**Maximo Business Rules Scripting**

MBR provides a cloud safe way to extend Maximo business logic. MBR is an interpreted language specifically developed for Maximo. MBR is one of the languages available in Maximo Scripting and will be the only language available in the Maximo SaaS Public deployment. While JS and Python provide great functionality and flexibility, MBR provides a more sandboxed scripting which can benefit Maximo deployments that are shared between multiple customers.

In the initial release MBR will only support scripting points for Object (Mbo), attribute, conditions and actions. Support will be available also for library scripts. Anyone familiar with scripting would just need to learn the syntax of this new language to be productive. Below we describe the fundamentals of this scripting language.

MBR code expressions are new line delimited. This means each line will contain zero or one MBR expression. MBR does not support an expression spanning multiple lines. MBR code allows comments with expressions prefixed by # character. MBR code allows defining functions with expressions prefixed with: character. MBR code get syntax validated at save time. Some of the restrictions imposed by design on this language are listed below.

* There is no recursion allowed in this language. This implies we cannot call to a function or script and then call them again in the same call stack.
* The looping is only limited to MboSets and tokens (delimited strings). MboSet looping is restricted by the maximum fetch limit configured in Maximo.
* No direct access to any java class is allowed. All access go through the language provided functions and operators.

Note that currently there is no support for I/O interaction with MBR. Also inline comments – like the one shown below - are not allowed:

newmbo("opressure","assetmeter") #this is not allowed – put the comment in a new line

**Supported Operators**

**Mathematical Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Additive operator |
| - | Subtraction operator |
| \* | Multiplication operator |
| / | Division operator |
| % | Remainder operator (Modulo) |
| ^ | Power operator |

**Boolean Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equals |
| == | Equals |
| != | Not equals |
| <> | Not equals |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| && | Boolean and |
| || | Boolean or |

Note that Boolean “not” is implemented by a function and not using the traditional “!” operator.

**Supported Functions**

**Common Mathematical Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| not(*expression*) | Boolean negation, 1 (means true) if the expression is not zero |
| random() | Produces a random number between 0 and 1 |
| min(*e1*,*e2*) | Returns the smaller of both expressions |
| max(*e1*,*e2*) | Returns the bigger of both expressions |
| abs(*e*) | Returns the absolute (non-negative) value of the expression |
| round(*e*) | Rounds a value to a certain number of digits, uses the current rounding mode |
| floor(*e*) | Rounds the value down to the nearest integer |
| ceiling (*e*) | Rounds the value up to the nearest integer |
| log(*e*) | Returns the natural logarithm (base e) of an expression |
| log10(*e*) | Returns the common logarithm (base 10) of an expression |
| sqrt(*e*) | Returns the square root of an expression |
| sin(*e*) | Returns the trigonometric sine of an angle (in degrees) |
| cos(*e*) | Returns the trigonometric cosine of an angle (in degrees) |
| tan(*e*) | Returns the trigonometric tangens of an angle (in degrees) |
| asin(*e*) | Returns the angle of asin (in degrees) |
| acos(*e*) | Returns the angle of acos (in degrees) |
| atan(*e*) | Returns the angle of atan (in degrees) |
| sinh(*e*) | Returns the hyperbolic sine of a value |
| cosh(*e*) | Returns the hyperbolic cosine of a value |
| tanh(*e*) | Returns the hyperbolic tangens of a value |
| rad(*e*) | Converts an angle measured in degrees to an approximately equivalent angle measured in radians |
| deg(*e*) | Converts an angle measured in radians to an approximately equivalent angle measured in degrees |
| pct(e1,e2) | Generate the percentage value of expression e1 with respect to e2. |

**Setting and getting variables**

|  |  |
| --- | --- |
| **Function** | **Description** |
| setvar(*varname*, *e[, global]*) | Set or create a variable *varname* with a value of *e*. By default the variable is set at the local scope ie global is false by default. You can pass TRUE for the 3rd parameter (optional) to set it to global scope. |
| getvar(*varname*) | Get the variable *varname* value. If the variable is defined in local and global scope, the local one would be used. |

