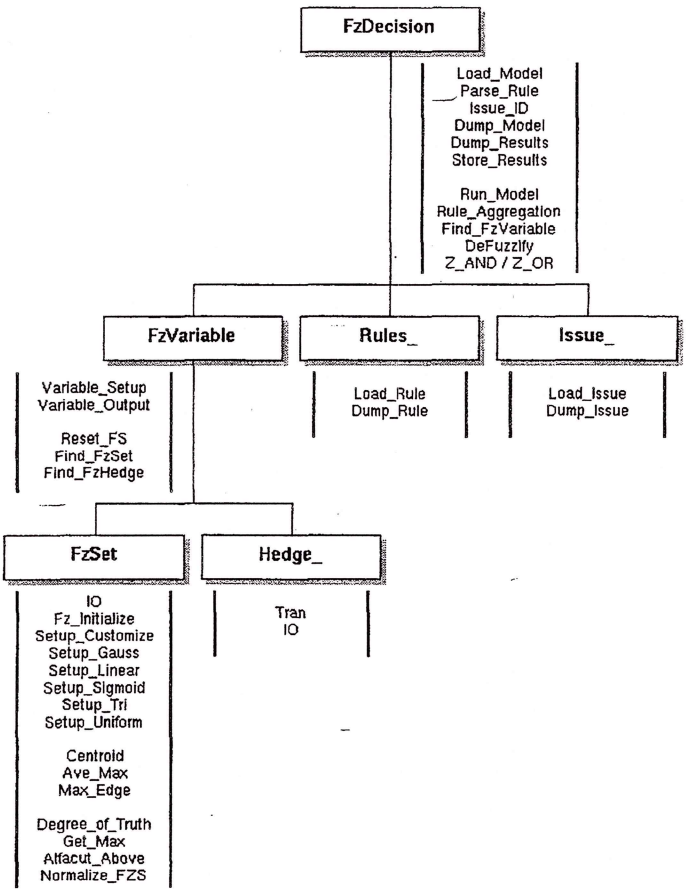
// **+++++ Fuzzy Decision Algorythms (FzDecision) +++++**

class FzDecision

```
{
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//          This class represents the highest (or lowest) block of atributes and
//          functions needed to perform fuzzy modeling. All other classes are
//          supporting members of the FzDecision class.
//
//          References:
//          Cox,E., The Fuzzy Systems Handbook, AP Professional
//          (a div of Harcourt Brace & Co.), 1994, ISBN 0-12-194270-8
//          Klir, G.J., Folger, T.A., Fuzzy Sets, Uncertainty & Informtion,
//          Prentice Hall, 1988, ISBN 0-13-345984-5
//
//          Code structure :
//
```

