Harbour Mini Rules Engine

1. Introduction

1.1 What is a rules engine?

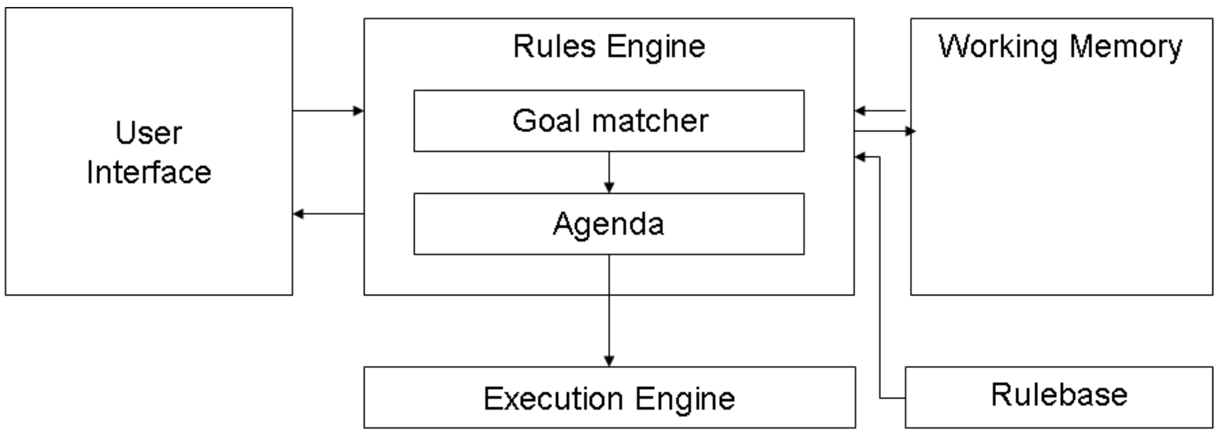
A rules engine is a platform where business rules can be managed apart from the processes where they are used. This mini engine focuses on computations and derivations.

An important characteristic of a rules engine is that rules stand on their own, apart from their usage. The behaviour of a rule is dependent on the state of the system for which the rule has been defined. This state is determined by all values of all objects within that system at a certain time, we talk about the state of the objectmodel.

The added value of a rules engine lies in the separation of rules, they can be managed without the software need to be adjusted. The maintenance of rules can be done in Excel. The added value increases when rules are complex and subject to change, for example when they have to deal with changing markets or changing legislation. Rules can be adjusted and implemented fast.

1.2 Structure rules engine

The following figure depicts the structure of a rules engine.



The rules engine encompasses the following components:

- the rules engine itself
  The rules engine orchestrates the communication between the surrounding components
- the goal matcher
  The goal matcher selects candidate rules from the rulebase, based on the fact to be calculated, and puts them in the agenda.
- the agenda
  The agenda contains the list of rules to be evaluated.
- The execution engine
  This engine evaluates the rules. Each rule consists of a condition and an action pair. The evaluation of a condition always results in true or false, while the evaluation of an action always results in a value.
- the working memory
  The business objectmodel is stored into the working memory, together with additional facts.
- the rulebase
  The rulebase contains rule sets.
- the user interface
  The user interface is used to provide facts that can not be derived or found in the factset.

1.3 How does the rules engine work?

The rules engine starts working when the function Evaluate is called. The goal, the fact to be calculated, the composed ruleset en the factset are passed as parameters. The factset contains additional facts like a calculation date.
Before the function Evaluate is called, the required objectmodel has to be constructed.
In this basic version of the rules engine the objectmodel is referenced directly from the working memory.

The goal matcher selects candidate rules from the rulebase, based on the fact to be calculated, and puts them in the agenda. Before the first condition can be evaluated, the engine looks up the facts that are used within the condition in the factset. A fact is recognizable by the name, that starts with an underscore.
When a fact has not been found, that fact becomes a new goal en this process is started again recursively.
When all facts within the condition have a value, the condition is passed to the execution engine that evaluates the condition. When the condition evaluates to true, the action part of the rule is evaluated. Also the action part can contain facts that have to be looked up or derived first.
When the first condition evaluates to false, the second one is evaluated and so on.

When all condition within an agenda evaluate to false, the fact can not be derived by any action part and the user has to provide a value via the user interface. Of course this situation may not occur in a production environment.

1.4 Harbour Mini Rules Engine

1.4.1 Harbour

The Harbour Project is a free open source software effort to build a multiplatform Clipper language compiler. Harbour consists of the xBase language compiler and the runtime libraries with different terminal plug-ins and different databases.
Clipper is out-of-date for many years, but one way or another for a large community the Clipper language never became out of sight. Clipper has been used successfully for many solid applications that ran for years. The charm of Clipper was its simplicity. However Clipper was a 16-bits compiler and Clipper didn’t support a graphical user interface. Harbour is a modern 32/64-bits cross-platform compiler that supports many platforms like Windows, OS/2, GNU/Linux and Mac OS X.
Harbour also offers extensions for object oriented programming and due to its open architecture many third-party product like HMG (Harbour MiniGui) can be used to develop effective graphical applications in a fast way. Harbour can be used to make open source applications, free or commercial products.

Harbour is a so called pre-compiler, the sourcecode written in Clipper is translated to C.
For the translation from C to machine code you might use an open source compiled like MinGW as well.

1.4.2 MiniRE

Why a *mini* engine? The mini engine aims to evaluate computation rules and conditional computation rules using a backward chaining mechanism, nothing more. The rules don't need priorities and they may be specified in any order.
But of course you still have to set up your rules in a proper way. Rules may not conflict with each other, and conditions must be mutual exclusive.
The mini engine cannot iterate over collections and, in my opinion, doesn't have to. Within the concept of the mini concept process logic and computation rules are separated.

