) > *(child + j))    /* lower node first  */
            {
                temp = *(child + i);
                *(child + i) = *(child + j);
                *(child + j) = temp;
            }
        }

    if (temp == -13)
        return(FALSE);
    else
        return(TRUE);

}

util_or_sort_children(pnode, child)
NODE *pnode;
short *child;
/*=========================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Sort children from lowest to highest weights.
RETURN:     FALSE - already sorted
            TRUE  - sort complete
NOTE:
=========================================================================*/
{
    short i,j;
    short temp = -13;
    long  diff;

    if (*(child + 1) == 0 || *child == 0)
        return(FALSE);

    for (i=0; *(child + i + 1) != 0; i++)
```

```
        for (j=i+1; *(child + j) != 0; j++)
        {
            if ((diff = cmp_or_weight(&pnode[*(child + i)],
                    &pnode[*(child + j)])) > 0)
            {
                temp = *(child + i);
                *(child + i) = *(child + j);
                *(child + j) = temp;
            }
            else if (diff == 0 &&
                     *(child + i) > *(child + j))    /* lower node first  */
            {
                temp = *(child + i);
                *(child + i) = *(child + j);
                *(child + j) = temp;
            }
        }

    if (temp == -13)
        return(FALSE);
    else
        return(TRUE);

}
long cmp_or_weight(a,b)
NODE *a;
NODE *b;
/*=================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Compare the or weights of two nodes
RETURN:     -n if a < b
             0 if a==b
             n if a>b
NOTES: <TH>
      * return (b.cost/b.value - a.cost/a.value)
      * CANT_ANSWER sorted to the top
=================================================================*/
{
    if (a->value <= 0.0)
        return(1);
    else if (b->value <= 0.0)
        return(-1);
    else
        return((a->cost / a->value) - (b->cost / b->value));
}

long cmp_and_weight(a,b)
NODE *a;
NODE *b;
/*=================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Compare the and weights of two nodes
RETURN:     -n if a < b
             0 if a==b
             n if a>b
NOTES: <TH>
      * see cmp_or_weight comments
      * return (b.cost/(1 - b.value) - a.cost/(1 - a.value))
      * CANT_ANSWER and TRUE are sorted to the top.
=================================================================*/
{
    if (a->value == 1.0 || a->value == CANT_ANSWER)
        return(1);
    else if (b->value == 1.0 || b->value == CANT_ANSWER)
        return(-1);
    return(a->cost / (1.0 - a->value) - b->cost / (1.0 - b->value));
```

```
}

util_and_value(pn      r, child)
NODE *pnode;
short r;
short *child;
/*=================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Calculate the value of a rule node with children

NOTES:      Table D-4:
            * value of 1 means rule can FIRE.
            * value of 0 means CANT_FIRE
            * value of CANT_ANSWER means just that.
            * value between 0 and 1 means UNKNOWN

RETURN:     SUCCESS
PRE:        * cost & values of children must have been calculated
=================================================================*/
{
    short *i;
    short ca = 0;

    pnode[r].value = 1.0;
    for (i = child; *i != 0; i++)
    {
        if (pnode[*i].value == CANT_ANSWER)
            ca++;
        pnode[r].value *= pnode[*i].value;
    }

    if (pnode[r].value != 0 && ca > 0)
        pnode[r].value = CANT_ANSWER;

    return(SUCCESS);
}

util_sor_value(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*=================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Calculate the value of a node with marker "*"
RETURN:     SUCCESS
PRE:        * cost & values of children must have been calculated
=================================================================*/
{
    short *i;

    pnode[r].value = .9999;

    for (i = child; *i != 0 && pnode[*i].value != CANT_ANSWER; i++)
    {
        if (pnode[*i].value == 1.0)
            pnode[r].value = 1.0;
    }

    return(SUCCESS);
}
util_or_value(pno      , child)
NODE *pnode;
short r;
short *child;
/*=================================================================
```

```
AUTHOR:     Paul Cuddihy
SUMMARY:    Calculate the value of a rule node with children
RETURN:     SUCCESS
PRE:        * cost & values of children must have been calculated
============================================================================*/
{
    short *i;

    pnode[r].value = 1.0;

    for (i = child; *i != 0 && pnode[*i].value != CANT_ANSWER; i++)
    {
        pnode[r].value *= (1 - pnode[*i].value);
    }

    if ( i != 0 && pnode[*i].value == CANT_ANSWER)
    {
        if (pnode[r].value == 1)                /* all CA or F -> CA */
            pnode[r].value = CANT_ANSWER;
        else if (i == child)                          /* all CA -> CA */
            pnode[r].value = CANT_ANSWER;
        else
            pnode[r].value = 1 - pnode[r].value;
    }
    else
        pnode[r].value = 1 - pnode[r].value;

    return(SUCCESS);
}

util_and_cost(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*============================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Calculate the cost of a rule node with children.
            Sort the children from lowest to highest cost.
RETURN:     SUCCESS
PRE:        * cost & values of children must have been calculated
            * children must be sorted from lowest to highest weights
            * there must be children
============================================================================*/
{
    short *i;
    float cost;
    short offset;

    for (offset=0;
         *(child + offset) != 0 &&
         !fired(pnode[*(child+offset)].value);
         offset++)
        continue;
    offset --;

    cost = pnode[*(child + offset--)].cost;

    for (i = child + offset; *i != 0; i--)
    {
        cost *= pnode[*i].value;
        cost += pnode[*i].cost;
    }

    pnode[r].cost = (short) cost;

    return(SUCCESS);
```

```
}

util_or_cost(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*=============================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Calculate the cost of a rule node with children.
            Sort the children from lowest to highest cost.
RETURN:     SUCCESS
PRE:        * cost & values of children must have been calculated
            * children must be sorted from lowest to highest weights
            * there must be children
=============================================================================*/
{
    short *i;
    float cost;
    short offset;


    for (offset=0;
         *(child + offset) != 0 &&
         !fired(pnode[*(child+offset)].value);
         offset++)
        continue;

    offset --;

    if (offset < 0)
        return(SUCCESS);

    cost = pnode[*(child + offset--)].cost;

    for (i = child + offset; *i != 0 ; i--)
    {
        cost *= (1.0 - pnode[*i].value);
        cost += pnode[*i].cost;
    }

    pnode[r].cost = (short) cost;

    return(SUCCESS);
}



util_and_update(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*=============================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Update costs and values for an and-node
INTENT:     * if node has fired, do nothing
            * sort, get value and cost for node

RETURN:     TRUE  - if new value is TRUE
            FALSE - otherwise

NOTE:       The reason for the return values is that TRUE is the only
            value which can be arrived at through calculations that
            should be propagated to other sibling nodes.
=============================================================================*/
{
    if (fired(pnode[r].value))
```

```
        return(FALSE);

    util_and_sort_children(pnode, child);
    util_and_value(pnode, r, child);
    util_and_cost(pnode, r, child);

    if (pnode[r].value == 1.0)
        return(TRUE);
    else return(FALSE);
}

util_sor_update(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*=============================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Update costs and values for an or-node with marker *
INTENT:     * if node has fired, do nothing
            * sort, get value and cost for node

RETURN:     TRUE  - if new value is TRUE or FALSE
            FALSE - otherwise

NOTE:       util_or_sort_children() is used.  The goal selection could
            differ from Gen-x 2.2, and this has been confirmed as a
            bug in Gen-x 2.2 (9/21/90)  which will be fixed in the
            next release.
            Example:
                 1   2   3      Gen-x 2.2 picks rule (2) before rule (3)
 A              >T              even though (3) has a lower or-weight.
 B               *  >T  >F
 C     .5 100            *     w(2) = 100 / .5     = 200
 D     .5 100        T         w(3) = 100 / .9999 ~= 100

                               This compiler correctly picks rule 3.
=============================================================================*/
{
    if (fired(pnode[r].value))
        return(FALSE);

    util_or_sort_children(pnode, child);
    util_sor_value(pnode, r, child);
    util_or_cost(pnode, r, child);

    if (torf(pnode[r].value))
        return(TRUE);
    else return(FALSE);
}

util_or_update(pnode, r, child)
NODE *pnode;
short r;
short *child;
/*=============================================================================
AUTHOR:     Paul Cuddihy
SUMMARY:    Update costs and values for an or-node
INTENT:     * if node has fired, do nothing
            * sort, get value and cost for node

RETURN:     TRUE  - if new value is TRUE or FALSE

=============================================================================*/
{
    if (fired(pnode[r].value))
        return(FALSE);
```

```
    util_or_sort_children(pnode, child);
    util_or_value(pnode, r, child);
    util_or_cost(pnode, r, child);

    if (torf(pnode[r].value))
        return(TRUE);
    else return(FALSE);
}
```

Appendix E. GFB output header file "sample.h"

```
#ifndef SAMPLE_GFB_H
#define SAMPLE_GFB_H  1

#include "util.h"

#define GFB_SIZE    33

#define SAMPLE_SNAME0   ""
#define SAMPLE_LNAME0   ""
#define SAMPLE_IVAL0    0.5
#define SAMPLE_SNAME1   "filter"
#define SAMPLE_LNAME1   ""
#define SAMPLE_IVAL1    0.500000
#define SAMPLE_SNAME2   "sm. tot. value"
#define SAMPLE_LNAME2   ""
#define SAMPLE_IVAL2    0.500000
#define SAMPLE_SNAME3   "volume crazy"
#define SAMPLE_LNAME3   ""
#define SAMPLE_IVAL3    0.500000
#define SAMPLE_SNAME4   "small bid-ask%"
#define SAMPLE_LNAME4   ""
#define SAMPLE_IVAL4    0.200000
#define SAMPLE_SNAME5   "large bid-ask%"
#define SAMPLE_LNAME5   ""
#define SAMPLE_IVAL5    0.200000
#define SAMPLE_SNAME6   "hot   bid-ask%"
#define SAMPLE_LNAME6   ""
#define SAMPLE_IVAL6    0.100000
#define SAMPLE_SNAME7   "good bid%close"
#define SAMPLE_LNAME7   ""
#define SAMPLE_IVAL7    0.500000
#define SAMPLE_SNAME8   "crazy volume"
#define SAMPLE_LNAME8   ""
#define SAMPLE_IVAL8    0.500000
#define SAMPLE_SNAME9   "volume high"
#define SAMPLE_LNAME9   ""
#define SAMPLE_IVAL9    0.100000
#define SAMPLE_SNAME10   "volume low"
#define SAMPLE_LNAME10   ""
#define SAMPLE_IVAL10    0.100000
#define SAMPLE_SNAME11   "bid%vol high"
#define SAMPLE_LNAME11   ""
#define SAMPLE_IVAL11    0.200000
#define SAMPLE_SNAME12   "bid%vol low"
#define SAMPLE_LNAME12   ""
#define SAMPLE_IVAL12    0.200000
#define SAMPLE_SNAME13   "ask%vol high"
#define SAMPLE_LNAME13   ""
#define SAMPLE_IVAL13    0.200000
#define SAMPLE_SNAME14   "ask%vol low"
#define SAMPLE_LNAME14   ""
#define SAMPLE_IVAL14    0.200000
#define SAMPLE_SNAME15   "volume normal"
#define SAMPLE_LNAME15   ""
#define SAMPLE_IVAL15    0.800000
#define SAMPLE_SNAME16   ""
#define SAMPLE_LNAME16   ""
#define SAMPLE_IVAL16    0.000000
#define SAMPLE_SNAME17   ""
#define SAMPLE_LNAME17   ""
#define SAMPLE_IVAL17    0.000000
#define SAMPLE_SNAME18   "con2"
#define SAMPLE_LNAME18   ""
#define SAMPLE_IVAL18    0.500000
#define SAMPLE_SNAME19   "con1"
#define SAMPLE_LNAME19   ""
```