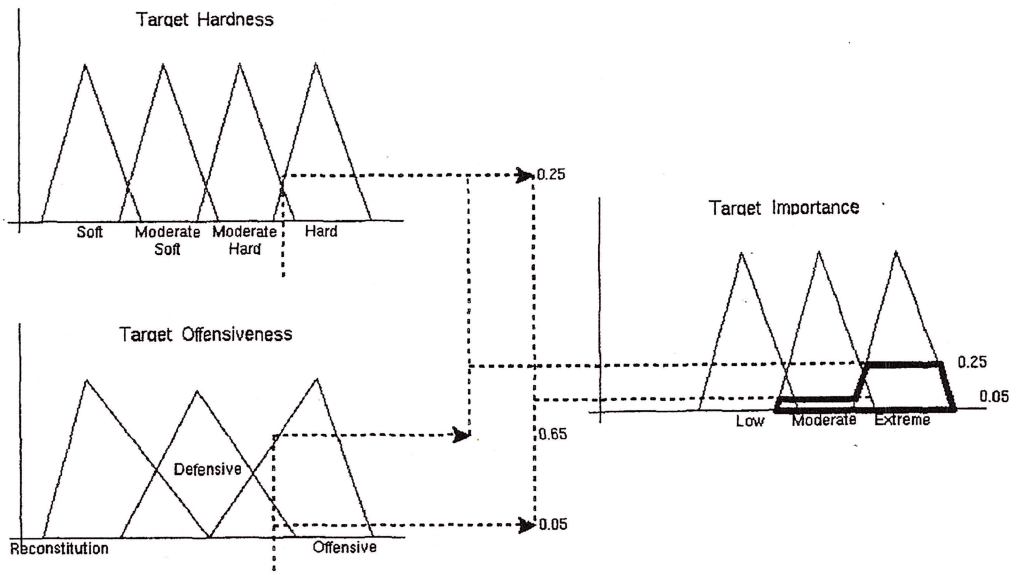



```
//
```

// Program construction:
//
// Main program requires a FzDecision variable declaration, a call to
// **Load_Model**, a call to **Run_Model**, and **Dump_Results** to
// constitute a fuzzy program. An example follows:
//

```
// FzDecision bs1;
// bs1.Load_Model( argv[1] );
// bs1.Dump_Model( "z_info" );
// bs1.Run_Model( "z_tracking" );
// bs1.Dump_Results( "z_result" );
```

//
// *File names are passed as arguments for defining; input, tracking,*
// and output.
//
// Member Functions include:
//
// Setup and output related functions:
// **Load_Model, Dump_Model, Dump_Results, Issue_ID,**
// Manipulation & transformation related functions:
// **Store_Results, Find_FzVariable, Parse_Rule, Reset_Var,**
// **Rule_Aggregation,**
// Execution related functions:
// **Run_Model, DeFuzzify, Z_AND, Z_OR( );**
//
// The following depicts the processing of a fuzzy rule(s):



If **Target Hardness** is **Hard** and **Target Offensiveness** is **Offensive** then **Target Importance** is **Extreme**

if **Target Hardness** is **Moderate Hard** and **Target Offensiveness** is **Defensive** then **Target Importance** is **Moderate**

```
//
//              The degree of truth of each preposition is acertained and then a
//              Zadah type AND_ function is activated to determin the level to
//              associate with the then part of the proposition.  As a result of the
//              2 rules the heavy surface is defined from which defuzification will
//              produce a quantitative assessment of the targeting importance.
//
//              The Issue_  construct of these algorithms permits a structured
//              approach for answering multiple issues as well as coupling a
//              number of variables into a global model.
//
//              Input Format:
//
//              Load_Model() controls input of all elments of the fuzzy model, it
//              may transfer control to another function but the call is initiated in
//              FzDecision member functions.
//
//              Input Keywords include:
//              rule, variables, issues, defuzzify, runs, & eoi for termination.
//
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  public:

  int De_Fuz;
  int max_runs;
  FzVariable* var;
  FzVariable* last_var;

  Rules_* first_rule;
  Rules_* last_rule;
  Rules_* current_rule;

  Issue_* first_issue;
  Issue_* last_issue;
  Issue_* current_issue;

  struct Parsed_
   {
//
//              Note:  more space has been allocated(4)than allowed(3) for
//              rule atoms to mitigate a problem that surfaced during initial
//              development.  The problem surface in the function which parses
//              the rules.     no idea 10/30/94
//
          char ID[5];
          int n_pred;
```

```
            FzVariable* Vr[4];
            Hedge_* Hd[4];
            FzSet* St[4];
            int Opij[4];
            int Aggregation_Op ;
            Parsed_ *next;
            Parsed_( Parsed_ *p )
              { next=p; n_pred=3; }
  } ;     // Block: 115

  Parsed_* pi_r;
  Parsed_* pi_first;

  double final[VECMAX];
  double wrk[VECMAX];
//
//              Define Addiitonal Member Functions:
//
                FzDecision();
  void          Assess_Issues( float*, float* );
  void          DeFuzzify( int );
  void          Dump_Model( char* );
  void          Dump_Results( char* );
  FzVariable*   Find_FzVariable( char* );
  void          Issue_ID( );
  void          Load_Model( char* );
  void          Parse_Rule();
  void          Reset_Var();
  void          Rule_Aggregation( FzVariable*, int );
  void          Run_Model( char* );
  void          Store_Results( int, ostream& out );
  void          Z_AND( );
  void          Z_OR( );
  int           Z_Truth( );
  } ;     // Block: 116
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//              NOTE:  The integers in the constructor must be adjusted to reflect
//              the number of variables and arguments in the model

  FzDecision::FzDecision( )

  {
//              Define default conditions
  De_Fuz=CENTROID;

  first_rule=0;
```

```
  last_rule=0;

  first_issue=0;
  last_issue=0;

  last_var=0;
  pi_first=0;

  max_runs=1;
 } ;     // Block: 117
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//              Function That Runs The Fuzzy Logic Algorythms for an issue
//              assuming that the algorithm is buried in a code that can set
//              case scler values on the fly.
```