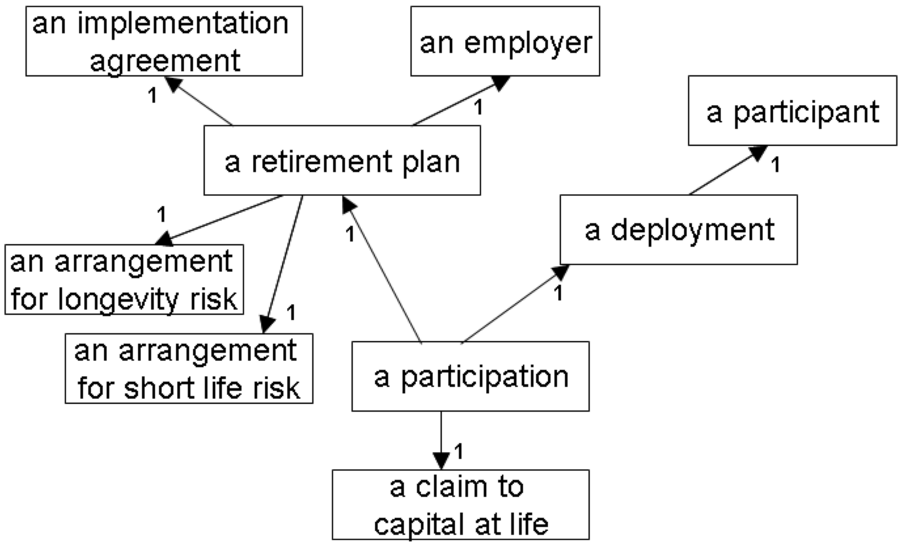
2. Usage of the rules engine

The objectmodel plays an important role within the concept of the mini engine. With the term objectmodel we really mean an UML objectmodel or UML object instancemodel.
An UML objectmodel contains concrete instances of UML classes, called objects, that have 1-to-1 links with each other. An analysis model shows abstractions of objects and multiplicity relationships, see 2.2.

2.1 The objectmodel

The objectmodel is a concrete representation of business objects that plays a role within the problem domain. Because an objectmodel shows *concrete* objects, and not *abstractions* of objects, the model is very useful to discuss the required system behaviour with stakeholders.

Example:


an implementation agreement
an employer
a participant
a retirement plan
a deployment
an arrangement for longevity risk
an arrangement for short life risk
a participation
a claim to capital at life
1
1
1
1
1
1
1
1
1

The objectmodel is part of the "working memory", objects "live" in the internal memory of the computer.
Since objects have 1-to-1 links with each other, Objects can be accessed directly, they don't have to be looked up.

Clarification:
The mindset for working with an objectmodel differs from working with for example a relational model. When you park your car somewhere, you don't have to look for your car when you return to your car, you keep your 1-to-1 link with your car all the time. When you return to the street where you parked your car, you don't have to check the registration number of each car in the street to determine whether your car's registration number matches or not, you don't have to search in a relational way.

Rules operate on the objectmodel, and on the facts that have been provided or have been calculated before. Before you fire a rule, first the following activities have to take place:

- load (instantiate) objectmodel
- load additional facts
- load rulesets

The values of object attributes may be considered as facts as well. Environmental factors can be provided as additional facts by means of a so called factset, an array with name/value pairs.

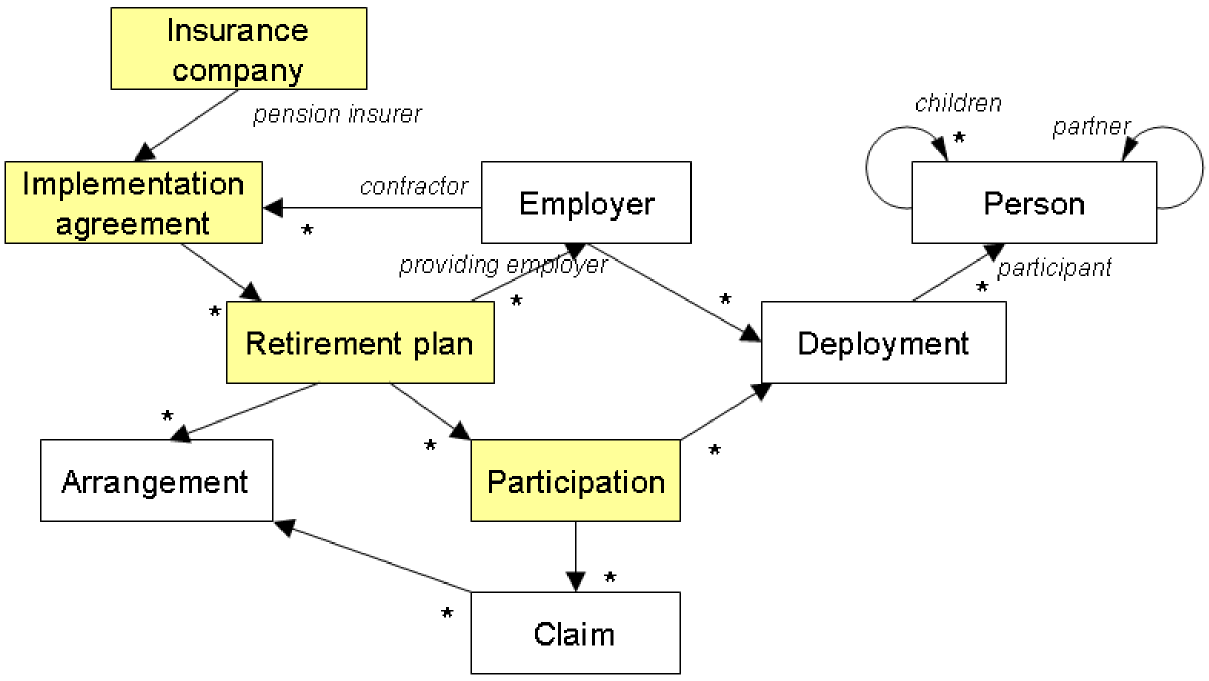
Example factset:

_calculationdate	ctod("01-01-2013")
_premiumvolume	0

The function ctod() is a standard Harbour function that converts a textual and readable date to a date in an internal format.

