Scripting With Maximo

Anamitra Bhattacharyya [Lead Developer]
Sampath Sriramadhesikan [Lead Designer]

Scripting is a new feature that was introduced in Maximo version 7.5 based on the feedback from the user community who yearned to have a simpler yet effective way of customizing the product without having to go through the pain of system downtime and steep learning curve.

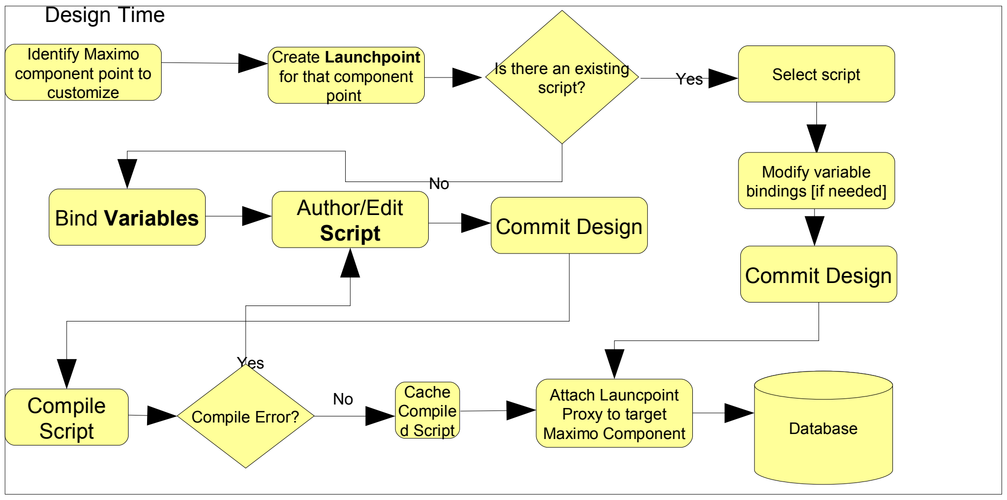
While Maximo provides great many customization points, most of them needs JAVA coding skills to do any meaningful [logic based] customization. It also would often need deep knowledge of Maximo apis as well as the Maximo internals. That can often be a daunting task even for an experienced programmer.

In comes Maximo scripting to help ease some of these concerns. Maximo scripting is primarily based on the JSR 223 specification which is part of JAVA 6. This JSR allows a JAVA application [in this case Maximo] to host script engines which are compliant to this specification. The engines that are supported in an OOTB Maximo 75 are

1. Mozilla Rhino (JavaScript) which ships with the IBM/Oracle(Sun) JDK
2. Jython which is included as part of Maximo

This basically implies that users can use either of these 2 scripting languages to customize Maximo using the Maximo scripting framework. We do understand that there are other popular JSR 223 compliant scripting engines like JRuby/Groovy and it should be fairly simple to add support for these by adding these engines [jars] in Maximo application classpath. The way the scripting framework is written – it should be able to detect those jars from the classpath and show them as available languages in the scripting application. However I would like to mention that at this point Maximo has only been tested with the Rhino-JavaScript and Jython engines and JSR 223 being fairly new, a lot of the “compliant” engines may have potential issues with their implementation which can prevent seamless integration with Maximo.

Now lets get familiar with the different artifacts of the scripting framework. Below is a figure which can give you a visual feel for the design and run-time artifacts.


Design Time
Identify Maximo component point to customize
Create Launchpoint for that component point
Is there an existing script?
Yes
Select script
Modify variable bindings [if needed]
Commit Design
No
Bind Variables
Author/Edit Script
Commit Design
Compile Script
Compile Error?
Yes
No
Cache Compiled Script
Attach Launcpoint Proxy to target Maximo Component
Database

Based on this design flow chart we can see that there are 3 distinct artifacts – Launchpoint, Script and Variables which make up the framework.

Script

First of course is the **script** which is a text file that you can edit inside Maximo or outside of Maximo [in your choice of editors and import back into Maximo]. Every script has an associated attribute – scriptlanguage which helps the run-time figure out the appropriate script engine to invoke for processing the script. The value list for available script languages come from the providing script engines in the classpath. Its common for script engines to provide multiple alias or short names for the language support provided by that engine. For example the jython engine provides 2 names – jython and python – both referring to the same engine/script language. Selecting either one is fine and produces identical behavior. Most engines support script compilation which eventually converts the script to a executable bytecode [for the JVM]. The ones that we support OOTB – JavaScript and Jython both support complied scripts. When the deployer is commiting the design process – in the background the framework would compile and cache the script. This process is often referred to as generating "hot" scripts which are ready to execute. Note if there is a compile failure the process will not commit and the deployer has to fix the script to proceed.

Before we jump further into these artifacts we will look into the application support for the scripting framework.

Script/Launch point Creation

This is done via wizards launched from the list tab in the autoscript application. The autoscript application can be launched from the GoTo → System Configuration → Autoscript menu. In the list tab drop down actions list there are a slew of wizards that lets you

1. Create a vanilla script without any launch point.
2. Create Scripts with Object launch point.
3. Create Scripts with Attribute launch point.
4. Create Scripts with Action launch point.
5. Create scripts with custom condition launch point.

These wizards will drive you through the process of creating a script and associating launch points with it. We will discuss launch points in details in a later section.

Script/Launch point maintenance

After the script and launch point have been created – users can come to the autoscript application for maintenance. This can be done using the main/variables/launch point tabs in that application. One can modify the script, add or update the variables and their bindings as well as the bindings in the launch point level [if they are overridable]. Deletion of variables are not allowed unless none of the launch points refer to that variable.

Variables and Bindings

Next in line are the **variables** and their **bindings**. Variables are what the scripts use to interact with Maximo. Variables can be IN, INOUT or OUT. These follow the classical defintion of IN- pass by value, INOUT/OUT – passed by reference. The script can modify only the INOUT and OUT type of variables. Modification of IN variables in the script has no impact outside the script. Variables can be bound to a Maximo artifact like a mbo attribute, a maxvar, a maximo system property or can be bound to a literal value which does not tie back to any Maximo artifact. Note variables bound to a maxvar or a system property are of type IN only as they cannot be modified by the script. Variables can be scalar or array type. Array type variables are only supported for mbo attribute bindings and are always of IN type. More on array type variables later as this would need a dedicated section to discuss. Variable data type for mbo attributes is driven by the mbo attribute datatype. So for example a variable bound to Purchase Order mbos totalcost attribute will inherit its type ie a double. Variables bound to maxvar and system properties are always of type String. Variables bound to literal values can define their datatype explicitly by setting “literaldatatype” attribute in the autoscriptvars table. The supported literal data types are – ALN, INTEGER, SMALLINT, DECIMAL, YORN, DATETIME and FLOAT. Variable bindings can be defined both at the script level and the **launchpoint** level - provided the definition at the script level allows override of that value. More about this on the **Launchpoints** section.