void FzDecision::Assess_Issues(float* info_in, float* info_out)

```
 {
  FzVariable* V[3];
  Hedge_* H[3];
  FzSet* S[3];
  double tr_V;
  double final_truth;
  int i,j,i_if,i_then, aggr_op;
  Issue_::RO_* loc_ro;
  int issue_index=0;
//
//              Set Outer Outer Loop To Handle Multiple Issues In
//              The Decision Space.
//
//              Issues correlate to finding consequent fuzzy sets,  e.g. A problem is
//              searching for priorities of a process as well as trying to determine
//              overall risk, in this case we are dealing with two basic issues.
//
  for( current_issue= first_issue; current_issue != 0; current_issue= current_issue-
>next )
    {
    int j=0;
      {
      for(i=0; i<VECMAX; i++)
        final[i]=0;
//
//              Each Issue Is Comprised Of A Number Of Rules, 1-Many.
//
      for( loc_ro= current_issue->rl1; loc_ro != 0; loc_ro= loc_ro->next )
        {
```

```
        pi_r= (Parsed_*)loc_ro->add_;
        Reset_Var();
//
//              Identify All Variables In The Rule, All Hedges And All Fuzzy Sets
//
        for( i=0; i<pi_r->n_pred; i++ )
          {
          V[i]= (pi_r->Vr[i]);
          H[i]= (pi_r->Hd[i]);
          S[i]= (pi_r->St[i]);
          }
//
//              Perform All Hedge Operations
//
        for( i=0; i<pi_r->n_pred; i++ )
        if( H[i] != 0 )
        H[i]->Tran( *S[i] );
//
//              This Identifies The Proposition Predicate Portion Of The Rule And
//              The Consequent Portion Of The Rule Under Evaluation.
//
        i_if=i_then=0;
        for( i=0; i<pi_r->n_pred; i++ )
          {
          if( pi_r->Opij[i] == IF_ ) i_if=i;
          if( pi_r->Opij[i] == THEN_ ) i_then=i;
          }
//
//              Assess The Degree Of Truth For The If Predicates.  Each
//              proposition predicate exhibits a degree of truth based on the value
//              of the scalers used in the current run.  These values are used to
//              condition the consequent.
//
        final_truth=1.0;
        for( i=i_if; i<i_then; i++ )
          {
          V[i]->scaler[j] = info_in[ i ];
          switch ( pi_r->Opij[i] )
            {
            case IF_ :
            case AND_ :
              tr_V= S[i]->Degree_of_Truth( V[i]->scaler[j] );
              if( tr_V < final_truth )
                final_truth= tr_V;
              break;
            case OR_ :
```

```
                tr_V= S[i]->Degree_of_Truth( V[i]->scaler[j] );
                if( tr_V > final_truth )
                  final_truth= tr_V;
                break;
              default :
                break;
              }
            }
//
//              Work The Then Side Of The Predicates;assumes A Zadah Type And
//
        for( int k=0; k<VECMAX; k++ )
        if( final_truth < S[i_then]->mu_wrk[k] )
          wrk[k]= final_truth;
        else
          wrk[k]= S[i_then]->mu_wrk[k];
//
//              Final rule(s) combinatorics
//
        if( i_then <= 0 )
          aggr_op=MIN_MIN;
        else
          aggr_op=pi_r->Aggregation_Op ;
        Rule_Aggregation( V[i_then], aggr_op );
        }
//
// Com 36      De-fuzzification Must Now Be Performed
//
      DeFuzzify( j );
      }
//              ------------------------------
    V[i_then]->result_var = FALSE;
    info_out[ issue_index ] = V[i_then]->scaler[j] ;
    issue_index +=1;
//              ------------------------------
    }
  return;
 } ;     // Block: 118
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  //Fuction which de fuzzifies the solution variable after a model run

  void FzDecision::DeFuzzify( int ijk )

 {
  double result=0;
  FzSet* fs_tmp;
```

```
    Reset_Var();

    var=last_var;
    while( var!=0 )
      {
      if( var->result_var == TRUE )
        {
        fs_tmp = var->fs_result;
        if( var->num_scaler <= 0 )
          {
          var->num_scaler= max_runs;
          var->scaler= new double[ max_runs ];
          var->D_o_T = new double[ max_runs ];
          }
        switch ( De_Fuz )
          {
          case CENTROID :
            result= fs_tmp->Centroid();
            break;
          case AVGMAXIMUM :
            result= fs_tmp->Ave_Max();
            break;
          case MAXIMUM :
            result= fs_tmp->Max_Edge();
            break;
          default :
            result=1.0e9;
            break;
          }
        var->scaler[ijk]= result;
        var->D_o_T[ijk]= fs_tmp->Degree_of_Truth(result);
        }
      var=var->next;
      }

    return;
    } ;     // Block: 119
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Dump_Model(char* O_file)