**Maximo business specific functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| invokescript(*scriptname*) | Invoke another script or a function local to the current script. For invoking local functions the scriptname needs to start with colon (:). |
| newmbo(*varname*, *relation[,global*) | Add a new mbo for the relation *relation*. The owner mbo would be the launch point mbo. The newly created mbo is set to the variable *varname*. The 3rd parameter is optional and defaults to 0 which means that the variable *varname* is local ie going to be visible only in the current scope. |
| setvaluetombo(*varname*,*attrname,e1*) | Set value to a mbo bound to the variable *varname*. The attribute name is identified by *attrname* and the value is an expression *e1*. All values are set to the mbo using the NOACCESSCHECK flag. |
| setvalue(*attrname*, *e1*) | Set value to the launch point mbo. The attribute The attribute name is identified by *attrname* and the value is an expression *e1*. All values are set to the mbo using the NOACCESSCHECK flag. |
| setvaluenull(*attrname*) | Set value null to attribute *attrname* on the launch point mbo. |
| setvaluenulltombo(*varname*,*attrname*) | Set value null to mbo attribute. The mbo is bound to the variable *varname*. |
| error(*errgrp, errkey*) | Throw MXException with the error group *errgrp* and error key *errkey.* This is useful in validation scenarios when we would like to throw an error. Note that the *errgrp* and *errkey* needs to get registered in Maximo messages. |
| setrequired(*attrname, bool*) | Set an attribute to required or not required based on the *bool* value being 1 (TRUE) or 0 (FALSE). |
| setreadonly(*attrname, bool*) | Set an attribute to read only or not required based on the *bool* value being 1 (TRUE) or 0 (FALSE). |
| sethidden(*attrname, bool*) | Set an attribute to read only or not required based on the *bool* value being 1 (TRUE) or 0 (FALSE). |
| deletethismbo(*varname*) | Mark the mbo bound to the variable *varname* for delete. |
| deleteall(*relation*) | Delete all mbos from this related mboset for the relation *relation*. |
| deleteallfromthismbo(varname,*relation*) | Delete all mbos from this related mboset for the relation *relation* from the mbo bound to the variable *varname*. |
| isnullf(e1) | Returns TRUE if the expression e1 evaluates to null. For example if e1 can be an attribute name the expression may look like - isnullf(description) |
| invokeworkflow(*wfname*) | Invokes workflow *wfname* synchronously. |
| maxprop(*propname*) | Returns the value (as a string) for the maximo property *propname.* |
| maxcond(condname) | Evaluates the maximo condition to TRUE or FALSE. |
| tobeadded() | Determines if the context mbo is to be added ie new persistent mbo. Returns TRUE/FALSE. |
| tobesaved() | Determines if the context mbo is to be saved ie a persistent modified mbo. Returns TRUE/FALSE. |
| tobedeleted() | Determines if the context mbo is to be deleted ie a persistent mbo marked for delete. Returns TRUE/FALSE. |
| setevalresult(TRUE/FALSE) | This is used to set the script variable “evalresult” which is used in condition launch points. |
| setthisattrvalue(value) | This is used in the Atribute init value launchpoint to set the current attribute initial value. |
| scriptvar(varName) | This function returns the values for the script implicit vars like “interactive”, “app” etc. The varname is always specified as a string value like if(scriptvar(“app”)==”WOTRACK”..) |

**MboSet aggregation functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| countf(*e1[,e2, e3]*) | Count of a related mboset, where e1 is the relation name from the context mbo. Optional parameters e2 and e3 specify the timeline attribute name and the duration (respectively) upto which we should filter the result set for this operation. |
| avgf(*e1,e2[,e3,e4]*) | Average of an attribute value over a collection of mbos in a related mboset. E1 is the relation name and e2 is the attribute name. e3 (optional) is the timeline attribute name and e4 (optional) is the duration. |
| maxf(*e1,e2[,e3,e4]*) | Similar to avgf and just doing the max of an attribute value over a collection of mbos in a related mboset. |
| minf(*e1,e2[,e3,e4]*) | Similar to avgf and just doing the min of an attribute value over a collection of mbos in a related mboset. |