One other important aspect of the OUT and INOUT variables is how their values can be set back to the Mbos. Mbo attributes can be set with the NOACTION flag which determines if modifying the value of the mbo attribute will cause the field validations action routine to be called or not. Mbo attributes can be set with the NOVALIDATION flag which determines if modifying the value of the mbo attribute will cause the field validations validate routine to be called or not. Mbo atributes can be set using NOACCESSCHECK flags which determine if the Action/Validation routine of the mbo attributes will get called or not. A mbo attribute can be set with any combination of these flags. All these can be done from the script/launch point creation wizards and maintenance application by selecting those [checkbox] options.

Implicit and Explicit Variables

Another important feature is the concept of **Implicit** and **Explicit** variables.

Explicit variables are what we just read about – the ones that you define and bind in the variables page explicitly. Implicit variables are the ones that you do not define in that page and are provided to you behind the scenes by the framework. Implicits follow the convention over configuration pattern where the framework will intelligently inject variables at run-time which might otherwise would have needed JAVA coding to fetch/set. Some of the implicits are injected based on the explicit variables defined and some are injected irrespective of them. Lets cover the ones that are injected irrespective of the explicit variables first. The list of them is as below with a little blurb describing what they are for.

Name	***Type***	***Description***	***Applicability***
app	String	Name of the Maximo application which initiated the script execution.	All launch points
user	String	Name of the user whose action initiated the script execution.	All launch points
mbo	psdi.mbo.Mbo	The current mbo in the context of the script execution. For example in case of the Object Launch Point this will be the Mbo which is generating the events on which the script framework is listening on. For attribute launch point this is the attributes owner mbo. For Action launch point this is say the Escalation or workflows mbo.	All launch points
mboname	String	The name of the current mbo in the context of the script exection.	All launch points
errorkey	String	This one is for throwing MXExceptions from the script without having to explicitly import or refer to that API. This refers to the error key in the MXException. This works together with the errorgroup and the params implicit variables. For those not familiar with the	All launch points

Name	***Type***	***Description***	***Applicability***
		MXException api – its the standard way to throw Exceptions from Maximo based components. The exception message is translated which is the main advantage of using this api as opposed to just raising a standard Java Exception which is not going to be translated.	
errorgroup	String	Its usage is the same as the previous one ie the errorkey. This one points to the error group of the MXException. Together with the errorkey it helps uniquely point to a error message in Maximo message repository.	All launch points
params	String[]	This is the params for the MXException error message. So if the MXException being thrown using this mechanism is parameterized then this params implicit variable should be used to set the parameters.	All launch points
interactive	boolean	Its a boolean variable indicating whether the script is executed in an interactive/UI session [value true] or a background session [like say Integration].	All launch points
evalresult	boolean	This is a boolean variable of type OUT to indicate the result of the condition evaluation.	Only Condition Launch point
onadd	boolean	This boolean variable indicates where the Mbo in the script is being added [ie new Mbo – value true] or not. The script developer can use this to do	All Launch points. Ideally would be valuable for Object Launch Points where the

Name	***Type***	***Description***	***Applicability***
		conditional actions or validations based on the state of the Mbo.	script is applicable for multiple event types [Add, Update, Delete etc]
onupdate	boolean	This boolean variable indicates where the Mbo in the script is being updated [ie exiting Mbo – value true] or not. The script developer can use this to do conditional actions or validations based on the state of the Mbo.	Same as onadd – All Launch points.
ondelete	boolean	Boolean variable indicating whether the mbo in the script context is getting deleted [value true] or not.	Same as onadd – All Launch points.
action	String	The name of the Action that was generated from the Action Launch Point wizard.	Action Launch Point
scriptName	String	The name of the Script thats getting executed.	All Launch points
launchPoint	String	The name of the launch point for which the script is getting executed.	All Launch points
scriptHome	psdi.mbo.Mbo	This is the same as the implict variable “mbo” described earlier. This duplication is for backward compatibility.	Action Launch Point
wfinstance	psdi.workflow. WFInstance	The workflow instance mbo.	Action Launch Point – only when the Action is launched from a workflow.

Now lets talk about those implicit variables that are injected into the script based on the explicitly defined variables.

One thing to keep in mind – all these implicit variables that we describe below are based on the explicitly defined variables whose binding type is a Mbo attribute. There are no implicit variables for explicitly defined variables of other binding types like – Literals, Maxvars and System properties. Below is the list of the implicit variables. Assume “var” is the explicitly defined variable that binds to a mbo attribute.

Name	***Type***	***Description***	***Applicability***
var_required	Boolean	Required flag of the mbo attribute that var binds to.	All launch points. The script can modify it provided that var is of type OUT or INOUT.
var_readonly	Boolean	Readonly flag of the mbo attribute that var binds to.	All launch points. The script can modify it provided that var is of type OUT or INOUT.
var_hidden	Boolean	Hidden flag of the mbo attribute that var binds to.	All launch points. The script can modify it provided that var is of type OUT or INOUT.
var_internal	Same type as the mbo attribute to which var binds to.	For synonym domain kind of mbo attributes [like status] this represents the internal [maxvalue] value for the mbo attribute.	All launch points. Applicable only if var is bound to a mbo attribute that binds to a synonym domain. The script cannot modify it.
var_previous	Same type as the mbo attribute to which var binds to.	The previous value of the mbo attribute ie the value just before the attribute value got changed.	Attribute launch points – applicable only for the attribute that generated the event. The script cannot modify it.
var_initial	Same type as the mbo attribute to which var binds to.	The initial value of the mbo attribute ie the value assigned to that attribute when the mbo was initialized.	All launch points. The script cannot modify it.

Name	***Type***	***Description***	***Applicability***
var_modified	Boolean	Indicates whether the mbo attribute to which the variable var binds to has been modified or not.	All launch points. The script cannot modify it.

We will cover more about them using examples as part of launch points section.