```
#define SAMPLE_IVAL19    0.500000
#define SAMPLE_SNAME20   "tx top"
#define SAMPLE_LNAME20   ""
#define SAMPLE_IVAL20    0.500000
#define SAMPLE_SNAME21   "tx one"
#define SAMPLE_LNAME21   ""
#define SAMPLE_IVAL21    0.500000
#define SAMPLE_SNAME22   ""
#define SAMPLE_LNAME22   ""
#define SAMPLE_IVAL22    0.000000
#define SAMPLE_SNAME23   ""
#define SAMPLE_LNAME23   ""
#define SAMPLE_IVAL23    0.000000
#define SAMPLE_SNAME24   ""
#define SAMPLE_LNAME24   ""
#define SAMPLE_IVAL24    0.000000
#define SAMPLE_SNAME25   ""
#define SAMPLE_LNAME25   ""
#define SAMPLE_IVAL25    0.000000
#define SAMPLE_SNAME26   ""
#define SAMPLE_LNAME26   ""
#define SAMPLE_IVAL26    0.000000
#define SAMPLE_SNAME27   ""
#define SAMPLE_LNAME27   ""
#define SAMPLE_IVAL27    0.000000
#define SAMPLE_SNAME28   "factback2"
#define SAMPLE_LNAME28   ""
#define SAMPLE_IVAL28    0.500000
#define SAMPLE_SNAME29   ""
#define SAMPLE_LNAME29   ""
#define SAMPLE_IVAL29    0.000000
#define SAMPLE_SNAME30   "backfact"
#define SAMPLE_LNAME30   ""
#define SAMPLE_IVAL30    0.500000
#define SAMPLE_SNAME31   ""
#define SAMPLE_LNAME31   ""
#define SAMPLE_IVAL31    0.500000
#define SAMPLE_SNAME32   ""
#define SAMPLE_LNAME32   ""
#define SAMPLE_IVAL32    0.500000

#ifdef SAMPLE_GFB_C

#include "general.h"

short G_gfb_changed = FALSE;

float gfb[GFB_SIZE] = {
        SAMPLE_IVAL0,
        SAMPLE_IVAL1,
        SAMPLE_IVAL2,
        SAMPLE_IVAL3,
        SAMPLE_IVAL4,
        SAMPLE_IVAL5,
        SAMPLE_IVAL6,
        SAMPLE_IVAL7,
        SAMPLE_IVAL8,
        SAMPLE_IVAL9,
        SAMPLE_IVAL10,
        SAMPLE_IVAL11,
        SAMPLE_IVAL12,
        SAMPLE_IVAL13,
        SAMPLE_IVAL14,
        SAMPLE_IVAL15,
        SAMPLE_IVAL16,
        SAMPLE_IVAL17,
```

```
        SAMPLE_IVAL18,
        SAMPLE_IVAL19,
        SAMPLE_IVAL20,
        SAMPLE_IVAL21,
        SAMPLE_IVAL22,
        SAMPLE_IVAL23,
        SAMPLE_IVAL24,
        SAMPLE_IVAL25,
        SAMPLE_IVAL26,
        SAMPLE_IVAL27,
        SAMPLE_IVAL28,
        SAMPLE_IVAL29,
        SAMPLE_IVAL30,
        SAMPLE_IVAL31,
        SAMPLE_IVAL32};

#else

extern float gfb[GFB_SIZE];
extern short G_gfb_changed;


#endif SAMPLE_GFB_C
#endif SAMPLE_GFB_H
```

Appendix F. GFB output source file "sample.c"

```
#define SAMPLE_GFB_C
#include "sample.h"


sample_reset()
{
    G_gfb_changed = FALSE;

    gfb[0] = SAMPLE_IVAL0;
    gfb[1] = SAMPLE_IVAL1;
    gfb[2] = SAMPLE_IVAL2;
    gfb[3] = SAMPLE_IVAL3;
    gfb[4] = SAMPLE_IVAL4;
    gfb[5] = SAMPLE_IVAL5;
    gfb[6] = SAMPLE_IVAL6;
    gfb[7] = SAMPLE_IVAL7;
    gfb[8] = SAMPLE_IVAL8;
    gfb[9] = SAMPLE_IVAL9;
    gfb[10] = SAMPLE_IVAL10;
    gfb[11] = SAMPLE_IVAL11;
    gfb[12] = SAMPLE_IVAL12;
    gfb[13] = SAMPLE_IVAL13;
    gfb[14] = SAMPLE_IVAL14;
    gfb[15] = SAMPLE_IVAL15;
    gfb[16] = SAMPLE_IVAL16;
    gfb[17] = SAMPLE_IVAL17;
    gfb[18] = SAMPLE_IVAL18;
    gfb[19] = SAMPLE_IVAL19;
    gfb[20] = SAMPLE_IVAL20;
    gfb[21] = SAMPLE_IVAL21;
    gfb[22] = SAMPLE_IVAL22;
    gfb[23] = SAMPLE_IVAL23;
    gfb[24] = SAMPLE_IVAL24;
    gfb[25] = SAMPLE_IVAL25;
    gfb[26] = SAMPLE_IVAL26;
    gfb[27] = SAMPLE_IVAL27;
    gfb[28] = SAMPLE_IVAL28;
    gfb[29] = SAMPLE_IVAL29;
    gfb[30] = SAMPLE_IVAL30;
    gfb[31] = SAMPLE_IVAL31;
    gfb[32] = SAMPLE_IVAL32;

    czvolume_reset();
    filter_reset();
}
```

Appendix G. Header file "filter.h" generated by the compiler

```
#ifndef FILTER
#define FILTER
#include "util
#include "sampl

#define FILTER_N       23
#define FILTER_E       N 800.000000


#define FILTER_ICO  :0   0
#define FILTER_IVA_0     0.500000
#define FILTER_GFB0      -1
#define FILTER_BOOL0     GFB_UNKNOWN
#define FILTER_CHILD0    (filter_pt + 1)

#define FILTER_ICOST1    100
#define FILTER_IVAL1     0.500000
#define FILTER_GFB1      -1
#define FILTER_BOOL1     GFB_UNKNOWN
#define FILTER_CHILD1    (filter_pt + 4)

#define FILTER_ICOST2    100
#define FILTER_IVAL2     0.200000
#define FILTER_GFB2      -1
#define FILTER_BOOL2     GFB_UNKNOWN
#define FILTER_CHILD2    (filter_pt + 6)

#define FILTER_ICOST3    100
#define FILTER_IVAL3     0.200000
#define FILTER_GFB3      -1
#define FILTER_BOOL3     GFB_UNKNOWN
#define FILTER_CHILD3    (filter_pt + 8)

#define FILTER_ICOST4    180
#define FILTER_IVAL4     0.640000
#define FILTER_GFB4      -1
#define FILTER_BOOL4     GFB_UNKNOWN
#define FILTER_CHILD4    (filter_pt + 10)

#define FILTER_ICOST5    190
#define FILTER_IVAL5     0.180000
#define FILTER_GFB5      -1
#define FILTER_BOOL5     GFB_UNKNOWN
#define FILTER_CHILD5    (filter_pt + 13)

#define FILTER_ICOST6    0
#define FILTER_IVAL6     0.000000
#define FILTER_GFB6      -1
#define FILTER_BOOL6     GFB_UNKNOWN
#define FILTER_CHILD6    (filter_pt + 16)

#define FILTER_ICOST7    700
#define FILTER_IVAL7     0.500000
#define FILTER_GFB7      -1
#define FILTER_BOOL7     GFB_UNKNOWN
#define FILTER_CHILD7    (filter_pt + 17)

#define FILTER_ICOST8    450
#define FILTER_IVAL8     0.250000
#define FILTER_GFB8      -1
#define FILTER_BOOL8     GFB_UNKNOWN
#define FILTER_CHILD8    (filter_pt + 19)

#define FILTER_ICOST9    628
#define FILTER_IVAL9     0.320000
#define FILTER_GFB9      -1
```

```
#define FILTER_BOOL9    GFB_UNKNOWN
#define FILTER_CHILD9   (filter_pt + 22)

#define FILTER_ICOST10  764
#define FILTER_IVAL10   0.590000
#define FILTER_GFB10    1
#define FILTER_BOOL10   GFB_TRUE
#define FILTER_CHILD10  (filter_pt + 25)

#define FILTER_ICOST11  560
#define FILTER_IVAL11   0.745000
#define FILTER_GFB11    1
#define FILTER_BOOL11   GFB_FALSE
#define FILTER_CHILD11  (filter_pt + 28)

#define FILTER_ICOST12  100
#define FILTER_IVAL12   0.500000
#define FILTER_GFB12    2
#define FILTER_BOOL12   GFB_TRUE
#define FILTER_CHILD12  (filter_pt + 32)

#define FILTER_ICOST13  180
#define FILTER_IVAL13   0.360000
#define FILTER_GFB13    6
#define FILTER_BOOL13   GFB_TRUE
#define FILTER_CHILD13  (filter_pt + 33)

#define FILTER_ICOST14  180
#define FILTER_IVAL14   0.640000
#define FILTER_GFB14    6
#define FILTER_BOOL14   GFB_FALSE
#define FILTER_CHILD14  (filter_pt + 36)

#define FILTER_ICOST15  100
#define FILTER_IVAL15   0.200000
#define FILTER_GFB15    4
#define FILTER_BOOL15   GFB_TRUE
#define FILTER_CHILD15  (filter_pt + 38)

#define FILTER_ICOST16  100
#define FILTER_IVAL16   0.800000
#define FILTER_GFB16    4
#define FILTER_BOOL16   GFB_FALSE
#define FILTER_CHILD16  (filter_pt + 39)

#define FILTER_ICOST17  100
#define FILTER_IVAL17   0.200000
#define FILTER_GFB17    5
#define FILTER_BOOL17   GFB_TRUE
#define FILTER_CHILD17  (filter_pt + 40)

#define FILTER_ICOST18  100
#define FILTER_IVAL18   0.800000
#define FILTER_GFB18    5
#define FILTER_BOOL18   GFB_FALSE
#define FILTER_CHILD18  (filter_pt + 41)

#define FILTER_ICOST19  100
#define FILTER_IVAL19   0.500000
#define FILTER_GFB19    7
#define FILTER_BOOL19   GFB_TRUE
#define FILTER_CHILD19  (filter_pt + 42)

#define FILTER_ICOST20  100
#define FILTER_IVAL20   0.500000
                        7
```

```
#define FILTER_BC        GFB_FALSE
#define FILTER_C         (filter_pt + 43)

#define FILTER_          700
#define FILTER           0.500000
#define FILTER           8
#define FILTER_      1   GFB_TRUE
#define FILTER_C....21   (filter_pt + 44)

#define FILTER_ICOST22   700
#define FILTER_IVAL22    0.500000
#define FILTER_GFB22     8
#define FILTER_BOOL22    GFB_FALSE
#define FILTER_CHILD22   (filter_pt + 45)

#ifdef FILTER_C


short filter_pt[] = {0,11,10,0,12,0,15,0,17,0,16,18,0,19,13,0,0,21,0,20,22,0,14,2


NODE filter_node[FILTER_NODES] = {
    {FILTER_ICOST0, FILTER_IVAL0, FILTER_IVAL0},
    {FILTER_ICOST1, FILTER_IVAL1, FILTER_IVAL1},
    {FILTER_ICOST2, FILTER_IVAL2, FILTER_IVAL2},
    {FILTER_ICOST3, FILTER_IVAL3, FILTER_IVAL3},
    {FILTER_ICOST4, FILTER_IVAL4, FILTER_IVAL4},
    {FILTER_ICOST5, FILTER_IVAL5, FILTER_IVAL5},
    {FILTER_ICOST6, FILTER_IVAL6, FILTER_IVAL6},
    {FILTER_ICOST7, FILTER_IVAL7, FILTER_IVAL7},
    {FILTER_ICOST8, FILTER_IVAL8, FILTER_IVAL8},
    {FILTER_ICOST9, FILTER_IVAL9, FILTER_IVAL9},
    {FILTER_ICOST10, FILTER_IVAL10, FILTER_IVAL10},
    {FILTER_ICOST11, FILTER_IVAL11, FILTER_IVAL11},
    {FILTER_ICOST12, FILTER_IVAL12, FILTER_IVAL12},
    {FILTER_ICOST13, FILTER_IVAL13, FILTER_IVAL13},
    {FILTER_ICOST14, FILTER_IVAL14, FILTER_IVAL14},
    {FILTER_ICOST15, FILTER_IVAL15, FILTER_IVAL15},
    {FILTER_ICOST16, FILTER_IVAL16, FILTER_IVAL16},
    {FILTER_ICOST17, FILTER_IVAL17, FILTER_IVAL17},
    {FILTER_ICOST18, FILTER_IVAL18, FILTER_IVAL18},
    {FILTER_ICOST19, FILTER_IVAL19, FILTER_IVAL19},
    {FILTER_ICOST20, FILTER_IVAL20, FILTER_IVAL20},
    {FILTER_ICOST21, FILTER_IVAL21, FILTER_IVAL21},
    {FILTER_ICOST22, FILTER_IVAL22, FILTER_IVAL22},
};


float G_filter_epsilon =  FILTER_EPSILON;