2.2 The analysismodel

The analysismodel describes classes and their mutual relationships. A UML class is an abstraction of a collection objects that share the same characteristics and behaviour. The analysismodel describes the static structure, but also indicates in which way objects of classes can communicate with each other to realize a certain goal or result.



This analysismodel depicts business objects (classes) for an employer's pension arrangement. The classes insurance company, implementation agreement, retirement plan and participation constitutes a hierarchy. Insurance company represents the insurer who offers pension arrangements to employers. For each accepted pension arrangement an implementation agreement is made up. An implementation agreement can encompass a number of retirement plans, that describes pension conditions in detail. Under a retirement plan the participation of participants is administered.

Also rules are defined on different levels. The highest level is the insurance company level. On that level you define rules on product level. On implementation agreement level you can define rules about the product delivery like rates and reductions.
On retirement plan level you can define rules about which risk covers are mandatory for all employees and which risk covers are optional for a particular employee.
On participation level rules you can define rules that concern choices the participant can make.

The following table shows the relationship between the hierarchical structure of the objectmodel and corresponding rulesets.

Level	Ruleset
Insurance company	Product
Implementation agreement	Implementation agreement
Retirement plan	Retirement plan
Participation	Participation

The analysis model also depicts roles. Within the context a person can be a participant, the insured person, but the person can also play the role of a beneficiary. The beneficiary can be the partner or their children. The circular arrows show these relationships.

2.3 State transitions

The behaviour of a rule, whether a rule must be evaluated or not, depends on the actual state of the system. The state of the system is determined by all values of all relevant attributes of the objectmodel.
Events can cause state transitions, just like the time. State transitions caused by the time are called automated state transitions.

Examples of state transitions caused by the time:

- A young employee reaches the minimum participation age for the pension arrangement
- A participant reaches the retirement age

Examples of state transitions caused by events:

- Employment of a new employee
- Raised salary of a participant
- End of service of a participant (transition from active to inactive participant)

The condition part of a rule describes the state that is conditional for execution of the action part.
So it can be that during a prolongation run for January the contribution premium for a young participant doesn't have to be calculated, while one month later the participant has reached the minimum age and the contribution premium must be calculated.

Tip:
A Handy way to analyse the relevant state transitions the system has to cope with is to make a lifecycle analysis of an object like participant.

Within the context of the example a young employee can have the following relevant states during his employment: candidate participant, active (paying) participant, inactive (not paying) participant and retiree.
Also events like a marriage, a divorce, getting children and earlier retirement cause state transitions that can impact the pension arrangement for a participant.

2.4 Writing business rules

Rules are written in Harbour, the syntax is identical to Clipper.
Rules can be captured and managed using Excel, they are saved as a .dbf file in rulebase.dbf.

Arithmetic operators	+, -, *, / en **
	.and., .or. (,), en .not.
Assignment	:=
Relational operators	=, ==, <>, >, >=, <, <=

Fact	_<name>
Object attribute	<name object>:<name attribute>

Functions
- Get<name>
- ctod
- Round
- IsEmpty
- IsNotEmpty

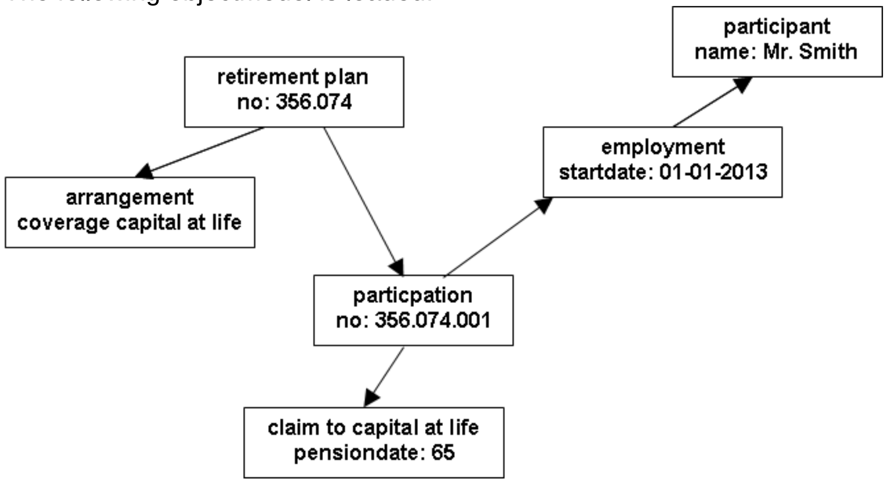
Conditions
- the name of a fact has to start with an underscore (_)
- a condition must be an expression that evaluates to true or false
- an action must be an expression that results in a value
- a self defined function must start with ‘Get’, followed by a name
- a condition may not exceed 254 characters
- an action may not exceed 254 characters

2.5 Example standard usage

In this example the requested goal is “calculate _contributionpremium” for the participant Mr. Smith.

For this request the rules engine needs objectmodel data about the participant (his birthdate, his employment (startdate, fulltime annualsalary, parttime percentage), the retirementplan his participation belongs to (choices and agreements about product delivery), and data about his own participation (startdate).