Array Variables

Lets talk a little bit about the array variables as that would introduce you to the concept of Mbo Relationship Path [MRP].

The MRP format is an extension of the current attribute path notation. The current mbo attribute path notation supports the dot "." to allow traversing related Mbos and fetching attribute value from them. An example of the current format is shown below [the examples are based of the PO mbo]:

poline.pocost.costlinenum – which gives us the first POLINE's [the relation name is POLINE] 1st POCOST's [the relation name is POCOST] colstlinenum attribute.

OR

poline[i].pocost[j].costlinenum - which gives us the i th POLINE's [the relation name is POLINE] j th POCOST's [the relation name is POCOST] colstlinenum attribute.

The MRP notation builds up on this attribute path notation. The one fundamental difference being that MRP leads to a List/Array of Mbo attributes. For example if we take the examples above an MRP may look like:

poline.pocost.costlinenum* – which give us a combined list or array of POCOST mbos for all POLINEs.

poline[linecost>100].pocost[percentage<100].costlinenum* – which gives us all costlinenum's where the POCOST mbos has percentage < 100 for all POLINEs with linecost>100. This is effectively filtering the relation clause [poline and pocost] to sift only those mbos that satisfy the specific condition. This can generically be represented by :

poline[condition1].pocost[condition2].costlinenum* where condition1 and condirion2 can be a Maximo condition or a raw SQL where. As you must have already noticed – the * at the end of this notation indicates arrays as opposed to just a scalar value.

Consider another variation of this as shown below:

poline[condition1].pocost[i].costlinenum

which implies costlinenum from all i th POCOST of every POLINE that satisfies condtion1.

We see that the content with the brace '[' and ']' can be overloaded in 3 different ways.

1. It can be a number signifying an array index.
2. It can be a Maximo condition which can be prefixed with the **cond:**
3. Can be a raw SQL where

Now let's take into consideration the different scenarios/context of this MRP evaluation. Note these cases described below can be mixed in a MRP as part of the different relation tokens.

MRP with no filter
This is a MRP where none of the relation tokens has any associated filter. This is the simplest case where the MRP looks like R1.R2… where R1 and R2 are relation names and there is no associated filter clause with these relations. In this case the evaluation would involve just iterating over the content of R1 and R2 MboSets and preparing a list of R2 mbos.

MRP with index filter:
This is a MRP where one or more of the relation tokens has an associated index filter. This is the case where the MRP looks like R1[index1].R[…]…. where a relation token [one or more] has index filter => for that Mboset identified by the relation R1 use the Mbo at index "index1" for the MRP evaluation. This is an in-memory filtering and does not impact the state of the MboSet on which the index filter is getting applied.

MRP with condition filter:
This is a MRP where one or more of the relation tokens has an associated filter with a Maximo condition. This is another example of in-memory filtering where the MRP looks like R1[cond:c1].R2[…] where c1 is a name of a Maximo condition and **cond** is the prefix indicating it's a Maximo condition. The evaluation is based on in memory filtering of the Mbos in MboSet R1 based on the condition c1 and does not impact the state of the MboSet on which the filter is getting applied.

MRP with a where filter:
This is a MRP where one or more of the relation tokens has an associated filter

with a where clause. This amounts to appending an additional where clause to the existing relation clause. This definitely would cause the related MboSet reset to be called and while it might work for certain cases it definitely would result is unpredictable output for interactive user sessions [user browsing an app]. The implementation should use temporary relations created dynamically to replace each of the relations associated with such a filter to evaluate this MRP. So for example if the MRP was R1[where].R2.R3[where] the implementation should replace [transparently] R1 and R3 with temporary relation names with the same relation clause as their original counterparts. This will work only if all the Mbos referred by the MRP relation tokens were not toBeSaved() => their in memory state represents their state at the persistent store.

One thing to note here is – its always recommended to just use the predefined relationships as opposed to dynamically adding filters to the relationship just keeping performance in mind. The reason is – in case we need to access this variable more than one times – the evaluator will always create a temporary relationship between the src and the target objects. This however will prevent the usage of a cached relationship set from the origin mbos.

The where clause approach will append the where clause condition. The the condition approach will filter the MboSet in memory without firing a sql where clause for the added filter. This however will take more time to sift than using the SQL clause. The conditions application can be used to define the filter confitions.

Launchpoint

Launchpoints are what we call the application customization points. Launchpoints define which application artifact the user wants to customize by attaching the script to that point. In effect the script is executed [**launched**] in the context of a launchpoint. For example if the user wants to customize the Asset mbos initialization routine the key words **Asset mbo** & **initialization routine** defines that launchpoint. Another example might be that the user wants to customize the Asset mbo's purchaseprice attribute's field validation routine. Here the key words that define the launchpoint are - **Asset mbo**, **purchaseprice attribute** and **field validation routine.** So in effect a launchpoint is a configuration to identify what application point we are trying to customize. A script can be associated with 1 or more launchpoint at the same time. For example a generic site level validation script can be associated with all site level objects in Maximo – where each association is defined as an individual launchpoint. A script can be associated with launchpoints of the one and only one type. For example a script associated with the Object launch point cannot be associated again with another attribute launch point. Once the first association is done between a script and a launch point – all subsequent launch points have to be of the same type as the first one.

Now lets take a look at the launchpoints [aka customization points] touched up by the scripting framework. Below is the list of the points supported currently.

1. Mbo initialization and save point logic [aka Object launchpoint].
2. Mbo attribute value modification – validation and action logic [aka Object attribute launchpoint].
3. Actions – which are used by a multitude of other components like Workflow, Escalation, UI Menu, UI Buttons [aka Action launchpoint].
4. Custom conditions – used by Workflow conditions, Conditional UI etc [aka custom condition launchpoint].