#else

extern short *filter_pt;
extern NODE filter_node[FILTER_NODES];
extern float G_filter_epsilon;

extern float filter();
#endif FILTER_C
#endif FILTER_H
```

Appendix H. Source file "filter.c" generated by the compiler

```
#define FILTER_C 1
#include "sample.h"
#include "filter.h"
#include "general.h"
#include "ask.h"

#ifdef DEBUG

#define debug_x_fired(x)          fprintf(stderr, "filter: Fire Rule %d\n", x)
#define debug_x_not_fired(x)      fprintf(stderr, "filter: Didn't Fire Rule %d\n"
#define debug_goal(x)             fprintf(stderr, "filter: Picking Goal %d\n", x+
#define debug_rule(x)             fprintf(stderr, "filter: Using Rule %d\n", x)
#define debug_rule_fired(x)       if (filter_node[x].value == 1.0)\
                                        fprintf(stderr, "filter: Fire Rule
                        else if (filter_node[x].value <= 0.0)\
                                      fprintf(stderr, "filter: Rule %d can't
                    else debug_and(x)
#define debug_rule_already_set(x) fprintf(stderr, "filter: Rule %d already set\n"
#define debug_afired(x)           fprintf(stderr, "filter: Row %d already fired\n"
#define debug_fired(x)            fprintf(stderr, "filter: Row %d fired\n", x+1)
#define debug_and(x)              fprintf(stderr, "filter: Rule %d given  value %
#define debug_a_and(x)            fprintf(stderr, "filter: Rule %d *kept* value %
#define debug_or(v,w,x)           fprintf(stderr, "filter: %d%c given  value %f a
#define debug_a_or(v,w,x)         fprintf(stderr, "filter: %d%c *kept* value %f a

#else

#define debug_x_fired(x)
#define debug_x_not_fired(x)
#define debug_goal(x)
#define debug_rule(x)
#define debug_rule_fired(x)
#define debug_rule_already_set(x)
#define debug_afired(x)
#define debug_fired(x)
#define debug_and(x)
#define debug_a_and(x)
#define debug_or(v,w,x)
#define debug_a_or(v,w,x)

#endif


float filter()
{
    filter_validate();
    filter_infer0();
    return(filter_node[0].value);
}

filter_call()
{
    filter_update_from_gfb();
    filter_validate();
    filter_infer0();
    filter_update_to_gfb();
    return(filter_node[0].value);
}


filter_infer0()
{
    debug_rule(0);
    while (
           !fired(filter_node[11].value)   ||
           !fired(filter_node[10].value)          )
```

```
    {
        G_filter_epsilon = FILTER_EPSILON;
        util_or_sort_children(filter_node, FILTER_CHILD0);
        switch (*FILTER_CHILD0)
        {
        case (11):
            if (filter_infer11())
                return(TRUE);
            break;
        case (10):
            if (filter_infer10())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
    }
    return(FALSE);
}


filter_infer1()
{
    float old_weight;
    float diff = 0;

    debug_rule(1);
    old_weight = util_and_weight(filter_node[1]);

    while (!fired(filter_node[1].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD1)
        {
        case (12):
            if (filter_infer12())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[1]) - old_weight;
    }
    return(FALSE);
}

filter_infer2()
{
    float old_weight;
    float diff = 0;

    debug_rule(2);
    old_weight = util_and_weight(filter_node[2]);

    while (!fired(filter_node[2].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD2)
        {
        case (15):
            if (filter_infer15())
                return(TRUE);
            break;
```

```
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[2]) - old_weight;
    }
    return(FALSE);
}

filter_infer3()
{
    float old_weight;
    float diff = 0;

    debug_rule(3);
    old_weight = util_and_weight(filter_node[3]);

    while (!fired(filter_node[3].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD3)
        {
        case (17):
            if (filter_infer17())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[3]) - old_weight;
    }
    return(FALSE);
}

filter_infer4()
{
    float old_weight;
    float diff = 0;

    debug_rule(4);
    old_weight = util_and_weight(filter_node[4]);

    while (!fired(filter_node[4].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD4)
        {
        case (16):
            if (filter_infer16())
                return(TRUE);
            break;
        case (18):
            if (filter_infer18())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[4]) - old_weight;
    }
    return(FALSE);
}

filter_infer5()
{
```

```
    float old_weight;
    float diff = 0;

    debug_rule(5);
    old_weight = util_and_weight(filter_node[5]);

    while (!fired(filter_node[5].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD5)
        {
        case (19):
            if (filter_infer19())
                return(TRUE);
            break;
        case (13):
            if (filter_infer13())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[5]) - old_weight;
    }
    return(FALSE);
}

filter_infer7()
{
    float old_weight;
    float diff = 0;

    debug_rule(7);
    old_weight = util_and_weight(filter_node[7]);

    while (!fired(filter_node[7].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD7)
        {
        case (21):
            if (filter_infer21())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[7]) - old_weight;
    }
    return(FALSE);
}

filter_infer8()
{
    float old_weight;
    float diff = 0;

    debug_rule(8);
    old_weight = util_and_weight(filter_node[8]);

    while (!fired(filter_node[8].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD8)
```

```
        {
        case (20):
            if (filter_infer20())
                return(TRUE);
            break;
        case (22):
            if (filter_infer22())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[8]) - old_weight;
    }
    return(FALSE);
}

filter_infer9()
{
    float old_weight;
    float diff = 0;

    debug_rule(9);
    old_weight = util_and_weight(filter_node[9]);

    while (!fired(filter_node[9].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD9)
        {
        case (14):
            if (filter_infer14())
                return(TRUE);
            break;
        case (22):
            if (filter_infer22())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(filter_node[9]) - old_weight;
    }
    return(FALSE);
}

filter_infer10()
{
    float old_weight;
    float diff = 0;

    debug_goal(0);
    old_weight = util_or_weight(filter_node[10]);

    while (!fired(filter_node[10].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD10)
        {
        case (5):
            if (filter_infer5())
                return(TRUE);
            break;
        case (7):
```

```
            if (filter_infer7())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(filter_node[10]) - old_weight;
    }
    return(FALSE);
}

filter_infer11()
{
    float old_weight;
    float diff = 0;

    debug_goal(0);
    old_weight = util_or_weight(filter_node[11]);

    while (!fired(filter_node[11].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD11)
        {
        case (1):
            if (filter_infer1())
                return(TRUE);
            break;
        case (8):
            if (filter_infer8())
                return(TRUE);
            break;
        case (9):
            if (filter_infer9())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(filter_node[11]) - old_weight;
    }
    return(FALSE);
}

filter_infer12()
{
    float value;

    debug_goal(2);

    value = is(shares * value < 2500);

    filter_node[12].value = value;
    debug_or(2,  'T', 12);
    filter_set12();
    if (filter_fire2())
        return(TRUE);
    else
    {
        filter_fire_goal2();
    }
    return(FALSE);
}
```

```
filter_infer13()
{
    float old_we
    float diff =

    debug_goal(4)
    old_weight =    _or_weight(filter_node[13]);

    while (!fired(  ter_node[13].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD13)
        {
        case (2):
            if (filter_infer2())
                return(TRUE);
            break;
        case (3):
            if (filter_infer3())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(filter_node[13]) - old_weight;
    }
    return(FALSE);
}

filter_infer14()
{
    float old_weight;
    float diff = 0;

    debug_goal(4);
    old_weight = util_or_weight(filter_node[14]);

    while (!fired(filter_node[14].value) &&
           (diff <= G_filter_epsilon &&  diff >= -G_filter_epsilon ))
    {
        switch (*FILTER_CHILD14)
        {
        case (4):
            if (filter_infer4())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(filter_node[14]) - old_weight;
    }
    return(FALSE);
}

filter_infer15()
{
    float value;

    debug_goal(5);

    value = is ((bid - ask) / last < .12);

    filter_node[15].value = value;
    debug_or(5,  'T', 15);
```

```
    filter_set15();
    if (filter_fire5())
        return(TRUE);
    else
    {
        filter_fire_goal5();
    }
    return(FALSE);
}

filter_infer16()
{
    float value;

    debug_goal(5);

    value = 1.0 - is ((bid - ask) / last < .12);

    filter_node[16].value = value;
    debug_or(5,  'F', 16);
    filter_set16();
    if (filter_fire5())
        return(TRUE);
    else
    {
        filter_fire_goal5();
    }
    return(FALSE);
}

filter_infer17()
{
    float value;

    debug_goal(6);

    value = is ((bid - ask) / last > .95);

    filter_node[17].value = value;
    debug_or(6,  'T', 17);
    filter_set17();
    if (filter_fire6())
        return(TRUE);
    else
    {
        filter_fire_goal6();
    }
    return(FALSE);
}

filter_infer18()
{
    float value;

    debug_goal(6);

    value = 1.0 - is ((bid - ask) / last > .95);

    filter_node[18].value = value;
    debug_or(6,  'F', 18);
    filter_set18();
    if (filter_fire6())
        return(TRUE);
    else
    {
                    al6();
```

```
    }
    return(FALSE)
}

filter_infer19()
{
    float value;

    debug_goal(8);

    value = is (bid / last > 1.05);

    filter_node[19].value = value;
    debug_or(8, 'T', 19);
    filter_set19();
    if (filter_fire8())
        return(TRUE);
    else
    {
        filter_fire_goal8();
    }
    return(FALSE);
}

filter_infer20()
{
    float value;

    debug_goal(8);

    value = 1.0 - is (bid / last > 1.05);

    filter_node[20].value = value;
    debug_or(8, 'F', 20);
    filter_set20();
    if (filter_fire8())
        return(TRUE);
    else
    {
        filter_fire_goal8();
    }
    return(FALSE);
}

filter_infer21()
{
    float value;

    debug_goal(10);

    filter_update_last();
    filter_update_to_gfb();
    value = czvolume_call();

    filter_node[21].value = value;
    debug_or(10, 'T', 21);
    filter_set21();
    if (filter_fire10())
        return(TRUE);
    else
    {
        filter_update_from_gfb();
        filter_fire_goal10();
    }
    return(FA
}
```

```
filter_infer22()
{
    float value;

    debug_goal(10);

    filter_update_last();
    filter_update_to_gfb();
    value = 1.0 - czvolume_call();

    filter_node[22].value = value;
    debug_or(10,  'F', 22);
    filter_set22();
    if (filter_fire10())
        return(TRUE);
    else
    {
        filter_update_from_gfb();
        filter_fire_goal10();
    }
    return(FALSE);
}

filter_set10()
{
    if (filter_node[10].value != CANT_ANSWER)
    {
        filter_node[11].value = 1.0 - filter_node[10].value;
        debug_or(0, 'F',  11);
    }
    else
        filter_node[11].value = CANT_ANSWER;
}

filter_set11()
{
    if (filter_node[11].value != CANT_ANSWER)
    {
        filter_node[10].value = 1.0 - filter_node[11].value;
        debug_or(0, 'T',  10);
    }
    else
        filter_node[10].value = CANT_ANSWER;
}

filter_set12()
{
}

filter_set13()
{
    if (filter_node[13].value != CANT_ANSWER)
    {
        filter_node[14].value = 1.0 - filter_node[13].value;
        debug_or(4, 'F',  14);
    }
    else
        filter_node[14].value = CANT_ANSWER;
}

filter_set14()
{
    if (filter_node[14].value != CANT_ANSWER)
    {
        filter_node[13].value = 1.0 - filter_node[14].value;
```