The following objectmodel is loaded:



The following rulesets are loaded:
- ruleset for rules on participation level
- ruleset for rules on retirementplan level

The factset is filled with:

- fact _calculationdate with value 01-01-2013

Then the rule is fired, in Harbour code:

```
LoadRuleset(ruleset, “Retirementplan”)
LoadRuleset(ruleset, “Participation”)
AddFact(factset, {“_calculationdate”, ctod(“01-01-2013”)})
contributionpremium := Evaluate(“_contributionpremium”, ruleset, factset)
```

The next rules are executed by the rules engine:

LEVEL	CONDITION	ACTION
Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)
Participation	.not. (_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage)	_contributionpremium := 0

Participation	_calculationdate >= participation:employment:startdate .and. (IsEmpty(participation:employment:enddate) .or. _calculationdate < participation:employment:enddate)	_hasEmployment := "Y"
Participation	.not. (_calculationdate >= participation:employment:startdate .and. (IsEmpty(participation:employment:enddate) .or. _calculationdate < participation:employment:enddate))	_hasEmployment := "N"

Participation	1 = 1	_participantage := GetAge(retirementplan:startdate, participant:birthdate)

Participation	retirementplan:choiceContributionScheme == "No adaption" .or. retirementplan:choiceContributionScheme == "Percentage of scheme"	_percentageContributionScheme := retirementplan:percentageContributionScheme * GetContributionPercentage(_calculationyear, _numberContributionScheme, retirementplan:rateContributionScheme, _participantage) / 100
Participation	retirementplan:choiceContributionScheme == "Fixed percentage"	_percentageContributionScheme := retirementplan:percentageFixedContribution
Participation	retirementplan:choiceContributionScheme == "Choice percentage"	_percentageContributionScheme := retirementplan:percentageContributionScheme * GetContributionPercentage(_calculationyear, _numberContributionScheme, retirementplan:rateContributionScheme, _participantage) / 2.25

Retirementplan	1 = 1	_calculationyear := GetYear(_calculationdate)

Retirementplan	retirementplan:productcode = "UnitLinked"	_numberContributionScheme := 3
Retirementplan	retirementplan:productcode = "Traditional"	_numberContributionScheme := 2

Participation	retirementplan:hasAdditionalArrangement = "N"	_pensionbase := GetMax(0, _amountPensionIncome - _amountStatePension)

Participation	retirementplan:hasAdditionalArrangement = "Y"	_pensionbase := GetMax(0, _amountPensionIncome - retirementplan:amountMinimumIncome)

Participation	1 = 1	_amountPensionIncome := participation:employment:annualsalary

Retirementplan	retirementplan:choiceStatePension == "Nominal amount"	_amountStatePension := retirementplan:amountFixed
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP unmarried"	_amountStatePension := 18738
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP married single"	_amountStatePension := 12898
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP married double"	_amountStatePension := 25796

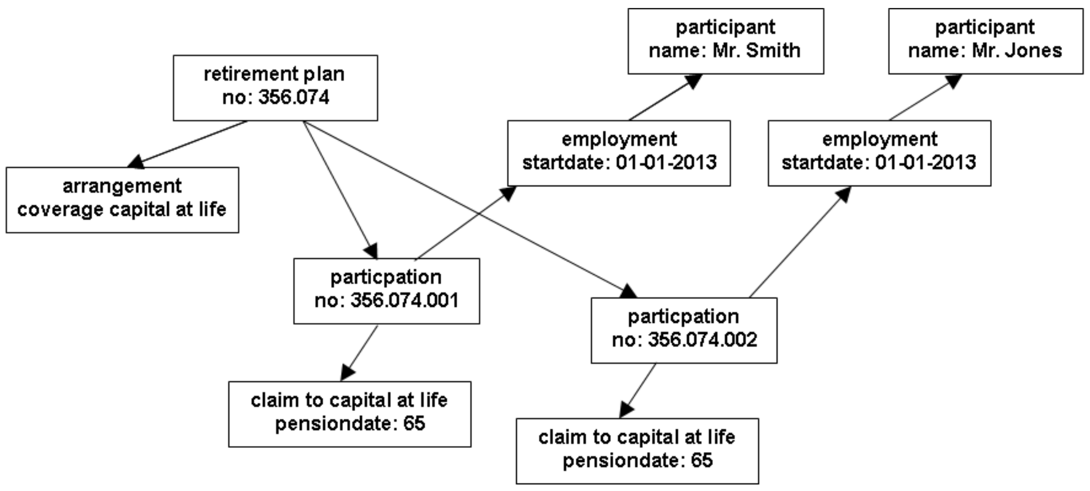
Note that intermediate results are stored in the factset, but after each calculation the engine returns the calculated fact and resets the factset with in this case only the calculationdate.

When a fact isn’t known yet (cannot be found in the factset), that fact becomes a new subgoal and the rules engine calculates the value for that fact first. At the end these rules deliver the desired result. Read more about the mechanism in chapter 3.

2.6 Example usage rules engine for iterating over collections

Now the requested goal is “calculate _premiumvolume” for all participants of the retirementplan.

The strategy is to repeat the calculation request of the previous example for each participant. Suppose we only have two participants, Mr. Smith and Mr Jones, then the objectmodel looks like this:



Note that now the arrow of participation to retirementplan points to the opposite direction. This is because we need a different objectmodel traverse strategy, starting from the retirementplan instead of starting from a given participation.