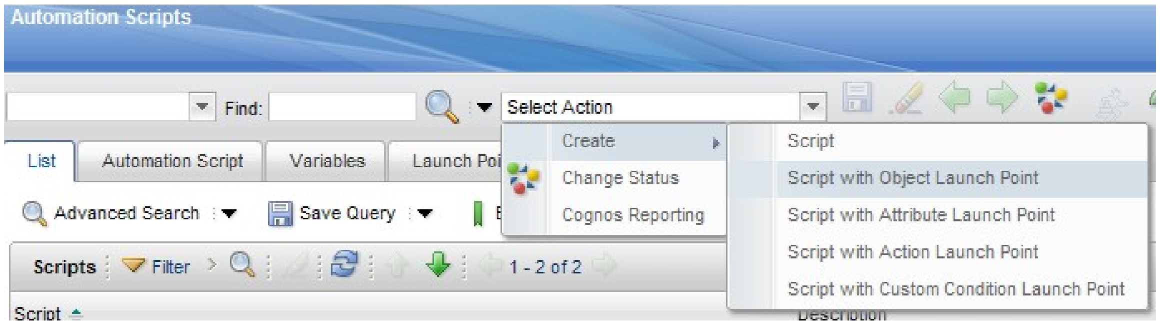
Launchpoints are what you think of first when you want to customize an application. No wonder all the wizards in the Script app starts with defining the launchpoint. Next lets explore the individual point types to understand what they bring into the table for customizers and deployers.

Object Launch Point

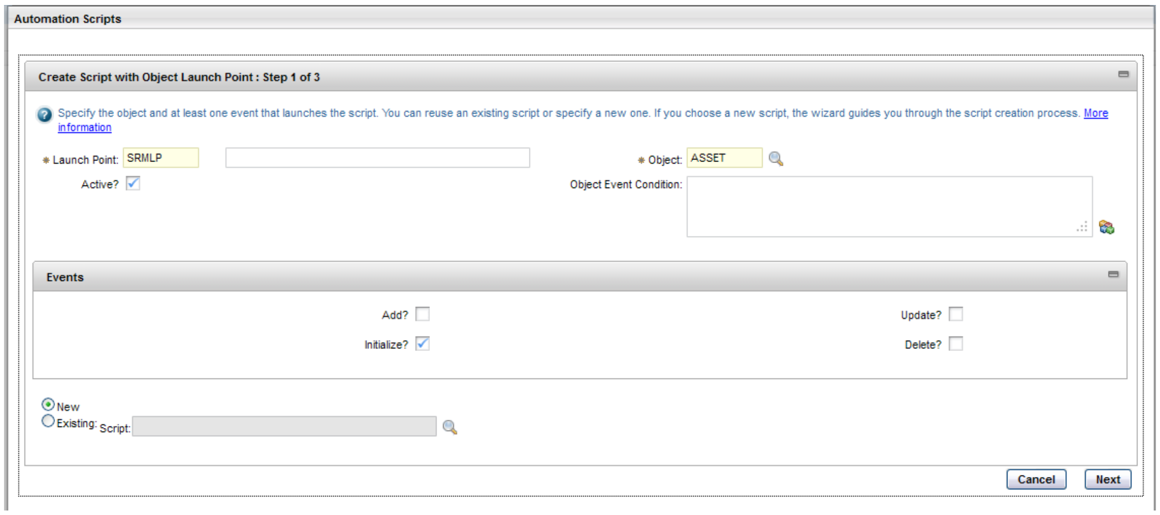
First in our list is the **object launch point**. This launchpoint lets you invoke scripts for the Mbo events – init and save point ones [add, update and delete]. A launch point can be configured to listen to one or more of these events at the same time. The script will have access to the event Mbo [via implicit variable "mbo"]as well as all the related Mbos. The initialization event based scripts can be used to set calculated fields, set fields as readonly/required/hidden or set conditional defaults to mbo attributes. The save point event based scripts can be used to implement save point object validations as well as save point actions.

Below is an example that will demonstrate a initialization point script and the next one would demonstrate a save point script.

So lets take the use case in mind before we jump into the code and configuration. Suppose we want to customize the Asset application to display the total spare part quantity in a new non-persistent Asset object attribute called sparepartqty. This boils down to the requirement - whenever an Asset mbo gets initialized the sparepartqty will display the sum of all the spare part quantities associated with that asset. So based on our knowledge of launch points we get that it will be a **Object Launch Point** for the **object Asset** and we need to attach the script to the **initialization event** of the Asset object. To do this we need to launch the "Create Scripts with Object Launch Point" wizard as shown below.



Once the wizard is launched the first thing we do is to create a launch point as shown below. Note that the “initialize” event is what we want to use for launching this script.


Automation Scripts

Create Script with Object Launch Point : Step 1 of 3

Specify the object and at least one event that launches the script. You can reuse an existing script or specify a new one. If you choose a new script, the wizard guides you through the script creation process. More information

Launch Point: SRMLP

Object: ASSET

Active? ✓

Object Event Condition:

Events

Add? ☐

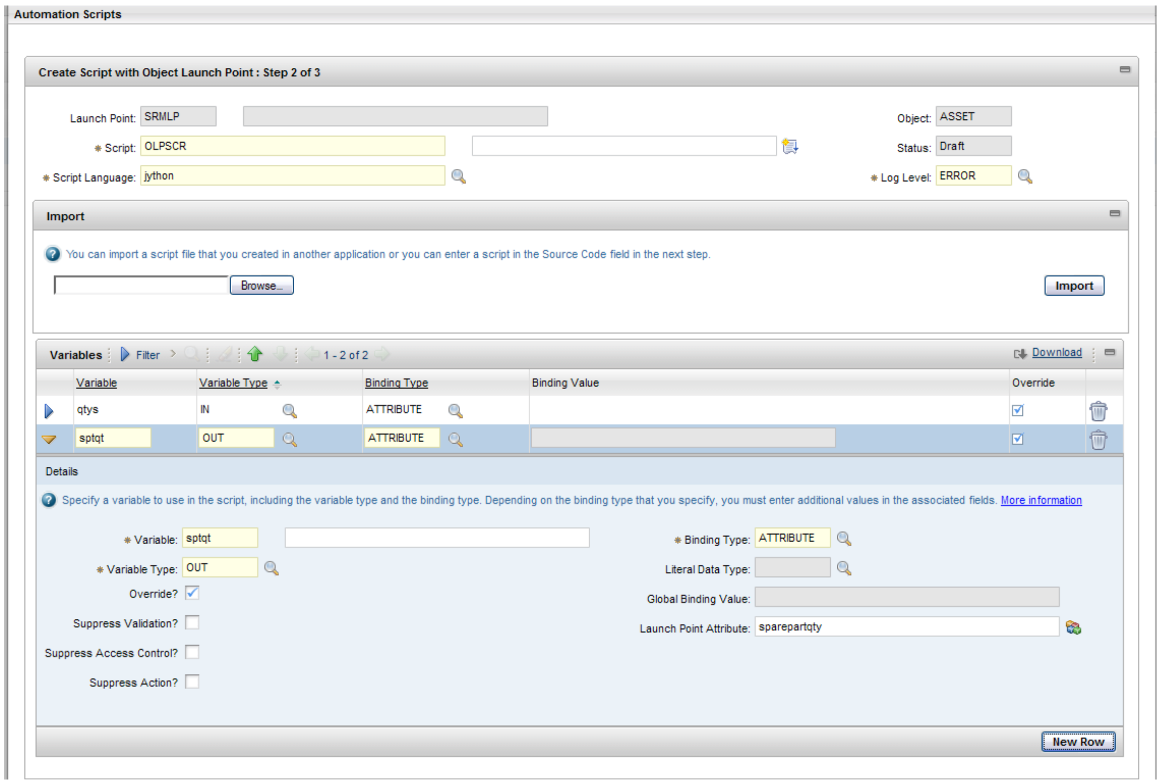
Update? ☐

Initialize? ✓

Delete? ☐

[● New] ○ Existing: Script:

So now lets look at the variables we might need to do this customization. First of course is a variable called sptqt which binds to the new Asset mbo attribute sparepartqty. Now we only intend to set the value of this attribute and hence this variable would be of type OUT. Next we need to get all the quantities from the related Sparepart Mbos of the Asset. To do that we use the array variable notation * to get an array of quantity values from the related sparepart MboSet. Lets say the array variable is qtys and its bind value would be <asset to sparepart relation name>.<attribute name>* which is sparepart.quantity*. The * at the end of-course indicates the array nature of this variable and also instructs the framework to form the array using the specified relationship.



Automation Scripts

Create Script with Object Launch Point : Step 2 of 3

Launch Point:	SRMLP	Object:	ASSET
* Script:	OLPSCR	Status:	Draft
* Script Language:	jython	* Log Level:	ERROR

Import

You can import a script file that you created in another application or you can enter a script in the Source Code field in the next step.

Variables | Filter | 1 - 2 of 2 | Download

Variable	Variable Type	Binding Type	Binding Value	Override
qtys	IN	ATTRIBUTE		✓
sptqt	OUT	ATTRIBUTE		✓

Details

Specify a variable to use in the script, including the variable type and the binding type. Depending on the binding type that you specify, you must enter additional values in the associated fields. More information

* Variable:	sptqt	* Binding Type:	ATTRIBUTE
* Variable Type:	OUT	Literal Data Type:	
Override?	✓	Global Binding Value:	
Suppress Validation?		Launch Point Attribute:	sparepartqty
Suppress Access Control?			
Suppress Action?			

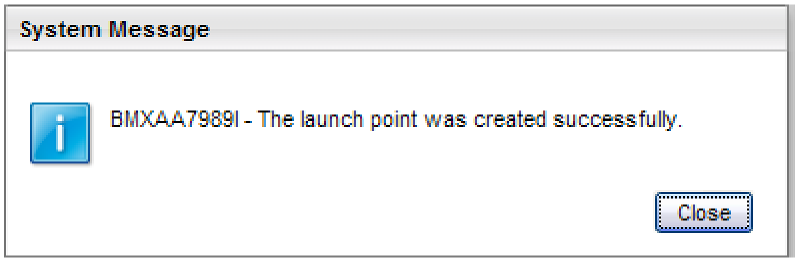
And as mentioned earlier array variables are always of type IN and that is perfect for this as we are not modifying the quantities – we are merely summing that up. So with these basic variables defined we next would attempt to write the script as below

```
if qtys is not None:
      sptqt = sum(qtys)
```

Basically a 2 liner which validates if there are infact sparepart Mbo's and if there is then sum them up and set it to the sptqt variable. The scripting framework picks that up and sets the value back to the binding of the sptqt ie sparepartqty. So the amount of Java coding done here is a BIG ZERO – its a pure jyhton script. Now going by the nature of the calculated fields – the sparepartqty should be always read only. And what best place to set that to read only other than this script – which embodies the Asset initialization event. Below is the final script which adds the code to set the sparepartqty attribute to read only.

```
sptqt_readonly=True
if qtys is not None:
      sptqt = sum(qtys)
```

Once you press the create button in the last wizard step to create the script – a successful creation of the script and the launch point will generate this response as shown below.



BMXAA7989I - The launch point was created successfully.

In case of a compilation error you would be forced to stay back on the last page till you cancell or fix this script.

Here you see the magic of this implicit variable concept. When you bound sptqt to sparepartqty attribute the scripting framework injected at runtime not only the variable sptqt but also some implicit variables like sptqt_readonly, sptqt_required and sptqt_hidden each of which are of type boolean and caters to the read only, required and hidden flags of the Mbo attribute. Setting a mbo field to readonly otherwise would have required java coding. A java code to do this same functionality would look like below

```
mbo.setFieldFlag(“sparepartqty”, MboConstants.READONLY, true);
MboSetRemote sparepartSet = mbo.getMboSet(“sparepart”);
int i = 0;
MboRemote sparepartMbo =  sparepartSet.getMbo(i);
double totalQty = 0;
while( sparepartMbo != null)
{
        totalQty += sparepartMbo.getDouble(“quantity”);
        sparepartMbo =  sparepartSet.getMbo(++i);
}
mbo.setValue(“sparepartqty”, totalQty,  MboConstants.NOACCESSCHECK);
```

So by this time you must have figured out what amount of pain the scripting framework has saved you!. Dont just think in terms of the lines of code [which is almost a 1:2] think also about the api knowledge that you would need for this simple task. And this code does not even address the pain that you will go through to attach this code to the Asset Mbo's initialization routine. There your choices are even more hairy – either you would have to extend the Asset mbo and override the init() method of that mbo to put your code [in which case in the the above code just replace the mbo variable with “this” pointer] or you would end up attaching your code as a listener to the Mbo's event – which is a separate api stack on its own!. In the scripting framwork all these are taken care for you the moment you have pressed the “Create” button to end your wizard [thus submitting your script]. As a script developer you just code the logic – the framework takes care of the behind the scenes plumbing work to manage and execute your script. So yes you will save time and money with this for sure.