```
        debug_or(4, 'T',  13);
    }
    else
        filter_node[13].value = CANT_ANSWER;
}

filter_set15()
{
    if (filter_node[15].value != CANT_ANSWER)
    {
        filter_node[16].value = 1.0 - filter_node[15].value;
        debug_or(5, 'F',  16);
    }
    else
        filter_node[16].value = CANT_ANSWER;
}

filter_set16()
{
    if (filter_node[16].value != CANT_ANSWER)
    {
        filter_node[15].value = 1.0 - filter_node[16].value;
        debug_or(5, 'T',  15);
    }
    else
        filter_node[15].value = CANT_ANSWER;
}

filter_set17()
{
    if (filter_node[17].value != CANT_ANSWER)
    {
        filter_node[18].value = 1.0 - filter_node[17].value;
        debug_or(6, 'F',  18);
    }
    else
        filter_node[18].value = CANT_ANSWER;
}

filter_set18()
{
    if (filter_node[18].value != CANT_ANSWER)
    {
        filter_node[17].value = 1.0 - filter_node[18].value;
        debug_or(6, 'T',  17);
    }
    else
        filter_node[17].value = CANT_ANSWER;
}

filter_set19()
{
    if (filter_node[19].value != CANT_ANSWER)
    {
        filter_node[20].value = 1.0 - filter_node[19].value;
        debug_or(8, 'F',  20);
    }
    else
        filter_node[20].value = CANT_ANSWER;
}

filter_set20()
{
    if (filter_node[20].value != CANT_ANSWER)
    {
        filter_node[19].value = 1.0 - filter_node[20].value;
```

```
        debug_or(8, 'T',  19);
    }
    else
        filter_node[19].value = CANT_ANSWER;
}

filter_set21()
{
    if (filter_node[21].value != CANT_ANSWER)
    {
        filter_node[22].value = 1.0 - filter_node[21].value;
        debug_or(10, 'F',  22);
    }
    else
        filter_node[22].value = CANT_ANSWER;
}

filter_set22()
{
    if (filter_node[22].value != CANT_ANSWER)
    {
        filter_node[21].value = 1.0 - filter_node[22].value;
        debug_or(10, 'T',  21);
    }
    else
        filter_node[21].value = CANT_ANSWER;
}

filter_fire_goal0()
{
    if (filter_node[10].value == 1.0)
        filter_node[0].value = 1.0;
    else if (filter_node[11].value == 1.0)
        filter_node[0].value = 0.0;
    else filter_node[0].value = CANT_ANSWER;
}

filter_fire_goal2()
{
    short stat = 0;
    stat += filter_fire_rule1();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 11, FILTER_CHILD11);
        debug_or(0, 'F', 11);
        if (stat == TRUE)
            filter_set11();

        filter_fire_goal0();
    }
}

filter_fire_goal4()
{
    short stat = 0;
    stat += filter_fire_rule9();
    stat += filter_fire_rule5();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 11, FILTER_CHILD11);
        debug_or(0, 'F', 11);
        if (stat == TRUE)
            filter_set11();

        stat = util_or_update(filter_node, 10, FILTER_CHILD10);
        debug_or(0, 'T', 10);
```

```
        if (stat == TRUE)
            filter_set10();

        filter_fire_goal0();
    }
}

filter_fire_goal5()
{
    short stat = 0;
    G_filter_epsilon = 0.0;
    stat += filter_fire_rule4();
    stat += filter_fire_rule2();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 14, FILTER_CHILD14);
        debug_or(4, 'F', 14);
        if (stat == TRUE)
            filter_set14();

        stat = util_or_update(filter_node, 13, FILTER_CHILD13);
        debug_or(4, 'T', 13);
        if (stat == TRUE)
            filter_set13();

        filter_fire_goal4();
    }
}

filter_fire_goal6()
{
    short stat = 0;
    G_filter_epsilon = 0.0;
    stat += filter_fire_rule4();
    stat += filter_fire_rule3();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 14, FILTER_CHILD14);
        debug_or(4, 'F', 14);
        if (stat == TRUE)
            filter_set14();

        stat = util_or_update(filter_node, 13, FILTER_CHILD13);
        debug_or(4, 'T', 13);
        if (stat == TRUE)
            filter_set13();

        filter_fire_goal4();
    }
}

filter_fire_goal8()
{
    short stat = 0;
    stat += filter_fire_rule8();
    stat += filter_fire_rule5();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 11, FILTER_CHILD11);
        debug_or(0, 'F', 11);
        if (stat == TRUE)
            filter_set11();

        stat = util_or_update(filter_node, 10, FILTER_CHILD10);
        debug_or(0, 'T', 10);
        if (stat == TRUE)
```

```
            filter_set10();

        filter_fire_goal0();
    }
}

filter_fire_goal10()
{
    short stat = 0;
    stat += filter_fire_rule9();
    stat += filter_fire_rule8();
    stat += filter_fire_rule7();
    if (stat > 0)
    {
        stat = util_or_update(filter_node, 11, FILTER_CHILD11);
        debug_or(0, 'F', 11);
        if (stat == TRUE)
            filter_set11();

        stat = util_or_update(filter_node, 10, FILTER_CHILD10);
        debug_or(0, 'T', 10);
        if (stat == TRUE)
            filter_set10();

        filter_fire_goal0();
    }
}

filter_fire_rule1()
{
    if (!fired(filter_node[1].value))
    {
        util_and_update(filter_node, 1, FILTER_CHILD1);
        debug_rule_fired(1);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(1);
        return(FALSE);
    }
}

filter_fire_rule2()
{
    if (!fired(filter_node[2].value))
    {
        util_and_update(filter_node, 2, FILTER_CHILD2);
        debug_rule_fired(2);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(2);
        return(FALSE);
    }
}

filter_fire_rule3()
{
    if (!fired(filter_node[3].value))
    {
        util_and_update(filter_node, 3, FILTER_CHILD3);
        debug_rule_fired(3);
        return(TRUE);
    }
```

```
    else
    {
        debug_rule_already_set(3);
        return(FALSE);
    }
}

filter_fire_rule4()
{
    if (!fired(filter_node[4].value))
    {
        util_and_update(filter_node, 4, FILTER_CHILD4);
        debug_rule_fired(4);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(4);
        return(FALSE);
    }
}

filter_fire_rule5()
{
    if (!fired(filter_node[5].value))
    {
        util_and_update(filter_node, 5, FILTER_CHILD5);
        debug_rule_fired(5);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(5);
        return(FALSE);
    }
}

filter_fire_rule7()
{
    if (!fired(filter_node[7].value))
    {
        util_and_update(filter_node, 7, FILTER_CHILD7);
        debug_rule_fired(7);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(7);
        return(FALSE);
    }
}

filter_fire_rule8()
{
    if (!fired(filter_node[8].value))
    {
        util_and_update(filter_node, 8, FILTER_CHILD8);
        debug_rule_fired(8);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(8);
        return(FALSE);
    }
}
```

```
filter_fire_rule9()
{
    if (!fired(filter_node[9].value))
    {
        util_and_update(filter_node, 9, FILTER_CHILD9);
        debug_rule_fired(9);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(9);
        return(FALSE);
    }
}

filter_fire0()
{
    return(FALSE);
}

filter_fire2()
{
    return(FALSE);
}

filter_fire4()
{
    return(FALSE);
}

filter_fire5()
{
    return(FALSE);
}

filter_fire6()
{
    return(FALSE);
}

filter_fire8()
{
    return(FALSE);
}

filter_fire10()
{
    return(FALSE);
}
filter_update_last()
{
    short i;
    for ( i=0; i < FILTER_NODES; i++)
        filter_node[i].last = filter_node[i].value;
    return(SUCCESS);
}

filter_update_from_gfb()
{
    short ret = FALSE;

    if (G_gfb_changed == FALSE)
        return(ret);

                        && filter_node[10].value != gfb[1])
```

```
    {
        ret = TRUE;
        filter_node[10].value = gfb[1];
        debug_or(0, 'T', 10);
        filter_node[11].value = 1.0 - gfb[1];
        debug_or(0, 'F', 11);
    }
    if (fired(gfb[2]) && filter_node[12].value != gfb[2])
    {
        ret = TRUE;
        filter_node[12].value = gfb[2];
        debug_or(2, 'T', 12);
    }
    if (fired(gfb[6]) && filter_node[13].value != gfb[6])
    {
        ret = TRUE;
        filter_node[13].value = gfb[6];
        debug_or(4, 'T', 13);
        filter_node[14].value = 1.0 - gfb[6];
        debug_or(4, 'F', 14);
    }
    if (fired(gfb[4]) && filter_node[15].value != gfb[4])
    {
        ret = TRUE;
        filter_node[15].value = gfb[4];
        debug_or(5, 'T', 15);
        filter_node[16].value = 1.0 - gfb[4];
        debug_or(5, 'F', 16);
    }
    if (fired(gfb[5]) && filter_node[17].value != gfb[5])
    {
        ret = TRUE;
        filter_node[17].value = gfb[5];
        debug_or(6, 'T', 17);
        filter_node[18].value = 1.0 - gfb[5];
        debug_or(6, 'F', 18);
    }
    if (fired(gfb[7]) && filter_node[19].value != gfb[7])
    {
        ret = TRUE;
        filter_node[19].value = gfb[7];
        debug_or(8, 'T', 19);
        filter_node[20].value = 1.0 - gfb[7];
        debug_or(8, 'F', 20);
    }
    if (fired(gfb[8]) && filter_node[21].value != gfb[8])
    {
        ret = TRUE;
        filter_node[21].value = gfb[8];
        debug_or(10, 'T', 21);
        filter_node[22].value = 1.0 - gfb[8];
        debug_or(10, 'F', 22);
    }
    return(ret);
}

filter_update_to_gfb()
{
    short ret = FALSE;

    if (torf(filter_node[10].value) &&
        gfb[1] != filter_node[10].value)
    {
        ret = TRUE;
        gfb[1] = filter_node[10].value;
    }
```

```
    if (torf(filter_node[12].value) &&
        gfb[2] != filter_node[12].value)
    {
        ret = TRUE;
        gfb[2] = filter_node[12].value;
    }
    if (torf(filter_node[13].value) &&
        gfb[6] != filter_node[13].value)
    {
        ret = TRUE;
        gfb[6] = filter_node[13].value;
    }
    if (torf(filter_node[15].value) &&
        gfb[4] != filter_node[15].value)
    {
        ret = TRUE;
        gfb[4] = filter_node[15].value;
    }
    if (torf(filter_node[17].value) &&
        gfb[5] != filter_node[17].value)
    {
        ret = TRUE;
        gfb[5] = filter_node[17].value;
    }
    if (torf(filter_node[19].value) &&
        gfb[7] != filter_node[19].value)
    {
        ret = TRUE;
        gfb[7] = filter_node[19].value;
    }
    if (torf(filter_node[21].value) &&
        gfb[8] != filter_node[21].value)
    {
        ret = TRUE;
        gfb[8] = filter_node[21].value;
    }

    if (G_gfb_changed == FALSE)
        G_gfb_changed = ret;

    return(ret);
}

filter_validate()
{
    while (filter_val_trav0() == TRUE)
        continue;
    util_or_value(filter_node, 0, FILTER_CHILD0);

    return(TRUE);
}


filter_val_trav0()
{
    short ret = FALSE;
    short status10;
    short status8;
    short status4;
    short status2;

    if (fired(filter_node[10].value))
    {
        if (filter_node[10].value != filter_node[10].last)
        {
            filter_node[10].last = filter_node[10].value;
```