De relevant Harbour code is as follows:

```
LoadRuleset(ruleset, “Retirementplan”)
LoadRuleset(ruleset, “Participation”)
AddFact(factset, {“_calculationdate”, ctod(“01-01-2013”)})
AddFact(factset, {“_premiumvolume”, 0})
for n := 1 to len(retirementplan:participations)
        participation := pensioenplan:participations[n]
        participant := participation:employment:participant
        Evaluate("_premiumvolume", ruleset, factset)
next
premiumvolume := GetFact(factset, "_premiumvolume")
```

In advance the fact _premiumvolume is stored into the factset with the initial value 0.
The code iterates over the participations of the retirementplan. For each participation the public variables participation and participant are filled, so the rules engine knows which is the current participation and participant. Then the rules engines is activated by the function Evaluate. You might capture the premiumvalue from the returnvalue of the function Evaluate() each time, but because the premiumvolume has been loaded as a fact in advance, the value can be retrieved with the function GetFact() as well.

Use of shortcuts:
Notice the assignments participation := retirementplan:participations[n] and participant := participation:employment:participant. These assignments make shortcuts to the long paths retirementplan:participations[n] and participation:employment:participant repectively.
A shortcut may be used by rules, but beware the shortcut must be declared public, because otherwise the rules engine cannot solve the reference to the shortcut. And also beware that the shortcut points to the right 'n'.

Tip for analysts:
As an analyst you don't need knowledge about how to construct an objectmodel from a relational model or so, objects are just there, you don't have to worry about their construction and how the objectmodel maps on a relation model or other model.

Tip for programmers:
A programmer does need knowledge about the mapping of the objectmodel on the database.
In most cases the objectmodel is constructed in the working memory from a relational database. Therefore you might use a method like Retirementplans():Get(id, retirementplan), where Retirementplans() creates an object of the collectionclass Retirementplans that represent a database table. The class Retirementplans can offer standard CRUD-methods like Find(), Get() and Save(). In this example the method Get() returns the address of a filled retirementplan object with the given id. The method Get() might look up and fill the underlying memberobjects of a retirementplan as well (arrangement and participations), including their parentobjects (for participation employment and participant)

The rule for the calculation of the fact _premiumvolume is:

Retirementplan	1 = 1	_premiumvolume := _premiumvolume + _contributionpremium

Each time this rule is called, the value of the fact _contributionpremium is added to _premiumvolume. Since the value of _contributionpremium is not know in the facset, that fact becomes a new subgoal and is calculated first, as described in the previous example.

2.7 Maintenance of rules using Excel

The rulebase is a .dbf file (rulebase.dbf) that can be maintained in Excel.
When you are specifying or modifying rules, it is handy to save the file as an Excel worksheet temporary.

When you are done, follow the following procedure:

- Save file as .csv file
- Adjust column headers:

  column "LEVEL" must have a columwidth 10
  column "CONDITION" must have a columwidth 254
  column "ACTION" must have a columwidth 254

- Selects the rows and columns, inclusing the headers, and save selected area as .dbf file (rulebase.dbf), ignore the warnings.

2.8 Logfile

The logging of rules can be configured when needed.

The logfile logs every condtion and action and shows how the rules engine works. You might also use the rules engine to improve the performance.

For the first given example the logfile shows:

```
condition ctod("01-01-2013") >= participation:employment:startdate := .T.
condition IsEmpty(participation:employment:enddate) := .T.
action _hasEmployment := "Y" := Y
condition "Y" = "Y" := .T.
condition 1 = 1 := .T.
action _participantage := GetAge(pensionplan:startdate, participant:birthdate) := 53.00000000
condition 53.00000000 >= pensionplan:minimumAgeArrangement := .T.
condition 53.00000000  < participation:GetClaim("CAL"):retirementage := .T.
condition pensionplan:choiceContributionScheme == "No adaption" := .T.
condition 1 = 1 := .T.
action _calculationyear := GetYear(ctod("01-01-2013")) := 2013
condition pensionplan:productcode = "UnitLinked" := .F.
condition pensionplan:productcode = "Traditional" := .T.
action _numberContributionScheme := 2 := 2
action _percentageContributionScheme := pensionplan:percentageContributionScheme *
GetContributionPercentage(2013, 2,  pensionplan:percentageInterest, 53.00000000) / 100 := 20.00
condition pensionplan:hasAdditionalArrangement = "N" := .T.
condition 1 = 1 := .T.
action _amountPensionIncome := participation:employment:yearsalary := 50000
condition pensionplan:choiceStatePension == "Nominal amount" := .T.
action _amountStatePension := pensionplan:amountFixed := 14000
action _pensionbase := GetMax(0, 50000 - 14000) := 36000
action _contributionpremium := round((20.00 * 36000 * participation:employment:parttimepercentage /
10000) / 12, 2) := 600.00
```

Tip
As stated before during iterating over a collection the rules engine starts every time with a ‘blank sheet’, the rules engine resets the factset except the facts you loaded yourself. So to preserve that certain facts are recalculated over and over again, you might evaluate them separately and put them in the factset yourself.

3. MiniRE at work

This chapters describes how the rules engine Works in detail, using the first example again.

Given:

Facts:
_calculationdate:= 01-01-2013

Objectmodel:

Retirementplan:
```
   retirementplan:startdate := ctod("01-01-2013")
   retirementplan:productcode := "Traditional"
```

```
  retirementplan:choiceContributionScheme := "No adaption"
  retirementplan:percentageContributionScheme := 100
  retirementplan:rateContributionScheme := 3
  retirementplan:choiceStatePension := "Nominal amount"
  retirementplan:hasAdditionalArrangement := "N"
  retirementplan:minimumAgeArrangement := 21
Participant:
  participant:name := "Smith"
  participant:birthdate := ctod("01-01-1960")
Employment:
  employment:startdate := ctod("01-01-2013")
  employment:enddate := ctod("  -  -   ")
  employment:annualsalary := 50000
  employment:parttimepercentage := 100
Claim:
  claim:pensiontype := "CAL" (capital-at-life)
  claim:retirementage := 65
```

Goal:
Evaluate(_contributionpremium, ruleset, factset)

The goal matcher selects the following rules and puts them in the agenda.