**Control flow functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| if(*condition*,*value\_if\_true[*,*value\_if\_false]*) | Returns one value if the condition evaluates to true or the other if it evaluates to false |
| foreachmbo(*relname*, *scriptname*[, *cond*]) | Loop through all mbos in a related mboset. This number of mbos that this can iterate on is limited by the max fetch limit. The 2nd parameter specifies the name of the script we want to run for each iteration with the mbo at that index context. The script name can be the name of a local function (starts with a colon char) or a different script. |
| foreachtoken(*e1*, *splitchar*,*scriptname*) | Loop through all the tokens of a string e1 (which can be either raw string or a mbo attribute whose value will be tokenized). The splitchar is the delimiter based on which the e1 will be tokenized. The 3rd parameter specifies the name of the script we want to run for each iteration with the “token” variable holding the token value. The script name can be the name of a local function (starts with a colon char) or a different script. |
| continue() | Continue to the next iteration in the loop. |
| break() | Break from the for loop. |

**String manipulation functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| concat(*e1*, *e2*) | Concat the strings from the 2 expressions e1 and e2. For example say you have asset mbo launch point and your expression is concat(description,assetnum) this will concat description and assetnum together. |
| tolower(*e1*) | Lowercase the string from expression e1. |
| toupper(*e1*) | uppercase the string from expression e1. |
| startswith(*e1*, *e2*) | Returns 1 (TRUE) when the string e1 starts with string e2. |
| endswith(*e1*, *e2*) | Returns 1 (TRUE) when the string *e1* ends with string *e2*. |
| substring(*e1*, *e2[, e3]*) | Substring *e1* starting with index *e2* to end or optionally to end index *e3*. |
| tokenat(*e1,e2,e3)* | Return the token at the index e3 for the string e1 with the separator as e2. |

**Date manipulation functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| now() | Generates current datetime. |
| duration(year,mon,day,hr,min,sec) | Represents a time duration. |
| date(*y,m,d*) | Creates a date object with year (y) month (m) and day(d). |
| datetime(*y,m,d,hh,mm,ss*) | Creates a date object with year (y) month (m) and day(d) and time part included as hh, mm, ss . |

**Other utility functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| nvl(e1,e2) | Returns e1 If e1 is not null, otherwise returns e2. |
| number(e1) | Converts a string e1 to a number. |
| str(e1) | Converts and number to a string. |

**Boolean primitives**

|  |  |
| --- | --- |
| **Token** | **Description** |
| TRUE | Boolean true. |
| FALSE | Boolean false |

**Accessing aspects of mbo attributes**

Mbo attributes can be accessed directly in MBR without needing to define script variables. So you will see in the samples, we refer to mbo attributes like assettype without needing to bind them to a variable (as in py and js scripts). A mbo attribute has lot more notations to support the access to other metadata or related data. Below we show what is supported as of now.

|  |  |
| --- | --- |
| **Notation Example** | **Description** |
| *rel1*$*rel2*$*attr* | This is the way to access attributes from related mbos. In this example it travels 2 relations rel1 and rel2 to get the attribute. |
| *owner$attr* | Access the attribute attr from the owner Mbo. This can be N levels deep – owner1$owner2$attr |
| *modified*$*attr* | Returns TRUE when the attribute attr is modified and FALSE otherwise. |
| *prev*$*attr* | Returns the initial value of the mbo attribute attr. |
| *internal*$*attr* | This will get the internal value of the attribute when the attribute is synonym domain bound. A good example would be if(internal$status="BROKEN",…,..) |

Below is an example of a MBR script that adds 2 meters O-PRESSUR and IN-PRESSUR when a new asset is created with asset type “GASENG”. As expected, we will leverage the Object Launch point scripting with the event being Save->Add->Before Save.