```
            _node[11].last = filter_node[11].value;
            fired(0);
            TRUE;
        }
        else
        {
            FALSE;
            _afired(0);
        }
    }
    else
    {
        status10 = filter_val_trav10();
        status4 = filter_val_trav4();

        if (status10 == TRUE ||
            status4 == TRUE)
        {
            util_and_update(filter_node, 9, FILTER_CHILD9);
            filter_node[9].last = filter_node[9].value;
            debug_and(9);
        }
        else
            debug_a_and(9);

        status8 = filter_val_trav8();

        if (status10 == TRUE ||
            status8 == TRUE)
        {
            util_and_update(filter_node, 8, FILTER_CHILD8);
            filter_node[8].last = filter_node[8].value;
            debug_and(8);
        }
        else
            debug_a_and(8);


        if (status10 == TRUE)
        {
            util_and_update(filter_node, 7, FILTER_CHILD7);
            filter_node[7].last = filter_node[7].value;
            debug_and(7);
        }
        else
            debug_a_and(7);


        if (status8 == TRUE ||
            status4 == TRUE)
        {
            util_and_update(filter_node, 5, FILTER_CHILD5);
            filter_node[5].last = filter_node[5].value;
            debug_and(5);
        }
        else
            debug_a_and(5);

        status2 = filter_val_trav2();

        if (sta      == TRUE)
        {
            u        _update(filter_node, 1, FILTER_CHILD1);
            f        ode[1].last = filter_node[1].value;
            d(1);
        }
```

```
        else
            debug_a_and(1);


        if (status10 == TRUE ||
            status8 == TRUE ||
            status4 == TRUE ||
            status2 == TRUE)
        {
            util_or_update(filter_node, 11, FILTER_CHILD11);
            filter_node[11].last = filter_node[11].value;
            ret = TRUE;
            debug_or(0, 'F', 11);
        }
        else
            debug_a_or(0, 'F', 11);


        if (status10 == TRUE ||
            status8 == TRUE ||
            status4 == TRUE)
        {
            util_or_update(filter_node, 10, FILTER_CHILD10);
            filter_node[10].last = filter_node[10].value;
            ret = TRUE;
            debug_or(0, 'T', 10);
        }
        else
            debug_a_or(0, 'T', 10);

    }

    return(ret);
}

filter_val_trav2()
{
    short ret = FALSE;

    if (filter_node[12].value != filter_node[12].last)
    {
        filter_node[12].last = filter_node[12].value;
        ret = TRUE;
    }

    return(ret);
}

filter_val_trav4()
{
    short ret = FALSE;
    short status6;
    short status5;

    if (fired(filter_node[13].value))
    {
        if (filter_node[13].value != filter_node[13].last)
        {
            filter_node[13].last = filter_node[13].value;
            filter_node[14].last = filter_node[14].value;
            debug_fired(4);
            ret = TRUE;
        }
        else
        {
            ret = FALSE;
```

```
            debug_afired(4);
        }
    }
    else
    {
        status6 = filter_val_trav6();
        status5 = filter_val_trav5();

        if (status6 == TRUE ||
            status5 == TRUE)
        {
            util_and_update(filter_node, 4, FILTER_CHILD4);
            filter_node[4].last = filter_node[4].value;
            debug_and(4);
        }
        else
            debug_a_and(4);


        if (status6 == TRUE)
        {
            util_and_update(filter_node, 3, FILTER_CHILD3);
            filter_node[3].last = filter_node[3].value;
            debug_and(3);
        }
        else
            debug_a_and(3);

        status5 = filter_val_trav5();

        if (status5 == TRUE)
        {
            util_and_update(filter_node, 2, FILTER_CHILD2);
            filter_node[2].last = filter_node[2].value;
            debug_and(2);
        }
        else
            debug_a_and(2);


        if (status6 == TRUE ||
            status5 == TRUE)
        {
            util_or_update(filter_node, 14, FILTER_CHILD14);
            filter_node[14].last = filter_node[14].value;
            ret = TRUE;
            debug_or(4, 'F', 14);
        }
        else
            debug_a_or(4, 'F', 14);


        if (status6 == TRUE ||
            status5 == TRUE)
        {
            util_or_update(filter_node, 13, FILTER_CHILD13);
            filter_node[13].last = filter_node[13].value;
            ret = TRUE;
            debug_or(4, 'T', 13);
        }
        else
            debug_a_or(4, 'T', 13);

    }

    return(ret);
```

```
}

filter_val_trav5()
{
    short ret = FALSE;

    if (filter_node[15].value != filter_node[15].last ||
        filter_node[16].value != filter_node[16].last)
    {
        filter_node[15].last = filter_node[15].value;
        filter_node[16].last = filter_node[16].value;
        ret = TRUE;
    }

    return(ret);
}

filter_val_trav6()
{
    short ret = FALSE;

    if (filter_node[17].value != filter_node[17].last ||
        filter_node[18].value != filter_node[18].last)
    {
        filter_node[17].last = filter_node[17].value;
        filter_node[18].last = filter_node[18].value;
        ret = TRUE;
    }

    return(ret);
}

filter_val_trav8()
{
    short ret = FALSE;

    if (filter_node[19].value != filter_node[19].last ||
        filter_node[20].value != filter_node[20].last)
    {
        filter_node[19].last = filter_node[19].value;
        filter_node[20].last = filter_node[20].value;
        ret = TRUE;
    }

    return(ret);
}

filter_val_trav10()
{
    short ret = FALSE;

    if (filter_node[21].value != filter_node[21].last ||
        filter_node[22].value != filter_node[22].last)
    {
        filter_node[21].last = filter_node[21].value;
        filter_node[22].last = filter_node[22].value;
        ret = TRUE;
    }

    return(ret);
}

filter_reset()
{
    G_filter_epsilon = FILTER_EPSILON;
```

```
    filter_pt[1] = 11;
    filter_pt[2] = 10;
    filter_pt[10] = 16;
    filter_pt[11] = 18;
    filter_pt[13] = 19;
    filter_pt[14] = 13;
    filter_pt[19] = 20;
    filter_pt[20] = 22;
    filter_pt[22] = 14;
    filter_pt[23] = 22;
    filter_pt[25] = 5;
    filter_pt[26] = 7;
    filter_pt[28] = 1;
    filter_pt[29] = 8;
    filter_pt[30] = 9;
    filter_pt[33] = 2;
    filter_pt[34] = 3;

    filter_node[0].cost = FILTER_ICOST0;
    filter_node[0].value = FILTER_IVAL0;
    filter_node[0].last = FILTER_IVAL0;
    filter_node[1].cost = FILTER_ICOST1;
    filter_node[1].value = FILTER_IVAL1;
    filter_node[1].last = FILTER_IVAL1;
    filter_node[2].cost = FILTER_ICOST2;
    filter_node[2].value = FILTER_IVAL2;
    filter_node[2].last = FILTER_IVAL2;
    filter_node[3].cost = FILTER_ICOST3;
    filter_node[3].value = FILTER_IVAL3;
    filter_node[3].last = FILTER_IVAL3;
    filter_node[4].cost = FILTER_ICOST4;
    filter_node[4].value = FILTER_IVAL4;
    filter_node[4].last = FILTER_IVAL4;
    filter_node[5].cost = FILTER_ICOST5;
    filter_node[5].value = FILTER_IVAL5;
    filter_node[5].last = FILTER_IVAL5;
    filter_node[6].cost = FILTER_ICOST6;
    filter_node[6].value = FILTER_IVAL6;
    filter_node[6].last = FILTER_IVAL6;
    filter_node[7].cost = FILTER_ICOST7;
    filter_node[7].value = FILTER_IVAL7;
    filter_node[7].last = FILTER_IVAL7;
    filter_node[8].cost = FILTER_ICOST8;
    filter_node[8].value = FILTER_IVAL8;
    filter_node[8].last = FILTER_IVAL8;
    filter_node[9].cost = FILTER_ICOST9;
    filter_node[9].value = FILTER_IVAL9;
    filter_node[9].last = FILTER_IVAL9;
    filter_node[10].cost = FILTER_ICOST10;
    filter_node[10].value = FILTER_IVAL10;
    filter_node[10].last = FILTER_IVAL10;
    filter_node[11].cost = FILTER_ICOST11;
    filter_node[11].value = FILTER_IVAL11;
    filter_node[11].last = FILTER_IVAL11;
    filter_node[12].cost = FILTER_ICOST12;
    filter_node[12].value = FILTER_IVAL12;
    filter_node[12].last = FILTER_IVAL12;
    filter_node[13].cost = FILTER_ICOST13;
    filter_node[13].value = FILTER_IVAL13;
    filter_node[13].last = FILTER_IVAL13;
    filter_node[14].cost = FILTER_ICOST14;
    filter_node[14].value = FILTER_IVAL14;
    filter_node[14].last = FILTER_IVAL14;
    filter_node[15].cost = FILTER_ICOST15;
    filter_node[15].value = FILTER_IVAL15;
    filter_node[15].last = FILTER_IVAL15;
```

```
    filter_node[16].cost = FILTER_ICOST16;
    filter_node[16].value = FILTER_IVAL16;
    filter_node[16].last = FILTER_IVAL16;
    filter_node[17].cost = FILTER_ICOST17;
    filter_node[17].value = FILTER_IVAL17;
    filter_node[17].last = FILTER_IVAL17;
    filter_node[18].cost = FILTER_ICOST18;
    filter_node[18].value = FILTER_IVAL18;
    filter_node[18].last = FILTER_IVAL18;
    filter_node[19].cost = FILTER_ICOST19;
    filter_node[19].value = FILTER_IVAL19;
    filter_node[19].last = FILTER_IVAL19;
    filter_node[20].cost = FILTER_ICOST20;
    filter_node[20].value = FILTER_IVAL20;
    filter_node[20].last = FILTER_IVAL20;
    filter_node[21].cost = FILTER_ICOST21;
    filter_node[21].value = FILTER_IVAL21;
    filter_node[21].last = FILTER_IVAL21;
    filter_node[22].cost = FILTER_ICOST22;
    filter_node[22].value = FILTER_IVAL22;
    filter_node[22].last = FILTER_IVAL22;
}
```

Appendix I. Header file "czvolume.h" generated by the compiler

```
#ifndef CZVOLUME_H
#define CZVOLUME_H  1
#include "util.h"
#include "sample.h"

#define CZVOLUME_NODES 19
#define CZVOLUME_EPSILON 400.000000


#define CZVOLUME_ICOST0         0
#define CZVOLUME_IVAL0  0.500000
#define CZVOLUME_GFB0   -1
#define CZVOLUME_BOOL0  GFB_UNKNOWN
#define CZVOLUME_CHILD0         (czvolume_pt + 1)

#define CZVOLUME_ICOST1         1
#define CZVOLUME_IVAL1  1.000000
#define CZVOLUME_GFB1   -1
#define CZVOLUME_BOOL1  GFB_UNKNOWN
#define CZVOLUME_CHILD1         (czvolume_pt + 3)

#define CZVOLUME_ICOST2         1
#define CZVOLUME_IVAL2  1.000000
#define CZVOLUME_GFB2   -1
#define CZVOLUME_BOOL2  GFB_UNKNOWN
#define CZVOLUME_CHILD2         (czvolume_pt + 4)

#define CZVOLUME_ICOST3         110
#define CZVOLUME_IVAL3  0.020000
#define CZVOLUME_GFB3   -1
#define CZVOLUME_BOOL3  GFB_UNKNOWN
#define CZVOLUME_CHILD3         (czvolume_pt + 5)

#define CZVOLUME_ICOST4         110
#define CZVOLUME_IVAL4  0.020000
#define CZVOLUME_GFB4   -1
#define CZVOLUME_BOOL4  GFB_UNKNOWN
#define CZVOLUME_CHILD4         (czvolume_pt + 8)