```
{
 ofstream out( O_file );

 for( current_rule= first_rule; current_rule !=0; current_rule= current_rule->next )
```

```
    {
    out << "\nRule: " << (*current_rule).ID << "\n";
    current_rule->Dump_rule( out );
    }
  for( var= last_var; var !=0; var= var->next )
    var->Variable_Output( out );
  return;
 } ;     // Block: 120
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Dump_Results(char* O_file)

```
 {
  ofstream out( O_file );
  int i;
  var=last_var;
  while( var!=0 )
    {
    if( var->result_ == TRUE )
      {
      out << "\n Fuzzy Variable:" << var->Fz_Var_ID << "\n";
      for( i=0; i<var->num_scaler; i++ )
        out << " " << var->Fz_Var_ID << "[ " << i+1 << " ] "
        << var->scaler[i] << " DoT "<< var->D_o_T[i] << "\n";
//
//              Print Out Again For Use With Plotting Programs
//
      out << "\n\n Fuzzy Variable:" << var->Fz_Var_ID << "(Scaler Values)\n";
      for( i=0; i<var->num_scaler; i++ )
        out << " " << var->scaler[i];
      out << "\n\n Fuzzy Variable:" << var->Fz_Var_ID << "(Deg of Truth)\n";
      for( i=0; i<var->num_scaler; i++ )
        out << " " << var->D_o_T[i];
      }
    var=var->next;
    }
//
//              for( var= last_var; var !=0; var= var->next )
//              var->Variable_Output( out );
  return;
 } ;     // Block: 121
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

FzVariable* FzDecision::Find_FzVariable(char* v_nam)

```
 {
```

```
  FzVariable* rtn_v;
  rtn_v=0;
  var=last_var;
  while( var!=0 )
    {
    if( strcmp( var->Fz_Var_ID, v_nam ) == 0 )
      rtn_v=var;
    var=var->next;
    }

  return( rtn_v );
 } ;     // Block: 122
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 // a then logic function

   void FzDecision::Issue_ID( )