#my first MBR code for asset

setvar("isgaseng",(assettype == "GASENG" && countf("assetmeter")==0))

if(getvar("isgaseng"),invokescript(":createmeters"))

#my MBR function that will add the 2 meter mbos to the asset mbo

:createmeters

newmbo("opressure","assetmeter")

setvaluetombo("opressure","metername","O-PRESSUR")

newmbo("inpressure","assetmeter")

setvaluetombo("inpressure","metername","IN-PRESSUR")

Lets analyze the code above. The code creates a local variable called **isgaseng** and sets the value to 1 (TRUE) is assettype is GASENG and the count of “assetmeter” is 0 ie the asset mbo has no assetmeters defined yet.

Next it checks to see if the isgaseng is TRUE, in which case it invokes the method **createmeters.** Note how the method is defined below with the colon ( : ) char prefixed to the name.

In the method createmeters, we first add a new mbo to the “assetmeter” relation mboset. The addmbo function would also assign that to the local variable opressure. In the next line we use that variable to set values to the newly created mbo. The function we use is setvaluetombo as in this case we have the mbo bound to a variable. If the mbo was the context mbo (in this case since the launchpoint is on Asset, the context mbo would be an asset mbo), we could have just used the setvalue function to set an attribute value to the context mbo. The next 2 lines does the same to create another mbo and set an attribute on that mbo.

Since all of this is done on save event, the Maximo transaction framework takes care of saving and committing the newly created mbos as part of the main transaction.

A small variation of this above script could have been to not have the method createmeters and have it instead as a script (say named as createassetmeters) as maybe that creation of meters is a reusable functionality and hence you want it in a separate script that can be used from multiple other scripts. Such scripts which host reusable code are often referred to as library scripts. The code to call that script would be like below (note that we do not prefix that with a : character as its not a method but a script)

invokescript("createassetmeters")

Now lets look at a attribute launch point. Say we want to validate that no asset of type GASENG can have a purchaseprice > 200. This could have been done as an object launch point ie at the save of the Asset, but it will be more real time if we do it as an attribute launchpoint. For this case we will base the attribute launch point (on validate event) on the purchaseprice attribute of the Asset. The code will be as below

if(not(isnullf(assettype)) && assettype=="GASENG" && not(isnullf(purchaseprice)) && purchaseprice>200,error("asset","toomuchcost"))

It’s a single line code with an **if** control structure. If the assettype is not null and the assettype is GASENG, and the purchaseprice is not null and greater than 200 then we want to throw an error with the error group “asset” and error key as “toomuchcost”. This assumes you have already registered the error group and key with the maximo messages.

As per attribute launch point semantics – this script will get invoked any time the purchaseprice attribute is modified.

Lets do one more on attribute launch point, this time with some date manipulation. Say the use case is – if the asset type is BUS we want to set a default endoflife to be installdate plus 1 year. If the installdate is null we want to add that 1 year from the current date. This is an attribute launch point on assettype attribute and the code will be like below:

if(assettype=="BUS",setvalue("estendoflife",nvl(installdate,now())+duration(1,0,0,0,0,0)))

We use the nvl function here to choose between installdate and now() which gives us the current datetime. We then add a duration of 1 year to it and we set it to the estendoflife attribute.

Say now we want to modify the script to add another rule – if the assettype is BUS – we want to make the priority required and otherwise not. Adding the line below will do that job.

if(assettype=="BUS",setrequired("priority",TRUE),setrequired("priority",FALSE))

Since this entails repeating the if condition check – we can make the code a little better by defining a method and calling it from the **if** statement – shown below:

if(assettype=="BUS",invokescript(":seteol"),setrequired("priority",FALSE))

:seteol

setvalue("estendoflife",nvl(installdate,now())+duration(1,0,0,0,0,0))

setrequired("priority",TRUE)