#define CZVOLUME_ICOST5         120
#define CZVOLUME_IVAL5  0.180000
#define CZVOLUME_GFB5   -1
#define CZVOLUME_BOOL5  GFB_UNKNOWN
#define CZVOLUME_CHILD5         (czvolume_pt + 11)

#define CZVOLUME_ICOST6         120
#define CZVOLUME_IVAL6  0.160000
#define CZVOLUME_GFB6   -1
#define CZVOLUME_BOOL6  GFB_UNKNOWN
#define CZVOLUME_CHILD6         (czvolume_pt + 14)

#define CZVOLUME_ICOST7         368
#define CZVOLUME_IVAL7  0.338476
#define CZVOLUME_GFB7   3
#define CZVOLUME_BOOL7  GFB_TRUE
#define CZVOLUME_CHILD7         (czvolume_pt + 17)

#define CZVOLUME_ICOST8         100
#define CZVOLUME_IVAL8  0.800000
#define CZVOLUME_GFB8   15
#define CZVOLUME_BOOL8  GFB_TRUE
#define CZVOLUME_CHILD8         (czvolume_pt + 22)

#define CZVOLUME_ICOST9         1
#define CZVOLUME_IVAL9  0.500000
#define CZVOLUME_GFB9   15
```

```
#define CZVOLUME_BOOL9  GFB_X
#define CZVOLUME_CHILD9         (czvolume_pt + 23)

#define CZVOLUME_ICOST10        100
#define CZVOLUME_IVAL10         0.100000
#define CZVOLUME_GFB10  9
#define CZVOLUME_BOOL10         GFB_TRUE
#define CZVOLUME_CHILD10        (czvolume_pt + 26)

#define CZVOLUME_ICOST11        100
#define CZVOLUME_IVAL11         0.900000
#define CZVOLUME_GFB11  9
#define CZVOLUME_BOOL11         GFB_FALSE
#define CZVOLUME_CHILD11        (czvolume_pt + 27)

#define CZVOLUME_ICOST12        1
#define CZVOLUME_IVAL12         0.500000
#define CZVOLUME_GFB12  9
#define CZVOLUME_BOOL12         GFB_X
#define CZVOLUME_CHILD12        (czvolume_pt + 28)

#define CZVOLUME_ICOST13        100
#define CZVOLUME_IVAL13         0.100000
#define CZVOLUME_GFB13  10
#define CZVOLUME_BOOL13         GFB_TRUE
#define CZVOLUME_CHILD13        (czvolume_pt + 30)

#define CZVOLUME_ICOST14        1
#define CZVOLUME_IVAL14         0.500000
#define CZVOLUME_GFB14  10
#define CZVOLUME_BOOL14         GFB_X
#define CZVOLUME_CHILD14        (czvolume_pt + 31)

#define CZVOLUME_ICOST15        100
#define CZVOLUME_IVAL15         0.200000
#define CZVOLUME_GFB15  11
#define CZVOLUME_BOOL15         GFB_TRUE
#define CZVOLUME_CHILD15        (czvolume_pt + 33)

#define CZVOLUME_ICOST16        100
#define CZVOLUME_IVAL16         0.200000
#define CZVOLUME_GFB16  12
#define CZVOLUME_BOOL16         GFB_TRUE
#define CZVOLUME_CHILD16        (czvolume_pt + 34)

#define CZVOLUME_ICOST17        100
#define CZVOLUME_IVAL17         0.200000
#define CZVOLUME_GFB17  13
#define CZVOLUME_BOOL17         GFB_TRUE
#define CZVOLUME_CHILD17        (czvolume_pt + 35)

#define CZVOLUME_ICOST18        100
#define CZVOLUME_IVAL18         0.200000
#define CZVOLUME_GFB18  14
#define CZVOLUME_BOOL18         GFB_TRUE
#define CZVOLUME_CHILD18        (czvolume_pt + 36)

#ifdef CZVOLUME_C


short czvolume_pt[] = {0,7,0,0,0,13,15,0,10,16,0,17,11,0,18,8,0,5,6,3,4,0,0,1,2,0


NODE czvolume_node[CZVOLUME_NODES] = {
    {CZVOLUME_ICOST0, CZVOLUME_IVAL0, CZVOLUME_IVAL0},
    {CZVOLUME_ICOST1, CZVOLUME_IVAL1, CZVOLUME_IVAL1},
```

```
    {CZVOLUME_ICOST2, CZVOLUME_IVAL2, CZVOLUME_IVAL2},
    {CZVOLUME_ICOST3, CZVOLUME_IVAL3, CZVOLUME_IVAL3},
    {CZVOLUME_ICOST4, CZVOLUME_IVAL4, CZVOLUME_IVAL4},
    {CZVOLUME_ICOST5, CZVOLUME_IVAL5, CZVOLUME_IVAL5},
    {CZVOLUME_ICOST6, CZVOLUME_IVAL6, CZVOLUME_IVAL6},
    {CZVOLUME_ICOST7, CZVOLUME_IVAL7, CZVOLUME_IVAL7},
    {CZVOLUME_ICOST8, CZVOLUME_IVAL8, CZVOLUME_IVAL8},
    {CZVOLUME_ICOST9, CZVOLUME_IVAL9, CZVOLUME_IVAL9},
    {CZVOLUME_ICOST10, CZVOLUME_IVAL10, CZVOLUME_IVAL10},
    {CZVOLUME_ICOST11, CZVOLUME_IVAL11, CZVOLUME_IVAL11},
    {CZVOLUME_ICOST12, CZVOLUME_IVAL12, CZVOLUME_IVAL12},
    {CZVOLUME_ICOST13, CZVOLUME_IVAL13, CZVOLUME_IVAL13},
    {CZVOLUME_ICOST14, CZVOLUME_IVAL14, CZVOLUME_IVAL14},
    {CZVOLUME_ICOST15, CZVOLUME_IVAL15, CZVOLUME_IVAL15},
    {CZVOLUME_ICOST16, CZVOLUME_IVAL16, CZVOLUME_IVAL16},
    {CZVOLUME_ICOST17, CZVOLUME_IVAL17, CZVOLUME_IVAL17},
    {CZVOLUME_ICOST18, CZVOLUME_IVAL18, CZVOLUME_IVAL18},
};


float G_czvolume_epsilon =  CZVOLUME_EPSILON;

#else

extern short *czvolume_pt;
extern NODE czvolume_node[CZVOLUME_NODES];
extern float G_czvolume_epsilon;

extern float czvolume();
#endif CZVOLUME_C
#endif CZVOLUME_H
```

Appendix J. Source file "czvolume.c" generated by the compiler

```
#define CZVOLUME_C 1
#include "sample.h"
#include "czvolume.h"
#include "general.h"
#include "ask.h"

#ifdef DEBUG

#define debug_x_fired(x)          fprintf(stderr, "czvolume: Fire Rule %d\n", x)
#define debug_x_not_fired(x)      fprintf(stderr, "czvolume: Didn't Fire Rule %d\
#define debug_goal(x)             fprintf(stderr, "czvolume: Picking Goal %d\n",
#define debug_rule(x)             fprintf(stderr, "czvolume: Using Rule %d\n", x)
#define debug_rule_fired(x)       if (czvolume_node[x].value == 1.0)\
                                      fprintf(stderr, "czvolume: Fire Ru
                      else if (czvolume_node[x].value <= 0.0)\
                                    fprintf(stderr, "czvolume: Rule %d can'
                  else debug_and(x)
#define debug_rule_already_set(x) fprintf(stderr, "czvolume: Rule %d already set\
#define debug_afired(x)           fprintf(stderr, "czvolume: Row %d already fired
#define debug_fired(x)            fprintf(stderr, "czvolume: Row %d fired\n", x+1
#define debug_and(x)              fprintf(stderr, "czvolume: Rule %d given  value
#define debug_a_and(x)            fprintf(stderr, "czvolume: Rule %d *kept* value
#define debug_or(v,w,x)           fprintf(stderr, "czvolume: %d%c given  value %f
#define debug_a_or(v,w,x)         fprintf(stderr, "czvolume: %d%c *kept* value %f

#else

#define debug_x_fired(x)
#define debug_x_not_fired(x)
#define debug_goal(x)
#define debug_rule(x)
#define debug_rule_fired(x)
#define debug_rule_already_set(x)
#define debug_afired(x)
#define debug_fired(x)
#define debug_and(x)
#define debug_a_and(x)
#define debug_or(v,w,x)
#define debug_a_or(v,w,x)

#endif


float czvolume()
{
    czvolume_validate();
    czvolume_infer0();
    return(czvolume_node[0].value);
}

czvolume_call()
{
    czvolume_update_from_gfb();
    czvolume_validate();
    czvolume_infer0();
    czvolume_update_to_gfb();
    return(czvolume_node[0].value);
}


czvolume_infer0()
{
    debug_rule(0);
    while (
           !fir   czvolume_node[7].value)          )
    {
```

```
        G_czvolume_epsilon = CZVOLUME_EPSILON;
        util_or_sort_children(czvolume_node, CZVOLUME_CHILD0);
        switch (*CZVOLUME_CHILD0)
        {
        case (7):
            if (czvolume_infer7())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
    }
    return(FALSE);
}


czvolume_infer1()
{
}

czvolume_infer2()
{
}

czvolume_infer3()
{
    float old_weight;
    float diff = 0;

    debug_rule(3);
    old_weight = util_and_weight(czvolume_node[3]);

    while (!fired(czvolume_node[3].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD3)
        {
        case (13):
            if (czvolume_infer13())
                return(TRUE);
            break;
        case (15):
            if (czvolume_infer15())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(czvolume_node[3]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer4()
{
    float old_weight;
    float diff = 0;

    debug_rule(4);
    old_weight = util_and_weight(czvolume_node[4]);

    while (!fired(czvolume_node[4].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
```

```
        switch          ME_CHILD4)
        {
        case (
            if        me_infer10())
                      (TRUE);
            bre
        case (16
            if (c.    ume_infer16())
                r     n(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(czvolume_node[4]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer5()
{
    float old_weight;
    float diff = 0;

    debug_rule(5);
    old_weight = util_and_weight(czvolume_node[5]);

    while (!fired(czvolume_node[5].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD5)
        {
        case (17):
            if (czvolume_infer17())
                return(TRUE);
            break;
        case (11):
            if (czvolume_infer11())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(czvolume_node[5]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer6()
{
    float old_weight;
    float diff = 0;

    debug_rule(6);
    old_weight = util_and_weight(czvolume_node[6]);

    while (!fired(czvolume_node[6].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD6)
        {
        case (18):
            if (czvolume_infer18())
                turn(TRUE);
            bre
```

```
        case (8):
            if (czvolume_infer8())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_and_weight(czvolume_node[6]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer7()
{
    float old_weight;
    float diff = 0;

    debug_goal(0);
    old_weight = util_or_weight(czvolume_node[7]);

    while (!fired(czvolume_node[7].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD7)
        {
        case (5):
            if (czvolume_infer5())
                return(TRUE);
            break;
        case (6):
            if (czvolume_infer6())
                return(TRUE);
            break;
        case (3):
            if (czvolume_infer3())
                return(TRUE);
            break;
        case (4):
            if (czvolume_infer4())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(czvolume_node[7]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer8()
{
    float value;

    debug_goal(2);

    value = is (volume > 25000 OR volume < 75000);

    czvolume_node[8].value = value;
    debug_or(2,  'T', 8);
    czvolume_set8();
    if (czvolume_fire2())
        return(TRUE);
    else
    {
```

```
        czvolume_fire_goal2();
    }
    return(FALSE);
}

czvolume_infer9()
{
    float old_weight;
    float diff = 0;

    debug_goal(2);
    old_weight = util_or_weight(czvolume_node[9]);

    while (!fired(czvolume_node[9].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD9)
        {
        case (1):
            if (czvolume_infer1())
                return(TRUE);
            break;
        case (2):
            if (czvolume_infer2())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(czvolume_node[9]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer10()
{
    float value;

    debug_goal(3);

    value = is (volume >= 75000);

    czvolume_node[10].value = value;
    debug_or(3,  'T', 10);
    czvolume_set10();
    if (czvolume_fire3())
        return(TRUE);
    else
    {
        czvolume_fire_goal3();
    }
    return(FALSE);
}

czvolume_infer11()
{
    float value;

    debug_goal(3);

    value = 1.0 - is (volume >= 75000);

    czvolume_node[11].value = value;
    debug_or(3,  'F', 11);
    czvolume_set11();
```