By the way the above java code would work [provided you do the jython syntax styling on it – like removing the ; and the curly braces and ..] even inside the Jython script – that is just in case you are an avid java programmer!. Just don't forget to import the following before you submit the script -

```
from psdi.mbo import MboRemote
from psdi.mbo import MboConstants
from psdi.mbo import MboSetRemote
```

which is jython way of importing external java libraries. This shows that while scripting framework gives tremendous power to script developers to get their Maximo customization done without knowing Maximo apis – it does not take away the Java coding power from the developers who are used to that. So effectively you can invoke all the Maximo apis [for example make a Web service call to fetch some external data using the Integration framework] from inside a script as long as you have imported them properly.

Next lets move onto some save point validations which hopefully will help demonstrate more features of this framework. As before lets deal with the use case first. The use case here is a need to customize the Asset mbo to enforce a naming convention for assets [assetnum] based on their types [assettype]. This effectively boils down to the requirement that **whenever we are creating Assets** we have to follow a naming convention for the assetnum. The key words here are in blocks which help us identify the launchpoint type and the event point in that type. Its an object launchpoint for the Asset mbos add event. So we use the Object Launch point wizard to create and deploy this custom logic. To start with we need to figure out the variables and their bindings. From the requirement its clear we need the 2 input values from the assetnum and assettype. So there are 2 IN variables called anum and atype which are bound to those attributes respectively. Those are the only 2 variables that we need to do this task. Below is the script code [in Jython]

```
def setError(prefix):
        global errorkey,errorgroup,params
        errorkey='invalidassetprefix'
        errorgroup='asset'
        params=[prefix,assettype]

if atype_internal=='FACILITIES' and not anum.startswith('FT'):
        setError('FT')
elif atype_internal=='FLEET' and not anum.startswith('FL'):
        setError('FL')
elif atype_internal=='IT' and not anum.startswith('IT'):
        setError('IT')
elif atype_internal=='PRODUCTION' and not anum.startswith('PR'):
    setError('PR')
```

Here we define a jython function called setError which is responsible for setting the error flags. We see the use of the <variable name>_internal implicit variable which is applicable only for attributes which uses a synonymdomain. The <variable name>_internal provides the internal value for that attribute based on its current external value. So this script uses the internal value of the assettype attribute to establish the naming convention. For example Assets with assettype [internal] value FACILITIES should have assetnum starting with “FT” etc.

This same code in java would have required to know the Maximo api to find the internal value of the assettype using the Translator api as below.

```
String domainId =
MXServer.getMXServer().getMaximoDD().getMboSetInfo("asset").getMboValue
Info(“assettype”).getDomainId();
String atypeInternal=
MXServer.getMXServer().getMaximoDD().getTranslator().
toInternalString(domainId,mbo.getString(“assettype”), mbo);
```

This is something even an avid Maximo developer would find difficult to remember and use and we dont even want to go into the plumbing work thats needed to execute this code.

We also see how errors can be thrown in this framework without using Maximo apis. Just set the errorgroup, errorkey and params [optional] to the configured error message group, key and the params array can be derived from the script variables. In this case we have predefined the error group asset and the error key “invalidassetprefix” using the Maximo database configuration application. The params as you can make out are derived at run-time from the script.

One thing to note here is that this way of setting error flag to throw error is not real-time ie when the script code is executing the exception would be thrown only after the script code has completed execution. At the end of the script execution the framework would detect that an error flag is set and it will throw the corresponding Maximo excpetion for that error group/key combination. So you should consider the fact that even after setting the error flags in the script – the script execution will continue and you should have adequate checks in your script code to bypass that code if the error flag is set. Hopefully this does not give you the wrong impression that you cannot throw the raw MXException from the script code – shown below is how you do it.

```
from psdi.util import MXApplicationException
....
....
if <some condition>:
    params = [prefix,assettype]
    raise MXApplicationException('asset','invalidassetprefix', params)
```

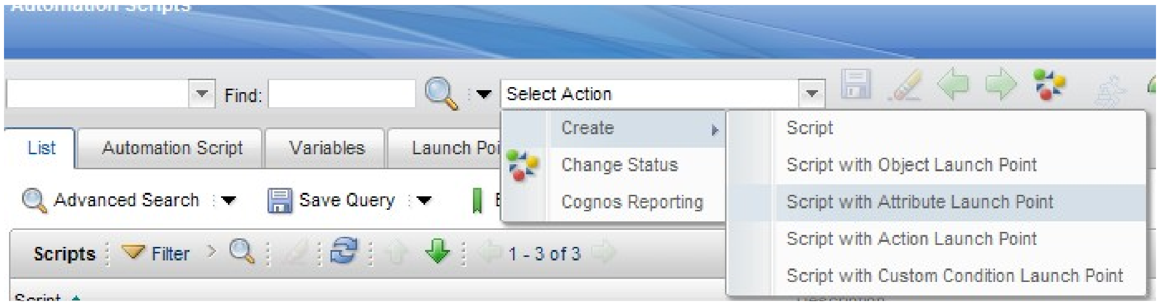
As before the scripting framework takes care of all the plumbing behind the scenes. Once you are done submitting your script using the wizard you can come to the Asset app and try to save a test asset. You should see your validation routine gets executed immediately. **No restart, No rebuilding ear and No redeployment**.