 {
  Issue_::RO_* loc_ro;

  for( current_issue= first_issue; current_issue != 0; current_issue= current_issue-
>next )
    {
    loc_ro=(current_issue->rl1);
    while (1)
      {
//
//              Now compare the rule with a parsed rule for a series of matches
//
      pi_r= pi_first;
      while( pi_r != 0 )
        {
        if( strcmp( loc_ro->r_ID, pi_r->ID ) == 0 )
          loc_ro->add_ = pi_r;
        pi_r= pi_r->next;
        }
      if( loc_ro == (current_issue->rln) )
        break;
      loc_ro= loc_ro->next;
      }
    }
  return;
 } ;     // Block: 123
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Load_Model(char* l_file)

```
{
 ifstream in( l_file );
 char tmp_in[15];
 in >> tmp_in;
 while( strcmp( tmp_in,"eoi" ) !=0 )
   {
 cout << "-----      FzDecision Keyword : " << tmp_in << "\n" << flush;
   if( strcmp(tmp_in,"rule") ==0 )
     {
     current_rule = new Rules_( last_rule );
     last_rule= current_rule;
     current_rule->Load_rule( in );
     in >> tmp_in;
     }
   else if( strcmp(tmp_in,"variables") ==0 )
     {
     var = new FzVariable( last_var );
     last_var= var;
     var->Variable_Setup( in );
     in >> tmp_in;
     }
   else if( strcmp(tmp_in,"issues") ==0 )
     {
     current_issue = new Issue_( last_issue );
     last_issue= current_issue;
     current_issue->Load_issue( in );
     in >> tmp_in;
     }
   else if( strcmp(tmp_in,"defuzzify") ==0 )
     {
     in >> tmp_in;
     if( strcmp( tmp_in, "CENTROID" ) == 0 )
       De_Fuz=CENTROID;
     else if( strcmp( tmp_in, "AVGMAXIMUM" ) == 0 )
       De_Fuz=AVGMAXIMUM;
     else if( strcmp( tmp_in, "MAXIMUM" ) == 0 )
       De_Fuz=MAXIMUM;
     else
       cout << "\nERROR: Not a developed defuzzification routine" << flush;
     in >> tmp_in;
     }
   else if( strcmp(tmp_in,"runs") ==0 )
     {
     in >> max_runs;
```

```
        in >> tmp_in;
        }
      else
        {
        cerr << "Error: (FzDecision Input) unaceptable input option : " << tmp_in
         << "\n" << flush;
        }
      };
//
//              Identify the first rule in the decision space
//
  for( current_rule= last_rule; current_rule->prev != 0;
   current_rule= current_rule->prev )
    {
    (current_rule->prev)->next= current_rule;
    }
  first_rule= current_rule;
//
// Com 37       Execute The Function Which Parses The Rules;
//              Rules And Variables Must Be Loaded
//
  Parse_Rule();
//
// Com 38       Identify The First Issue In The Decision Space
//
  for( current_issue= last_issue; current_issue->prev != 0; current_issue=
current_issue->prev )
    {
    (current_issue->prev)->next= current_issue;
    }
  first_issue= current_issue;
//
// Com 39       Identify & Associate The Address Of The Parsed Rule With An Issue
//
  Issue_ID();

  return;
 } ;     // Block: 124
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Parse_Rule( )

```
 {
  FzSet* tmp_St;
  Hedge_* tmp_Hd;
  FzVariable* tmp_Vr;
```

```
  int cnt;
//
// Com 40      Parse The Rules In Order To Prepare For Execution
//
  for( current_rule= last_rule; current_rule != 0; current_rule= current_rule->prev )
    {
    pi_r = new Parsed_( pi_first );
    pi_first= pi_r;
//
//              Zero Out Elements In The Parsed Structure
//
    for(int i=0; i<pi_r->n_pred; i++)
      {
      pi_r->Vr[i]=0;
      pi_r->Hd[i]=0;
      pi_r->St[i]=0;
      pi_r->Opij[i]=999;
      }

    strcpy( pi_r->ID, current_rule->ID );
    pi_r->Aggregation_Op=current_rule->Aggregation_Op;
    current_rule->at=current_rule->at1;
    cnt=0;
    while( current_rule->at != 0 )
      {
//
//              Check On Next Atom Being A Variable
//
      tmp_Vr = Find_FzVariable( (current_rule->at)->atom );
      if( tmp_Vr != 0 )
        {
        pi_r->Vr[cnt]= tmp_Vr;
        current_rule->at= (current_rule->at)->np1;
//
//              If We Found A Variable, Chect On Likelihood Of A Hedge
//
        tmp_Hd= tmp_Vr->Find_FzHedge((current_rule->at)->atom );
        if( tmp_Hd != 0 )
          {
          pi_r->Hd[cnt]= tmp_Hd;
          current_rule->at= (current_rule->at)->np1;
          }
//
//              if we found a variable, chect on likelihood of a fuzzy set
//
        tmp_St= tmp_Vr->Find_FzSet((current_rule->at)->atom );
```