LEVEL	CONDITION	ACTION
Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)
Participation	.not. (_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage)	_contributionpremium := 0

Tip:
For each else-part of a condition you have to specify a separate rule. For complex conditions it is recommended to make a .not (…) condition with the then-part within the parentheses.

The rules engine reads the first rule of the agenda.

Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)

Before the rules engine can evaluate the condition-part, first the values of the facts _hasEmployment and _particpantage have to be determined, starting with _hasEmployment.
First the engine looks up the fact in the factset, where it is not found. This means that the value of the fact _hasEmployment must be calculated. The calculation of the fact _hasEmployment becomes a new subgoal and the process is repeated recursively. A new subagenda is opened where the goal matcher places the following rules.

Participation	_calculationdate >= participation:employment:startdate .and. (IsEmpty(participation:employment:enddate) .or. _calculationdate < participation:employment:enddate)	_hasEmployment := "Y"
Participation	.not. (_calculationdate >= participation:employment:startdate .and. (IsEmpty(participation:employment:enddate) .or. _calculationdate < participation:employment:enddate))	_hasEmployment := "N"

For the evaluation of the first condition the value of the fact _calculationdate has to be determined. This fact is provided by the user and stored in the factset, which means that the first part of the condition can be evaluated immediately. The comparison 01-01-2013 >= 01-01-2013 evaluated to true. Since the First condition is followed by an .and. operator, the second part of the condition has to be evaluated as well. The function IsEmpty(participation:employment:enddate) returns true, which means that the condition part after the .or. operator isn't relevant anymore.
So the whole condition evaluates to true, which means that the action part can be evaluated, _hasEmployment gets the value "Y". The goal has been reached and the fact is stored in the factset.
The second rule doesn't have to be evaluated anymore and the rules engine returns to:

Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)

Now the fact _hasEmployment can be replaced by its value and the comparison "Y" = "Y" gives true. Because of the .and. operators also the second and third part of the condition have to be evaluated, starting with the second one. Therefore the value of the fact _participantage must be determined. The value cannot be found in the factset yet, so the calculation becomes a new subgoal with the following agenda.

Participation	1 = 1	_participantage := GetAge(retirementplan:startdate, participant:birthdate)

The condtion 1 = 1 always gives true. This condition is used for unconditional computation rules. The age is calculated by the function GetAge(), the function returns value 53.

Again we go back to the previous rule:

Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)

The second comparions _participantage >= retirementplan:minimumAgeArrangement is substituted by 53 >= 21, that gives true, so now the third part must be evaluated.
The fact _participantage can be retreived from the factset and de comparison can be made at once. _participantage < participation:GetClaim("CAL"):retirementage is substituted by 53 < 65 that results in true.

Note:
The method GetClaim() selects one claim of one pensiontype from the collection claims.
The usage of the method here makes the rule more future proof than making use of fixed 1-to-1 links to claimobjects for "capital-at-life" and "capital-at-death".

Because the whole condition evaluates to true, the action part is evaluated.
The action part has two facts, percentageContributionScheme and _pensionbase, their values have to be calculated first.

For _percentageContributionScheme the following agenda is made up:

Participation	retirementplan:choiceContributionScheme == "No adaption" .or. retirementplan:choiceContributionScheme == "Percentage of scheme"	_percentageContributionScheme := retirementplan:percentageContributionScheme * GetContributionPercentage(_calculationyear, _numberContributionScheme, retirementplan:rateContributionScheme, _participantage) / 100
Participation	retirementplan:choiceContributionScheme == "Fixed percentage"	_percentageContributionScheme := retirementplan:percentageFixedContribution
Participation	retirementplan:choiceContributionScheme == "Choice percentage"	_percentageContributionScheme := retirementplan:percentageContributionScheme * GetContributionPercentage(_calculationyear, _numberContributionScheme, retirementplan:rateContributionScheme, _participantage) / 2.25

The First part of the first condition evaluates to true, which implies we are done and the action part can be evaluated. The action part has two facts that have to be solved first.

For _calculationyear the following rule is selected.

Retirementplan	1 = 1	_calculationyear := GetYear(_calculationdate)

This rule results in the value 2013.

For _numberContributionScheme the following rules are selected.

Retirementplan	retirementplan:productcode = "UnitLinked"	_numberContributionScheme := 3
Retirementplan	retirementplan:productcode = "Traditional"	_numberContributionScheme := 2

In this case the condition of the first rule evaluates to false, but the condition of the second rule gives true, so the fact _numberContributionScheme gets the value 2.

Back to:

Participation	retirementplan:choiceContributionScheme == "No adaption" .or. retirementplan:choiceContributionScheme == "Percentage of scheme"	_percentageContributionScheme := retirementplan:percentageContributionScheme * GetContributionPercentage(_calculationyear, _numberContributionScheme, retirementplan:rateContributionScheme, _participantage) / 100

Now the _percentageContributionScheme can be calculated, using the following table:

Year	Rate	From	Until	I	II	III
2013	3	18	19	4,3	5,2	6,0
2013	3	20	24	5,0	6,0	6,9
2013	3	25	29	6,1	7,3	8,3
2013	3	30	34	7,4	8,9	10,0
2013	3	35	39	9,0	10,9	12,1
2013	3	40	44	11,0	13,3	14,6
2013	3	45	49	13,4	16,3	17,7
2013	3	50	54	16,5	20,0	21,4
2013	3	55	59	20,4	24,8	26,0
2013	3	60	99	25,5	31,1	31,7

So _percentageContributionScheme := pensionplan:percentageContributionScheme * GetContributionPercentage(2013, 2, pensionplan:percentageInterest, 53.00000000) / 100 gives value 20.