```
    if (czvolume_fire3())
        return(TRUE);
    else
    {
        czvolume_fire_goal3();
    }
    return(FALSE);
}

czvolume_infer12()
{
    float old_weight;
    float diff = 0;

    debug_goal(3);
    old_weight = util_or_weight(czvolume_node[12]);

    while (!fired(czvolume_node[12].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
        switch (*CZVOLUME_CHILD12)
        {
        case (1):
            if (czvolume_infer1())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(czvolume_node[12]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer13()
{
    float value;

    debug_goal(4);

    value = is (volume <=25000);

    czvolume_node[13].value = value;
    debug_or(4,  'T', 13);
    czvolume_set13();
    if (czvolume_fire4())
        return(TRUE);
    else
    {
        czvolume_fire_goal4();
    }
    return(FALSE);
}

czvolume_infer14()
{
    float old_weight;
    float diff = 0;

    debug_goal(4);
    old_weight = util_or_weight(czvolume_node[14]);

    while (!fired(czvolume_node[14].value) &&
           (diff <= G_czvolume_epsilon &&  diff >= -G_czvolume_epsilon ))
    {
```

```
        switch (*CZVOLUME_CHILD14)
        {
        case (2):
            if (czvolume_infer2())
                return(TRUE);
            break;
        default:
            assert(0);
            break;
        }
        diff = util_or_weight(czvolume_node[14]) - old_weight;
    }
    return(FALSE);
}

czvolume_infer15()
{
    float value;

    debug_goal(6);

    value = is (bidsize / volume > .25);

    czvolume_node[15].value = value;
    debug_or(6,  'T', 15);
    czvolume_set15();
    if (czvolume_fire6())
        return(TRUE);
    else
    {
        czvolume_fire_goal6();
    }
    return(FALSE);
}

czvolume_infer16()
{
    float value;

    debug_goal(7);

    value = is (bidsize / volume < .125);

    czvolume_node[16].value = value;
    debug_or(7,  'T', 16);
    czvolume_set16();
    if (czvolume_fire7())
        return(TRUE);
    else
    {
        czvolume_fire_goal7();
    }
    return(FALSE);
}

czvolume_infer17()
{
    float value;

    debug_goal(9);

    value = is(asksize / volume >= .25);

    czvolume_node[17].value = value;
    debug_or(9,  'T', 17);
    czvolume_set17();
```

```
    if (czvolume_fire9())
        return(TRUE);
    else
    {
        czvolume_fire_goal9();
    }
    return(FALSE);
}

czvolume_infer18()
{
    float value;

    debug_goal(10);

    value = is(asksize / volume < .12);

    czvolume_node[18].value = value;
    debug_or(10,  'T', 18);
    czvolume_set18();
    if (czvolume_fire10())
        return(TRUE);
    else
    {
        czvolume_fire_goal10();
    }
    return(FALSE);
}

czvolume_set7()
{
}

czvolume_set8()
{
    if (czvolume_node[8].value != CANT_ANSWER)
    {
        czvolume_node[9].value = czvolume_node[8].value;
        debug_or(2, 'X',  9);
    }
    else
        czvolume_node[9].value = CANT_ANSWER;
}

czvolume_set9()
{
    if (czvolume_node[9].value != CANT_ANSWER)
    {
        czvolume_node[8].value = czvolume_node[9].value;
        debug_or(2, 'T',  8);
    }
    else
        czvolume_node[8].value = CANT_ANSWER;
}

czvolume_set10()
{
    if (czvolume_node[10].value != CANT_ANSWER)
    {
        czvolume_node[11].value = 1.0 - czvolume_node[10].value;
        debug_or(3, 'F',  11);
        czvolume_node[12].value = czvolume_node[10].value;
        debug_or(3, 'X',  12);
    }
    else
        czvolume_node[11].value = czvolume_node[12].value = CANT_ANSWER;
```

```
}

czvolume_set11()
{
    if (czvolume_node[11].value != CANT_ANSWER)
    {
        czvolume_node[10].value = 1.0 - czvolume_node[11].value;
        debug_or(3, 'T',  10);
        czvolume_node[12].value = 1.0 - czvolume_node[11].value;
        debug_or(3, 'X',  12);
    }
    else
        czvolume_node[10].value = czvolume_node[12].value = CANT_ANSWER;
}

czvolume_set12()
{
    if (czvolume_node[12].value != CANT_ANSWER)
    {
        czvolume_node[10].value = czvolume_node[12].value;
        debug_or(3, 'T',  10);
        czvolume_node[11].value = 1.0 - czvolume_node[12].value;
        debug_or(3, 'F',  11);
    }
    else
        czvolume_node[10].value = czvolume_node[11].value = CANT_ANSWER;
}

czvolume_set13()
{
    if (czvolume_node[13].value != CANT_ANSWER)
    {
        czvolume_node[14].value = czvolume_node[13].value;
        debug_or(4, 'X',  14);
    }
    else
        czvolume_node[14].value = CANT_ANSWER;
}

czvolume_set14()
{
    if (czvolume_node[14].value != CANT_ANSWER)
    {
        czvolume_node[13].value = czvolume_node[14].value;
        debug_or(4, 'T',  13);
    }
    else
        czvolume_node[13].value = CANT_ANSWER;
}

czvolume_set15()
{
}

czvolume_set16()
{
}

czvolume_set17()
{
}

czvolume_set18()
{
}
```

```
czvolume_fire_goal0()
{
    if (czvolume_node[7].value == 1.0)
        czvolume_node[0].value = 1.0;
    else czvolume_node[0].value = CANT_ANSWER;
}

czvolume_fire_goal2()
{
    short stat = 0;
    short status2;
    short status1;

    status2 = czvolume_x_trav2();
    status1 = czvolume_x_trav1();

    stat += czvolume_fire_rule6();

    if (status1 == TRUE)
    {
        czvolume_fire_rule5();
        stat++;
    }

    if (status1 == TRUE)
    {
        czvolume_fire_rule4();
        stat++;
    }

    if (status2 == TRUE)
    {
        czvolume_fire_rule3();
        stat++;
    }
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal3()
{
    short stat = 0;
    short status1;

    status1 = czvolume_x_trav1();

    stat += czvolume_fire_rule5();
    stat += czvolume_fire_rule4();

    if (status1 == TRUE)
    {
        czvolume_fire_rule6();
        stat++;
    }

    if (status1 == TRUE)
    {
        czvolume_fire_rule3();
        stat++;
```

```
    }
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal4()
{
    short stat = 0;
    short status2;

    status2 = czvolume_x_trav2();

    stat += czvolume_fire_rule3();

    if (status2 == TRUE)
    {
        czvolume_fire_rule6();
        stat++;
    }

    if (status2 == TRUE)
    {
        czvolume_fire_rule5();
        stat++;
    }

    if (status2 == TRUE)
    {
        czvolume_fire_rule4();
        stat++;
    }
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal6()
{
    short stat = 0;
    stat += czvolume_fire_rule3();
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal7()
{
```

```
    short stat = 0;
    stat += czvolume_fire_rule4();
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal9()
{
    short stat = 0;
    stat += czvolume_fire_rule5();
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_goal10()
{
    short stat = 0;
    stat += czvolume_fire_rule6();
    if (stat > 0)
    {
        stat = util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
        debug_or(0, 'T', 7);
        if (stat == TRUE)
            czvolume_set7();

        czvolume_fire_goal0();
    }
}

czvolume_fire_rule3()
{
    if (!fired(czvolume_node[3].value))
    {
        util_and_update(czvolume_node, 3, CZVOLUME_CHILD3);
        debug_rule_fired(3);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(3);
        return(FALSE);
    }
}

czvolume_fire_rule4()
{
    if (!fired(czvolume_node[4].value))
    {
        util_and_update(czvolume_node, 4, CZVOLUME_CHILD4);
        debug_rule_fired(4);
        return(TRUE);
    }
```

```
        czv       node[16].value = gfb[12];
        deb       7, 'T', 16);
    }
    if (fire      [13]) && czvolume_node[17].value != gfb[13])
    {
        ret =     ;
        czvolu    ode[17].value = gfb[13];
        debug_    , 'T', 17);
    }
    if (fired(gfb[14]) && czvolume_node[18].value != gfb[14])
    {
        ret = TRUE;
        czvolume_node[18].value = gfb[14];
        debug_or(10, 'T', 18);
    }
    return(ret);
}

czvolume_update_to_gfb()
{
    short ret = FALSE;

    if (torf(czvolume_node[7].value) &&
        gfb[3] != czvolume_node[7].value)
    {
        ret = TRUE;
        gfb[3] = czvolume_node[7].value;
    }
    if (torf(czvolume_node[8].value) &&
        gfb[15] != czvolume_node[8].value)
    {
        ret = TRUE;
        gfb[15] = czvolume_node[8].value;
    }
    if (torf(czvolume_node[10].value) &&
        gfb[9] != czvolume_node[10].value)
    {
        ret = TRUE;
        gfb[9] = czvolume_node[10].value;
    }
    if (torf(czvolume_node[13].value) &&
        gfb[10] != czvolume_node[13].value)
    {
        ret = TRUE;
        gfb[10] = czvolume_node[13].value;
    }
    if (torf(czvolume_node[15].value) &&
        gfb[11] != czvolume_node[15].value)
    {
        ret = TRUE;
        gfb[11] = czvolume_node[15].value;
    }
    if (torf(czvolume_node[16].value) &&
        gfb[12] != czvolume_node[16].value)
    {
        ret = TRUE;
        gfb[12] = czvolume_node[16].value;
    }
    if (torf(czvolume_node[17].value) &&
        gfb[13] != czvolume_node[17].value)
    {
        ret = TRUE;
        gfb[13] = czvolume_node[17].value;
    }
    if (torf(czvolume_node[18].value) &&
        gfb[14] != czvolume_node[18].value)
```

```
    else
    {
        debug_rule_already_set(4);
        return(FALSE);
    }
}

czvolume_fire_rule5()
{
    if (!fired(czvolume_node[5].value))
    {
        util_and_update(czvolume_node, 5, CZVOLUME_CHILD5);
        debug_rule_fired(5);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(5);
        return(FALSE);
    }
}

czvolume_fire_rule6()
{
    if (!fired(czvolume_node[6].value))
    {
        util_and_update(czvolume_node, 6, CZVOLUME_CHILD6);
        debug_rule_fired(6);
        return(TRUE);
    }
    else
    {
        debug_rule_already_set(6);
        return(FALSE);
    }
}

czvolume_fire0()
{
    return(FALSE);
}

czvolume_fire2()
{
    return(FALSE);
}

czvolume_fire3()
{
    return(FALSE);
}

czvolume_fire4()
{
    return(FALSE);
}

czvolume_fire6()
{
    return(FALSE);
}

czvolume_fire7()
{
    return(FALSE);
}
```

```
czvolume_fire9()
{
    return(FALSE);
}

czvolume_fire10()
{
    return(FALSE);
}
czvolume_update_last()
{
    short i;
    for ( i=0; i < CZVOLUME_NODES; i++)
        czvolume_node[i].last = czvolume_node[i].value;
    return(SUCCESS);
}

czvolume_update_from_gfb()
{
    short ret = FALSE;

    if (G_gfb_changed == FALSE)
        return(ret);

    if (fired(gfb[3]) && czvolume_node[7].value != gfb[3])
    {
        ret = TRUE;
        czvolume_node[7].value = gfb[3];
        debug_or(0, 'T', 7);
    }
    if (fired(gfb[15]) && czvolume_node[8].value != gfb[15])
    {
        ret = TRUE;
        czvolume_node[8].value = gfb[15];
        debug_or(2, 'T', 8);
        czvolume_node[9].value = gfb[15];
        debug_or(2, 'X', 9);
    }
    if (fired(gfb[9]) && czvolume_node[10].value != gfb[9])
    {
        ret = TRUE;
        czvolume_node[10].value = gfb[9];
        debug_or(3, 'T', 10);
        czvolume_node[11].value = 1.0 - gfb[9];
        debug_or(3, 'F', 11);
        czvolume_node[12].value = gfb[9];
        debug_or(3, 'X', 12);
    }
    if (fired(gfb[10]) && czvolume_node[13].value != gfb[10])
    {
        ret = TRUE;
        czvolume_node[13].value = gfb[10];
        debug_or(4, 'T', 13);
        czvolume_node[14].value = gfb[10];
        debug_or(4, 'X', 14);
    }
    if (fired(gfb[11]) && czvolume_node[15].value != gfb[11])
    {
        ret = TRUE;
        czvolume_node[15].value = gfb[11];
        debug_or(6, 'T', 15);
    }
    if (fired(gfb[12]) && czvolume_node[16].value != gfb[12])
    {
        ret = TRUE;
```