```
            if( tmp_St != 0 )
              {
              pi_r->St[cnt]= tmp_St;
              current_rule->at= (current_rule->at)->np1;
              }
            else
              pi_r->St[cnt]= tmp_Vr->fs_result;
            cnt++;
            }
          else
//
//                chect on likelihood of a operator
//
            {
            if( strcmp( (current_rule->at)->atom, "AND" ) == 0 ||
            strcmp( (current_rule->at)->atom, "and" ) == 0)
              pi_r->Opij[cnt]= AND_;
            else if( strcmp( (current_rule->at)->atom, "OR" ) == 0 ||
             strcmp( (current_rule->at)->atom, "or" ) == 0)
              pi_r->Opij[cnt]= OR_;
            else if( strcmp( (current_rule->at)->atom, "THEN" ) == 0 ||
             strcmp( (current_rule->at)->atom, "then" ) == 0)
              pi_r->Opij[cnt]= THEN_;
            else if( strcmp( (current_rule->at)->atom, "IF" ) == 0 ||
             strcmp( (current_rule->at)->atom, "if" ) == 0)
              pi_r->Opij[cnt]= IF_;
            else
              cerr << "ERROR: could not find an operator for RULE ID "
               << current_rule->ID << flush;
            current_rule->at= (current_rule->at)->np1;
            }
          }
        }
      return;
     } ;     // Block: 125
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Reset_Var( )

```
{
 var=last_var;
 while( var !=0 )
   {
   var->Reset_FS();
   var= var->next;
   }
```

15 Barrland Street • Glasgow • G41 1QH •
Tel: 0141 423 1222 Fax: 0141 433 2821 e-mail: scnd@banthebomb.org

```
 for(int i=0; i<VECMAX; i++)
   {
   wrk[i]=0;
   }
 return;
 } ;     // Block: 126
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//              A "Then" Logic Function.  This function provides the mechanism for
//              combining sets of rule consequents.  Given a set of rules in which
//              RISK is a consequent variable, the results must be convolved to
//              generate the final consequent fuzzy set.
//
```