Attribute Launch Point

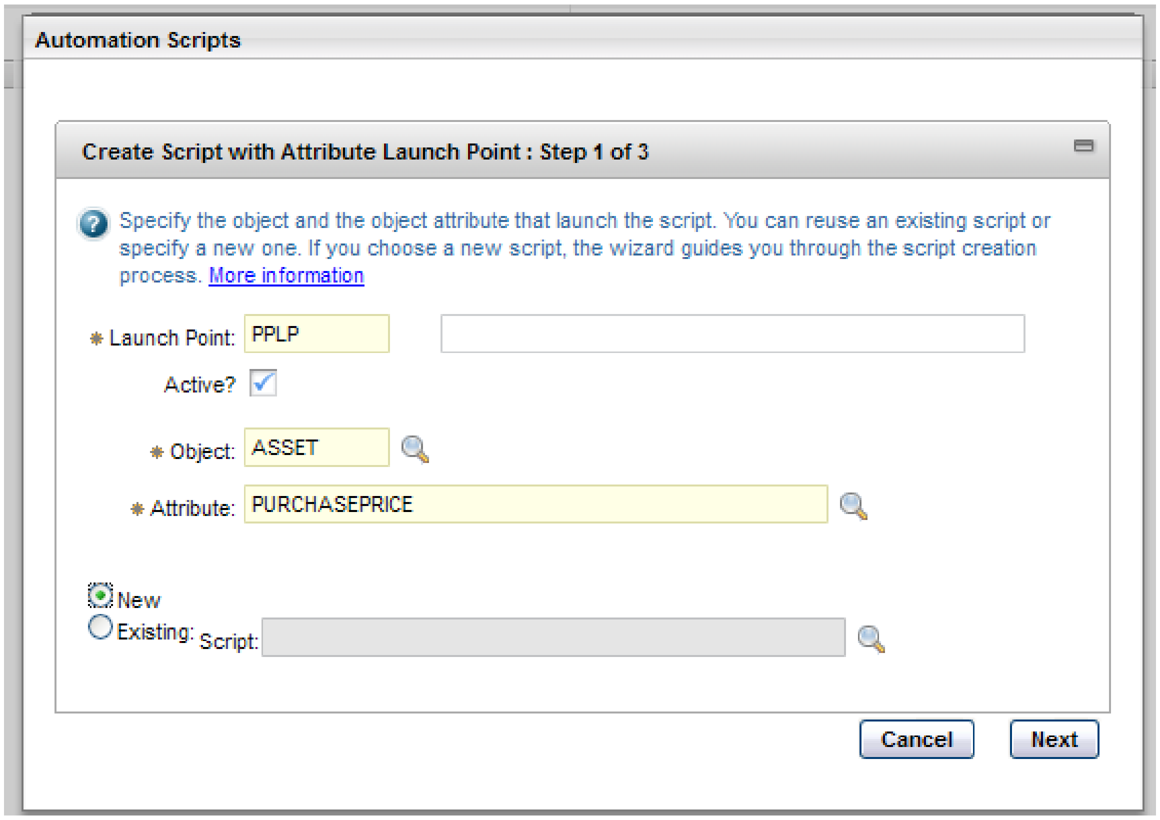
Next we cover the **Attribute Launch Point** where one can customize the field validation and actions using the scripting framework. As before the script will have access to the event Mbo [via implicit variable "mbo"] as well as all the related Mbos. In addition the modified attribute would also be available implicitly as a variable inside the script. The variable name would be the lower case value of the modified attribute name. A couple of examples below would help explain this better.

As before we continue customizing the Asset Mbo. The use case this time is to add custom business logic based on the **Asset purchase price** [attribute purchaseprice in Asset] **value**. For example we want to set the vendor field required or not required based on the purchaseprice attribute value and also to calculate the replacement cost based on the purchase price. We also want to make sure the purchase price does not exceed a maximum allowed value.

Based on the use case we know that the logic has to injected at the modification of the purchasprice attribute value – which implies it should be modeled as an Attribute Launch Point. Below is a way to get to the wizard.



Once we launch the wizard our step 1 is to create the launch point as shown be


Automation Scripts

Create Script with Attribute Launch Point : Step 1 of 3

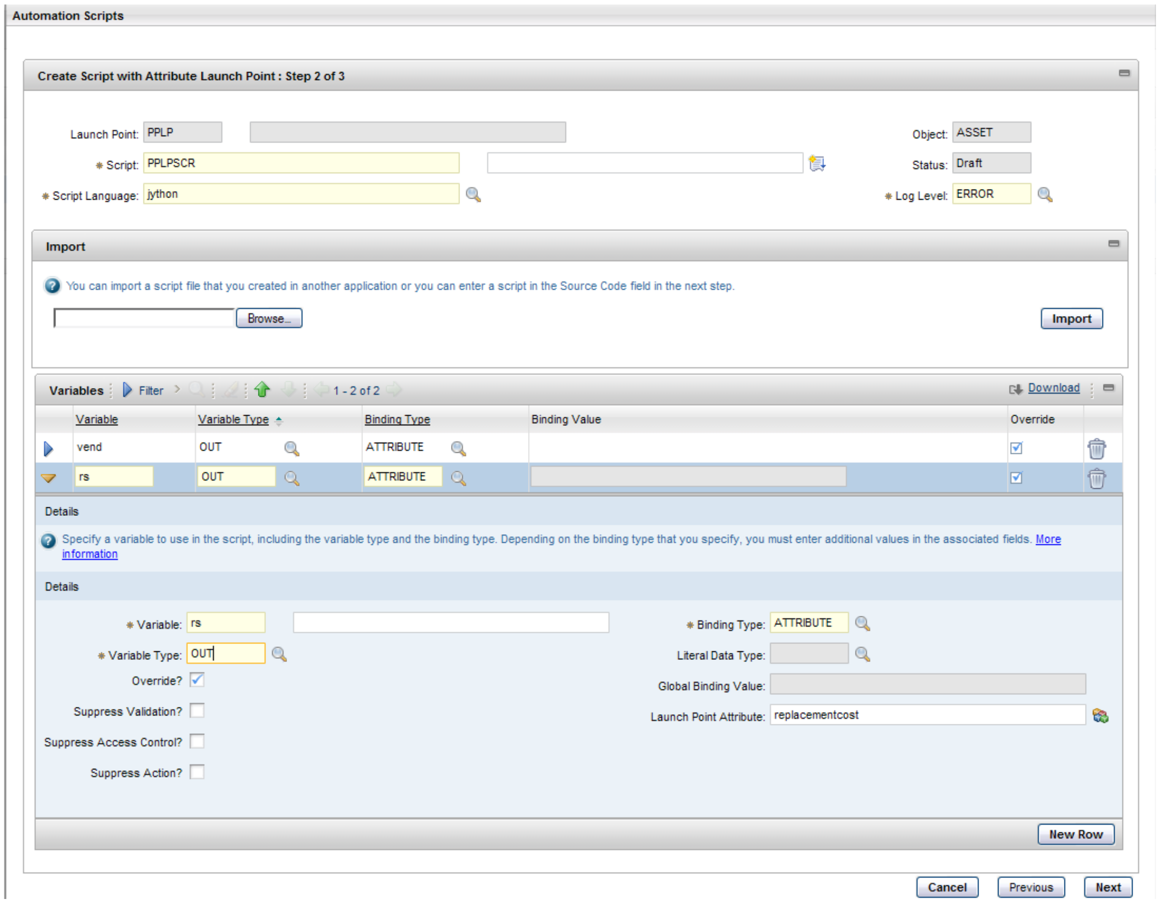
Specify the object and the object attribute that launch the script. You can reuse an existing script or specify a new one. If you choose a new script, the wizard guides you through the script creation process. More information