With this result the rules engines returns to:

Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)

The circle isn't round yet, cause the value of _pensionbase has to be calculated First.
But to keep it short, we only show the agenda's needed for the calculation of the underlying facts.

Participation	retirementplan:hasAdditionalArrangement = "N"	_pensionbase := GetMax(0, _amountPensionIncome - _amountStatePension)
Participation	retirementplan:hasAdditionalArrangement = "Y"	_pensionbase := GetMax(0, _amountPensionIncome - retirementplan:amountMinimumIncome)

Participation	1 = 1	_amountPensionIncome := participation:employment:annualsalary

Retirementplan	retirementplan:choiceStatePension == "Nominal amount"	_amountStatePension := retirementplan:amountFixed
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP unmarried"	_amountStatePension := 18738
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP married single"	_amountStatePension := 12898
Retirementplan	retirementplan:choiceStatePension == "10/7 x SP married double"	_amountStatePension := 25796

Given the values in the objectmodel, the calculation of the facts is as follows.

_amountPensionIncome := participation:employment:yearsalary := 50000

_amountStatePension := pensionplan:amountFixed := 14000

_pensionbase := GetMax(0, 50000 - 14000) := 36000

Again back to:

Participation	_hasEmployment = "Y" .and. _participantage >= retirementplan:minimumAgeArrangement .and. _participantage < participation:GetClaim("CAL"):retirementage	_contributionpremium := round((_percentageContributionScheme * _pensionbase * participation:employment:parttimepercentage / 10000) / 12, 2)

Now we can complete the calculation.

_contributionpremium := round((20.00 * 36000 * participation:employment:parttimepercentage / 10000) / 12, 2) := 600.00

3.1 Iterate over collections

Rules must must be self supporting and may not be dependent on any process where they are used. Process logic and rules are separated, iterate over a collection is process logic.

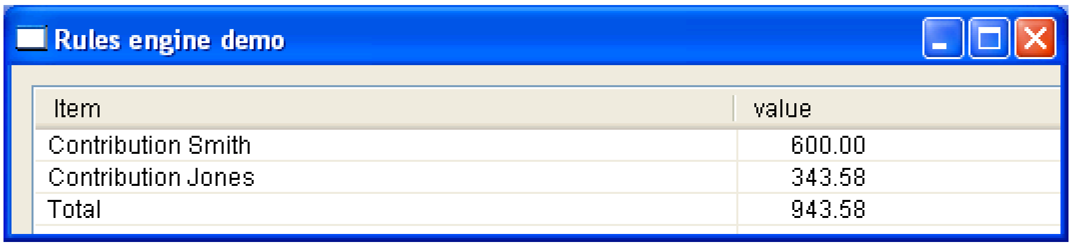
For the calculation of the premiumvolume for a proposal of a pension arrangement with 2 participants, the contributionpremiums of the participants have to be summed.

Traverse strategy in Harbour code:

```
LoadRuleset(ruleset, “Retirementplan”)
LoadRuleset(ruleset, “Participation”)
AddFact(factset, {“_calculationdate”, ctod(“01-01-2013”)})
AddFact(factset, {“_premiumvolume”, 0})
for n := 1 to len(retirementplan:participations)
        participation := retirementplan:participations[n]
        participant := participation:employment:participant
        Evaluate(“_premiumvolume”, ruleset, factset)
next
premiumvolume := GetFact(factset, “_premiumvolume”)
```

Deelname	1 = 1	_Premievolume := _premievolume + _beleggingspremie

Result:



Item	value
Contribution Smith	600.00
Contribution Jones	343.58
Total	943.58

It could be that in *another* process the premiumvolume has to be calculated in *another* way. In that case the same rules can be used.

In the real life application where this demo application has been based on, the customer gets quantity discount that depends on the number of participants within the implementation agreement.
The discountpercentage is derived from the total premiumvolume of all retirementplans within the implementation agreement.

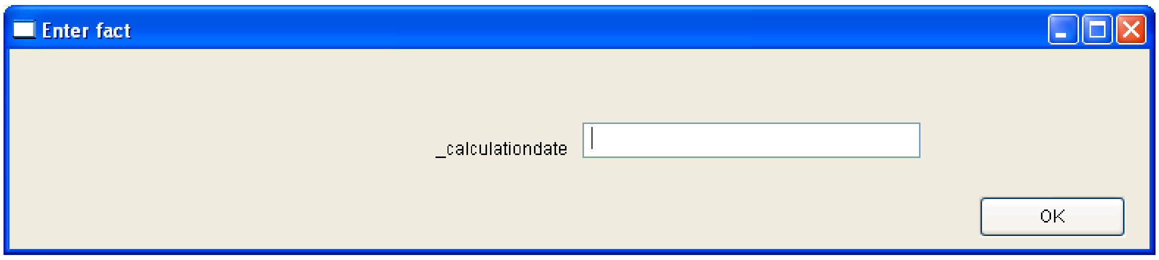
Then the traverse strategy to compute the premiumvolume should be as follows:

```
LoadRuleset(ruleset, “Retirementplan”)
LoadRuleset(ruleset, “Participation”)
AddFact(factset, {“_calculationdate”, ctod(“01-01-2013”)})
AddFact(factset, {“_premiumvolume”, 0})
for n := 1 to len(implementationagreement:retirementplans)
        retirementplan := implementationagreement:retirementplans[n]
        for m := 1 to len(retirementplan:participations)
                participation := retirementplan:participations[n]
                participant :=  participation:employment:participant
                Evaluate(“_premiumvolume”, ruleset, factset)
        next
next
premiumvolume := GetFact(factset, “_premiumvolume”)
```

Note:
For each arbitrary traverse strategy you need a specific subset of the objectmodel.