void FzDecision::Rule_Aggregation(FzVariable* A, int aggr_op)

```
 {
  FzSet* s_tmp;
  int j;
//
//              If the "solution fuzzy set" (sfs) is zero everywhere, then use a
//              MIN_MAX operator to define a default concequent fuzzy set.
//              (used in conjunction with unconditional propositions)
//
  if( Z_Truth() == FALSE )
    aggr_op = MIN_MAX;

  switch( aggr_op)
    {
    case MIN_MAX :
//
//              This is te rule combinatorics associated with precedents and
//              consequents using the Zadah intersection operaor.  (Z_AND)
//              (Cox, page 226, The Fuzzy Systems Handbook )
//
      for( j=0; j<VECMAX; j++ )
        if( wrk[j] > final[j] )
          final[j]= wrk[j];
      break;
    case MIN_MIN :
//
//              Used for unconditional consequents.  E.g. then RISK is HIGH
//              This is basically the Zadah Union operator (Z_OR)
//              (Cox, page 239, The Fuzzy Systems Handbook )
//
```

```
        for( j=0; j<VECMAX; j++ )
          if( wrk[j] < final[j] )
            final[j]= wrk[j];
        break;
      case ADDITIVE :
//
//              This rule aggregation mechanism is best used for situations
//              building evidence. (See Cox, The Fuzzy Systems Handbook,
//              page 227)
//
        for( j=0; j<VECMAX; j++ )
          {
          final[j]+= wrk[j];
          if( final[j] > 1.0 )
            final[j]=1.0;
          }
        break;
      default:
        for( j=0; j<VECMAX; j++ )
          final[j]=0.0;
      }

    s_tmp = A->fs_result;

    if( A->result_var != TRUE )
      {
      A->result_var=TRUE;
      A->result_ =TRUE;
      s_tmp->domain[0]= (A->last_fs)->domain[0];
      s_tmp->domain[1]= (A->last_fs)->domain[1];
      }
    for( j=0; j<VECMAX; j++ )
    s_tmp->mu[j]= final[j];

    return;
   } ;     // Block: 127
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Com 41       Function That Runs The Fuzzy Logic Algorythms

    void FzDecision::Run_Model( char* O_file )

   {
    ofstream tr_out( O_file );
    FzVariable* V[3];
    Hedge_* H[3];
    FzSet* S[3];
```

```
  double tr_V;
  double final_truth;
  int i,j,i_if,i_then, aggr_op;
  Issue_::RO_* loc_ro;
  ofstream out_store( "z_store" );
//
//              Set Outer Outer Loop To Handle Multiple Issues In
//              The Decision Space.
//
//              Issues correlate to finding consequent fuzzy sets,  e.g. A problem is
//              searching for priorities of a process as well as trying to determine
//              overall risk, in this case we are dealing with two basic issues.
//
  for( current_issue= first_issue; current_issue != 0; current_issue= current_issue-
>next )
    {
    tr_out << "\n1---> Processing " << current_issue->ID << " issues...." << flush;
//
//              Set Outer Loop To Handle Multiple Variable Value Runs;
//              basically what value do the consequent fuzzy variables become
//              given a set of MODEL variable settings.
//
    for( j=0; j<max_runs; j++ )
      {
      tr_out << "\n2-----> Processing case " << j << " ...." << flush;
//              ------------------------------
      for(i=0; i<VECMAX; i++)
        final[i]=0;
//
//              Each Issue Is Comprised Of A Number Of Rules, 1-Many.
//
      for( loc_ro= current_issue->rl1; loc_ro != 0; loc_ro= loc_ro->next )
        {
        tr_out << "\n3-------> Processing rule " << loc_ro->r_ID << " ...." << flush;
        pi_r= (Parsed_*)loc_ro->add_;
        Reset_Var();
//
//              Identify All Variables In The Rule, All Hedges And All Fuzzy Sets
//
        for( i=0; i<pi_r->n_pred; i++ )
          {
          V[i]= (pi_r->Vr[i]);
          H[i]= (pi_r->Hd[i]);
          S[i]= (pi_r->St[i]);
          }
//
```

```
//              Perform All Hedge Operations
//
        for( i=0; i<pi_r->n_pred; i++ )
        if( H[i] != 0 )
        H[i]->Tran( *S[i] );
//
//              This Identifies The Proposition Predicate Portion Of The Rule And
//              The Consequent Portion Of The Rule Under Evaluation.
//
        i_if=i_then=0;
        for( i=0; i<pi_r->n_pred; i++ )
          {
          if( pi_r->Opij[i] == IF_ ) i_if=i;
          if( pi_r->Opij[i] == THEN_ ) i_then=i;
          }
//
//              Assess The Degree Of Truth For The If Predicates.  Each
//              proposition predicate exhibits a degree of truth based on the value
//              of the scalers used in the current run.  These values are used to
//              condition the consequent.
//
        final_truth=1.0;
        for( i=i_if; i<i_then; i++ )
          {
          switch ( pi_r->Opij[i] )
            {
            case IF_ :
            case AND_ :
              tr_V= S[i]->Degree_of_Truth( V[i]->scaler[j] );
              if( tr_V < final_truth )
                final_truth= tr_V;
              break;
            case OR_ :
              tr_V= S[i]->Degree_of_Truth( V[i]->scaler[j] );
              if( tr_V > final_truth )
                final_truth= tr_V;
              break;
            default :
              break;
            }
          }
//
//              Work The Then Side Of The Predicates;assumes A Zadah Type And
//
        for( int k=0; k<VECMAX; k++ )
        if( final_truth < S[i_then]->mu_wrk[k] )
```

```
          wrk[k]= final_truth;
        else
          wrk[k]= S[i_then]->mu_wrk[k];
//
//              Final rule(s) combinatorics
//
        if( i_then <= 0 )
          aggr_op=MIN_MIN;
        else
          aggr_op=pi_r->Aggregation_Op ;
        Rule_Aggregation( V[i_then], aggr_op );
        }
//
// Com 42      De-fuzzification Must Now Be Performed
//
      DeFuzzify( j );
      Store_Results( j, out_store );
      }
//              ------------------------------
    V[i_then]->result_var = FALSE;
//              ------------------------------
    }
  return;
 } ;     // Block: 128
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

void FzDecision::Store_Results(int case_num, ostream& out)