```
    {
        ret = TRUE;
        gfb[14] = czvolume_node[18].value;
    }

    if (G_gfb_changed == FALSE)
        G_gfb_changed = ret;

    return(ret);
}

czvolume_validate()
{
    while (czvolume_val_trav0() == TRUE)
        continue;
    util_or_value(czvolume_node, 0, CZVOLUME_CHILD0);

    return(TRUE);
}


czvolume_val_trav0()
{
    short ret = FALSE;
    short status10;
    short status9;
    short status7;
    short status6;
    short status4;
    short status3;
    short status2;

    if (fired(czvolume_node[7].value))
    {
        if (czvolume_node[7].value != czvolume_node[7].last)
        {
            czvolume_node[7].last = czvolume_node[7].value;
            debug_fired(0);
            ret = TRUE;
        }
        else
        {
            ret = FALSE;
            debug_afired(0);
        }
    }
    else
    {
        status10 = czvolume_val_trav10();
        status2 = czvolume_val_trav2();

        if (status10 == TRUE ||
            status2 == TRUE)
        {
            util_and_update(czvolume_node, 6, CZVOLUME_CHILD6);
            czvolume_node[6].last = czvolume_node[6].value;
            debug_and(6);
        }
        else
            debug_a_and(6);

        status9 = czvolume_val_trav9();
        status3 = czvolume_val_trav3();

        if (status9 == TRUE ||
            status3 == TRUE)
```

```
        {
            util_and_update(czvolume_node, 5, CZVOLUME_CHILD5);
            czvolume_node[5].last = czvolume_node[5].value;
            debug_and(5);
        }
        else
            debug_a_and(5);

        status7 = czvolume_val_trav7();

        if (status7 == TRUE ||
            status3 == TRUE)
        {
            util_and_update(czvolume_node, 4, CZVOLUME_CHILD4);
            czvolume_node[4].last = czvolume_node[4].value;
            debug_and(4);
        }
        else
            debug_a_and(4);

        status6 = czvolume_val_trav6();
        status4 = czvolume_val_trav4();

        if (status6 == TRUE ||
            status4 == TRUE)
        {
            util_and_update(czvolume_node, 3, CZVOLUME_CHILD3);
            czvolume_node[3].last = czvolume_node[3].value;
            debug_and(3);
        }
        else
            debug_a_and(3);


        if (status10 == TRUE ||
            status9 == TRUE ||
            status7 == TRUE ||
            status6 == TRUE ||
            status4 == TRUE ||
            status3 == TRUE ||
            status2 == TRUE)
        {
            util_or_update(czvolume_node, 7, CZVOLUME_CHILD7);
            czvolume_node[7].last = czvolume_node[7].value;
            ret = TRUE;
            debug_or(0, 'T', 7);
        }
        else
            debug_a_or(0, 'T', 7);

    }

    return(ret);
}

czvolume_val_trav2()
{
    short ret = FALSE;

    czvolume_x_trav2();
    czvolume_x_trav1();

    if (czvolume_node[8].value != czvolume_node[8].last ||
        czvolume_node[9].value != czvolume_node[9].last)
    {
        czvolume_node[8].last = czvolume_node[8].value;
```

```
        czvolume_node[9].last = czvolume_node[9].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_val_trav3()
{
    short ret = FALSE;

    czvolume_x_trav1();

    if (czvolume_node[10].value != czvolume_node[10].last ||
        czvolume_node[11].value != czvolume_node[11].last ||
        czvolume_node[12].value != czvolume_node[12].last)
    {
        czvolume_node[10].last = czvolume_node[10].value;
        czvolume_node[11].last = czvolume_node[11].value;
        czvolume_node[12].last = czvolume_node[12].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_val_trav4()
{
    short ret = FALSE;

    czvolume_x_trav2();

    if (czvolume_node[13].value != czvolume_node[13].last ||
        czvolume_node[14].value != czvolume_node[14].last)
    {
        czvolume_node[13].last = czvolume_node[13].value;
        czvolume_node[14].last = czvolume_node[14].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_val_trav6()
{
    short ret = FALSE;

    if (czvolume_node[15].value != czvolume_node[15].last)
    {
        czvolume_node[15].last = czvolume_node[15].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_val_trav7()
{
    short ret = FALSE;

    if (czvolume_node[16].value != czvolume_node[16].last)
    {
        czvolume_node[16].last = czvolume_node[16].value;
        ret = TRUE;
    }
```

```
    return(ret);
}

czvolume_val_trav9()
{
    short ret = FALSE;

    if (czvolume_node[17].value != czvolume_node[17].last)
    {
        czvolume_node[17].last = czvolume_node[17].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_val_trav10()
{
    short ret = FALSE;

    if (czvolume_node[18].value != czvolume_node[18].last)
    {
        czvolume_node[18].last = czvolume_node[18].value;
        ret = TRUE;
    }

    return(ret);
}

czvolume_x_trav1()
{
    short ret = FALSE;

    if ((czvolume_node[9].value == 1.0 || czvolume_node[9].value == 0.0) &&
        !fired(czvolume_node[12].value))
    {
        czvolume_node[12].value = 1.0 - czvolume_node[9].value;
        czvolume_set12();
        debug_x_fired(1);
        ret = TRUE;
    }
    else if ((czvolume_node[12].value == 1.0 || czvolume_node[12].value == 0.0) &
        !fired(czvolume_node[9].value))
    {
        czvolume_node[9].value = 1.0 - czvolume_node[12].value;
        czvolume_set9();
        debug_x_fired(1);
        czvolume_x_trav2();
        ret = TRUE;
    }
    else debug_x_not_fired(1);

    return(ret);
}

czvolume_x_trav2()
{
    short ret = FALSE;

    if ((czvolume_node[9].value == 1.0 || czvolume_node[9].value == 0.0) &&
        !fired(czvolume_node[14].value))
    {
        czvolume_node[14].value = 1.0 - czvolume_node[9].value;
        czvolume_set14();
        debug_x_fired(2);
        ret = TRUE;
```

```
    }
    else if ((czvolume_node[14].value == 1.0 || czvolume_node[14].value == 0.0) &&
        !fired(czvolume_node[9].value))
    {
        czvolume_node[9].value = 1.0 - czvolume_node[14].value;
        czvolume_set9();
        debug_x_fired(2);
        czvolume_x_trav1();
        ret = TRUE;
    }
    else debug_x_not_fired(2);

    return(ret);
}

czvolume_reset()
{
    G_czvolume_epsilon = CZVOLUME_EPSILON;

    czvolume_pt[5] = 13;
    czvolume_pt[6] = 15;
    czvolume_pt[8] = 10;
    czvolume_pt[9] = 16;
    czvolume_pt[11] = 17;
    czvolume_pt[12] = 11;
    czvolume_pt[14] = 18;
    czvolume_pt[15] = 8;
    czvolume_pt[17] = 5;
    czvolume_pt[18] = 6;
    czvolume_pt[19] = 3;
    czvolume_pt[20] = 4;
    czvolume_pt[23] = 1;
    czvolume_pt[24] = 2;

    czvolume_node[0].cost = CZVOLUME_ICOST0;
    czvolume_node[0].value = CZVOLUME_IVAL0;
    czvolume_node[0].last = CZVOLUME_IVAL0;
    czvolume_node[1].cost = CZVOLUME_ICOST1;
    czvolume_node[1].value = CZVOLUME_IVAL1;
    czvolume_node[1].last = CZVOLUME_IVAL1;
    czvolume_node[2].cost = CZVOLUME_ICOST2;
    czvolume_node[2].value = CZVOLUME_IVAL2;
    czvolume_node[2].last = CZVOLUME_IVAL2;
    czvolume_node[3].cost = CZVOLUME_ICOST3;
    czvolume_node[3].value = CZVOLUME_IVAL3;
    czvolume_node[3].last = CZVOLUME_IVAL3;
    czvolume_node[4].cost = CZVOLUME_ICOST4;
    czvolume_node[4].value = CZVOLUME_IVAL4;
    czvolume_node[4].last = CZVOLUME_IVAL4;
    czvolume_node[5].cost = CZVOLUME_ICOST5;
    czvolume_node[5].value = CZVOLUME_IVAL5;
    czvolume_node[5].last = CZVOLUME_IVAL5;
    czvolume_node[6].cost = CZVOLUME_ICOST6;
    czvolume_node[6].value = CZVOLUME_IVAL6;
    czvolume_node[6].last = CZVOLUME_IVAL6;
    czvolume_node[7].cost = CZVOLUME_ICOST7;
    czvolume_node[7].value = CZVOLUME_IVAL7;
    czvolume_node[7].last = CZVOLUME_IVAL7;
    czvolume_node[8].cost = CZVOLUME_ICOST8;
    czvolume_node[8].value = CZVOLUME_IVAL8;
    czvolume_node[8].last = CZVOLUME_IVAL8;
    czvolume_node[9].cost = CZVOLUME_ICOST9;
    czvolume_node[9].value = CZVOLUME_IVAL9;
    czvolume_node[9].last = CZVOLUME_IVAL9;
    czvolume_node[10].cost = CZVOLUME_ICOST10;
    czvolume_node[10].value = CZVOLUME_IVAL10;
```

```
    czvolume_node[10].last = CZVOLUME_IVAL10;
    czvolume_node[11].cost = CZVOLUME_ICOST11;
    czvolume_node[11].value = CZVOLUME_IVAL11;
    czvolume_node[11].last = CZVOLUME_IVAL11;
    czvolume_node[12].cost = CZVOLUME_ICOST12;
    czvolume_node[12].value = CZVOLUME_IVAL12;
    czvolume_node[12].last = CZVOLUME_IVAL12;
    czvolume_node[13].cost = CZVOLUME_ICOST13;
    czvolume_node[13].value = CZVOLUME_IVAL13;
    czvolume_node[13].last = CZVOLUME_IVAL13;
    czvolume_node[14].cost = CZVOLUME_ICOST14;
    czvolume_node[14].value = CZVOLUME_IVAL14;
    czvolume_node[14].last = CZVOLUME_IVAL14;
    czvolume_node[15].cost = CZVOLUME_ICOST15;
    czvolume_node[15].value = CZVOLUME_IVAL15;
    czvolume_node[15].last = CZVOLUME_IVAL15;
    czvolume_node[16].cost = CZVOLUME_ICOST16;
    czvolume_node[16].value = CZVOLUME_IVAL16;
    czvolume_node[16].last = CZVOLUME_IVAL16;
    czvolume_node[17].cost = CZVOLUME_ICOST17;
    czvolume_node[17].value = CZVOLUME_IVAL17;
    czvolume_node[17].last = CZVOLUME_IVAL17;
    czvolume_node[18].cost = CZVOLUME_ICOST18;
    czvolume_node[18].value = CZVOLUME_IVAL18;
    czvolume_node[18].last = CZVOLUME_IVAL18;
}
```