Lets look at for looping with the next example. Say we want to set the difference between out and in pressure everytime when an asset meter value gets modified. We attach and object launchpoint to Asset (as when the meter value is modified – the asset mbo save is invoked too). The code below shows one way to do it.

foreachmbo("ACTIVEASSETMETER",":pressurediff")

setvalue("priority",getvar("op")-getvar("ip"))

:pressurediff

if(metername=="O-PRESSUR",setvar("op",number(newreading),TRUE))

if(metername=="IN-PRESSUR",setvar("ip",number(newreading),TRUE))

As you can see, the code does a for loop on all the mbos for the relation ACTIVEASSETMETER which is what is used from the “Enter meter readings” dialog in Maximo. Note that we call a method pressurediff (defined in the script with a colon prefixed – to indicate it’s a method). So effectively this method will be called as many times as the number of meters for that asset. As this method is called – the context mbo will be the assetmeter mbo at the loop index. When we compare the metername==”O\_PRESSUR” it’s the metername attribute in the assetmeter mbo at the loop index. As apparent from the code in pressurediff, it sets the “op” and “ip” variables at the global scope (the “global” parameter is set to TRUE). This implies that “op” and “ip” are visible at the script main code. The setvalue call at the main code sets the priority to the difference between “op” and “ip”. Also note that since newreading is a aln attribute, we use the number function to convert a string to number.

Lets take another example as to how we can trap the duplicate event to modify the workorder duplication process. Say the use case is to set a custom attribute “copiedfrom” from the wonum from the source mbo.

To do this we need to name the script like this WORKORDER.DUPLICATE. This will indicate to the scripting framework to call this script when the workorder is duplicated. The scripting framework will provide a global variable named “dupmbo” which holds a reference to the new created duplicate mbo. The context mbo would still be the source mbo. The code is shown below

setvaluetombo("dupmbo","copiedfrom", wonum)

**Support for Yes/No/Cancel**

Yes/No/Cancel is a feature of Maximo that asks the end user a question based on some change in existing state of a Mbo/attribute. Below example shows how we can use MBR language to add such functionality to the Asset priority attribute. This uses the attribute launch point for the attribute priority in Asset. If the priority is 1, the system will ask the user if they want to set the default vendor A0001 and if no, mark the vendor to not required.

if(priority==1,invokescript(":handle\_ync"))  
  
:handle\_ync  
if(isyncdflt(yncuserinput()),invokescript(":dflt"),if(isyncyes(yncuserinput()),invokescript(":yes"),if(isyncno(yncuserinput()),invokescript(":no"))))  
  
:yes  
setvalue("vendor","A0001")  
  
:no  
setrequired("vendor",FALSE)  
  
:dflt  
yncerror("asset", "assetpriority")

You will notice that the function “dflt” launches the YNC interaction by throwing the yncerror – which is a built-in MBR function that takes in the error group and error key as parameters. The error key needs to be of YNC type error as defined in the Messages app in Maximo. And then all we need is to invoke the “dflt” method on the condition that we need – for this example that being priority=1.

Note the usage of the other built-in functions

|  |  |
| --- | --- |
| Function | Usage |
| yncuserinput() | Gets the user input as a numeric value – the user input will be one of YES/NO/CANCEL/OK/DEFAULT |
| Isyncdflt(input) | Is the response a default. |
| isyncyes(input) | Is the response a yes. |
| Isyncno(input) | Is the response a no. |
| Isyncok(input) | Is the response a ok. |
| yncerror(grp,key) | Launches the ync interaction. |

**Support for Warning**

Maximo framework supports showing warning to users. Warnings are generally used to show some information to the end user which the application deems important for the user to know, but not critical enough to fail the user action. We have already covered how to throw errors using MBR. The below example shows how to generate a warning using MBR.

Say for example we want to show a warning to the end user when Purchase Orders are created without adding any lines (countf(“poline”==0). We would use an Object launch point for PO – on save of a new record. The script code will be as below

if(countf("poline")==0,warning("po","nolines"))

Note the use of the MBR function “warning(grp,key)” – very similar to the error function that we described before.