3.2 Alternative flow

When no rules for a fact are found or all conditions within the agenda evaluate to false, the user will be asked to enter the fact value.



_calculationdate

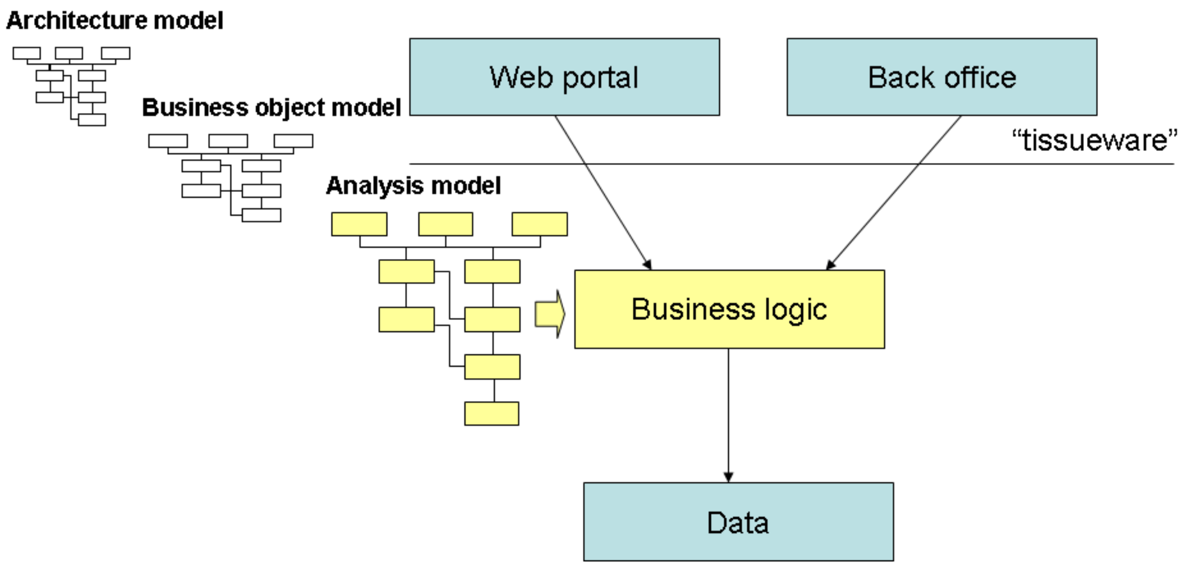
The user must enter a valid value, that depends on the datatype of the fact.

Datatype	Value
character	a characterstring within single or doube quotes
number	a number with optionally a decimal point
date	ctod(“dd-mm-eejj”)
logical	.T. .or .F.

Note that a date is entered as a characterstring within the function ctod (character-to-date)

4. Underlying concepts

4.1 Architecture multi-tier applications



Business rules are about business logic in the central layer. This layer is represented by the objectmodel. Busiess logic is enclosed by presentation channels like a web portal or a back office front-end application. Today you might use an iPhone app as well.
Organizations and processes (workflows) change in time, and together with them the presentation applications in the front-end. Therefore they are depicted as temporary tissueware.

The business layer and business rules (except process rules, see the next chapter) are more stable and future proof in time. When rules are set up properly, rules for a new product can be added easily.

4.2 Business rule types

Business rules can be categorized as follows:

- process rules
- data constraints

- unconditional computations
- conditional computations

Process rules, also called workflow or action rules, describe under which circumstances a process flow or a subflow has to be started. They live in the presentation layer. Many workflow engines enable the user to configure processes using this type of rules.

Data constraints, or validation rules, project the integrity of the data. Validation rules must be applied in the presentation layer, but since the presentation software is volatile, also in the business layer. In the old days, validation rules were applied in de data layer as stored procedures, but that idea brought in vendor dependency.
It is handy to distinguish the following types of validation rules:

1. validation of a single field, i.e. field required
2. fields in relation with each other, i.e. enddate must be beyond startdate
3. key validations, i.e. a new object must have a unique key
4. referential integrity, i.e. a participation cannot exist without a retirementplan

The rules engine covers the unconditional an the conditional business rules, the latter also called inference rules, using a backward chaining mechanism.

4.3 Design patterns

In this paper the separation of rules has already been emphasized. In fact this is the key to a successful implementation of the rules

Many rules engine offer the possibility to work with priorities of rules and many engines also support iterating over collections. In my opinion this is in conflict with principles for business rules, and you don’t’ want “programming” business rules within the engine, but outside the engine, because program logic is volatile business logic.

The book “Design patterns” describes 2 patterns to separate business logic, the strategy pattern and the visitor pattern.

4.4 Backward chaining vs forward chaining

For those who are new to rule chaining, the advantages and disadvantages of forward and backward chaining can be confusing. Therefore a quick definition of the terms.

- Forward chaining - When one or more conditions are shared between rules, they are considered "chained." Chaining refers to sharing conditions between rules, so that the same condition is evaluated once for all rules.
- Backward chaining - Backward chaining is very similar to forward chaining with one difference. Backward chaining engines query for new facts, whereas forward chaining relies on the application asserting facts to the rule engine. Backward chaining rule engines will implicitly create subgoals and use those subgoals to execute queries.

One of the benefits of backward chaining is the user doesn't have to explicitly write rules for the subgoals. The rule engine implicitly generate the subgoal for each object. This is generally done when the rules are loaded into the rule engine. As the conditions of each subgoal is met, a backward chaining rule engine will execute additional queries