```
 {
  FzSet* fs_tmp;
  int i;

  var=last_var;
  while( var!=0 )
    {
    if( var->result_var == TRUE )
      {
      fs_tmp = var->Find_FzSet( "RESULT_" );
      out << "\n Case Number:" << case_num+1 << "";
      out << " Fuzzy Variable:" << var->Fz_Var_ID << "\n";
      out << *fs_tmp;
      }
//
//              Print Out Again For Use With Plotting Programs
//
    out << "\n\n Fuzzy Variable:" << var->Fz_Var_ID << "(Scaler Values)\n";
```

```
    for( i=0; i<var->num_scaler; i++ )
      out << " " << var->scaler[i];
    out << "\n\n Fuzzy Variable:" << var->Fz_Var_ID << "(Deg of Truth)\n";
    for( i=0; i<var->num_scaler; i++ )
      out << " " << var->D_o_T[i];
    var=var->next;
    }
//
//              for( var= last_var; var !=0; var= var->next )
//              var->Variable_Output( out );
  return;
 } ;     // Block: 129
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//              Zadah and function
//

   void FzDecision::Z_AND( )

 {
  for( int j=0; j<VECMAX; j++ )
    if( wrk[j] < final[j] )
      final[j]= wrk[j];
  return;
 } ;     // Block: 130
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//              Zadah or function
//

   void FzDecision::Z_OR( )

 {
  for( int j=0; j<VECMAX; j++ )
    if( wrk[j] > final[j] )
      final[j]= wrk[j];
  return;
 } ;     // Block: 131
//              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//              Need a function to check the degree of truth of the "final" vector
//              for use with the setting of default consequent fuzzy sets.
//

   int FzDecision::Z_Truth( )

 {
  int z_truth=FALSE;
```

```
for( int j=0; j<VECMAX; j++ )
  if( final[j] > 0.001 )
    {
    z_truth= TRUE ;
    return z_truth ;
    }
return z_truth ;
} ;     // Block: 132
```

```
//      +++++ Fuzzy Hedge Algorythms ( Hedge_ ) +++++

    class Hedge_

 {
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
// Com 43   Class Defines Thecharacteristics Of Hedges.
//          Hedge_ are modifiers to members of a fuzzy variable's set.  They
//          are elements of the FzVariable class.  They are stored as linked
//          lists.
//
//          Hedges can be adders, factors or employ a power law.
//
//          Member functions include:
//
//          Overloaded input and output operators; operator>> & operator<<
//          and a transformation function; Tran which is used to perform the
//          fuzzy level modification.
//
//          Input format is as follows:
//
//          hedge
//          SIGNIFICANTLY power 2.0
//
//          Keyword is "hedge " (keyed on in Variable_Setup of FzVariable)
//          The semantic modifier "SIGNIFICANTLY" uses a power law with
//          the exponent set to 2.0.
//
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  public:
//
//          Assume all hedges are public...
//
  char *ID;
  double adder, factor, expon;
  Hedge_ *next;

                        Hedge_( Hedge_ *p );
  void                  Tran( FzSet&);
  friend istream&       operator>>(istream& in, Hedge_& t);
  friend ostream&       operator<<(ostream& out, Hedge_& t);
 } ;     // Block: 133
//          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```