* Launch Point:	PPLP
Active?	✓
* Object:	ASSET
* Attribute:	PURCHASEPRICE

[✓ New] Existing: Script:

Now lets look at the variables that we would need for this task. We definitely need the purchaseprice variable. But we don't need to define that explicitly as that's already available implicitly inside the script as its the attribute on which the script is listening on. Next we need the vendor attribute to set it required or not based on the purchaseprice value. Say the variable for that is **vend** and its type is OUT. The only remaining one is the replacementcost and say we bind that to the variable named **rc** with type as OUT**.**

Variable name	Variable type	Binding
vend	OUT	vendor
rs	OUT	replacementcost


Automation Scripts

Create Script with Attribute Launch Point : Step 2 of 3

Launch Point:	PPLP	Object:	ASSET
* Script:	PPLPSCR	Status:	Draft
* Script Language:	jython	* Log Level:	ERROR

Import

You can import a script file that you created in another application or you can enter a script in the Source Code field in the next step.

Variables 1 - 2 of 2

Variable	Variable Type	Binding Type	Binding Value	Override
vend	OUT	ATTRIBUTE		✓
rs	OUT	ATTRIBUTE		✓

Details

Specify a variable to use in the script, including the variable type and the binding type. Depending on the binding type that you specify, you must enter additional values in the associated fields. More information

Details

* Variable:	rs	* Binding Type:	ATTRIBUTE
* Variable Type:	OUT	Literal Data Type:	
Override?	✓	Global Binding Value:	
Suppress Validation?		Launch Point Attribute:	replacementcost
Suppress Access Control?			
Suppress Action?			

So now lets look at the script below which we enter in the next step of the wizard.

```
if purchaseprice > 200:
        errorgroup = "something"
        errorkey="else"
else:
        if purchaseprice >= 100:
                vend_required=True
        else:
                vend_required=False
        rc = purchaseprice/2
```

As we see the 3 variables purchaseprice, errorgroup and errorkey are implicit variables which you did not have to define. The only ones that you had to define for this script explicitly are vend and rc. And if purchaseprice is > 100 we set the vendor as required by setting implicit boolean variable vend_required to true and false otherwise. As we discussed earlier – when a variable is bound to a Mbo attribute the framework always injects implict variables that represent the meta data state of the attribute – like [vend_]readonly, [vend_]required and [vend_]hidden. Finally we calculate the replace cost by setting the variable rc to purchaseprice/2 [basically rc is a function of purchase price]. Note carefully that the whole logic is done inside the “else:” block. A faulty way to write the script would have been as below.

```
if purchaseprice > 200:
        errorgroup = "something"
        errorkey="else"

if purchaseprice >= 100:
        mbo.setFieldFlag(“vendor”,MboConstants.REQUIRED, true)
else:
        mbo.setFieldFlag(“vendor”,MboConstants.REQUIRED, false)
        mbo.setValue(“replacementcost”,purchaseprice/2)
```

This is using the faulty assumption that setting the errorgroup and errorflag will cause the script to stop execution and throw the error [like a real time throw exception which returns the control back to the caller]. As mentioned earlier this is not how the error flags work. Unlike throwing an exception its not real time. It will throw the exception only after the script execution completes. So in this faulty code case the vendor's required flag will be affected and the replace cost value will be modified and then the script will throw an exception which will not rollback those changes. This is because we used explicit mbo calls to set the value and the flag. Had we used the variables this would have still worked as the framework checks for the error flag right after the script execution end and before setting an OUT and INOUT variable values back to the Mbo's. So the code below would have still worked though it would not be as readable as the original.

```
if purchaseprice > 200:
        errorgroup = "something"
        errorkey="else"

if purchaseprice >= 100:
        vend_required=True
else:
        vend_required=Flase
        rc = purchaseprice/2
```

Now lets look at another example before we move on to the next launch point type. The next use case would be to make the calculated field that we created for the Asset sparepart total quantity calculation more real-time. For example if we change the quantities of the related spareparts we should see the calculated field [sparepartqty] value change real-time. For that we would need to create an script which will be associated with the sparepart quantity attribute. The explicit variable bindings are shown below.

Variable name	Variable type	Binding
sptqt	INOUT	&owner&.sparepartqty

As apparent from the binding the variable sptqt is bound to the owner's [Asset mbo] sparepartqty attribute. We do have to set the sptqt variable with the No Access Check as its marked as readonly by our original [Object launch point] script at the initialization of the Asset mbo. As explained before the variable quantity [the attribute on which the script is listening to] would be implicitly injected in the script as that's the attribute on which the script is listening. The script will look as below.

```
sptqt=sptqt+quantity-quantity_previous
```

Here we see the use of the implicit variable quantity_previous. This represents the value of the quantity attribute prior to the modification ie the value at the initialization of the sparepart Mbo. And the variable quantity of course holds the current modified value of the quantity attribute.

So combination of this script along with the Object Launch point script can give you a complete calculated field logic implemented using the scripting framework with 3 lines of script and a few clicks in the Script Wizard!.

Action Launch point

We all know about the Maximo Actions framework. In case you didn't know - Maximo has a built in library of Actions which can be invoked from Workflows, Escalations and UI Menu's and UI Buttons and a slew of other components. More often than less those built in library of Actions are not enough and implementers go out and develop their own custom Actions and of course the language you are forced to use is JAVA. Scripting addresses this concern where an Action can be scripted with a scripting language of your choice [OOTB – Jython and JavaScript]. Lets see how we can use a scripted Action to do some calculated meters for Assets.

As in before lets first study the requirement which is to be able to calculate an Asset meter value based on some other meters associated with that Asset. For example lets say we want to calculate the value of the PRESSURE meter based on the IN-PRESSUR and O-PRESSUR meter such that the last reading of the PRESSURE meter is the summation of the IN and O-PRESSUR meters readings. And lets say we choose to do it in an offline fashion where as opposed to a real time fashion [for which we would have needed to use Object Launch points to trap meter modification events]. The simplest way to do offline actions in a repeated fashion in Maximo is to use Escalations which are nothing but cron jobs which execute a predefined Action in the context of a Mbo. Now instead of writing the Action java code we will script it up. Shown below are the steps to do this.

First we define a relationship called **assetmeterip** [asset meter input pressure] using the DB-Config application which relates an Asset to the Asset meter named IN-PRESSUR. The where clause is as below.

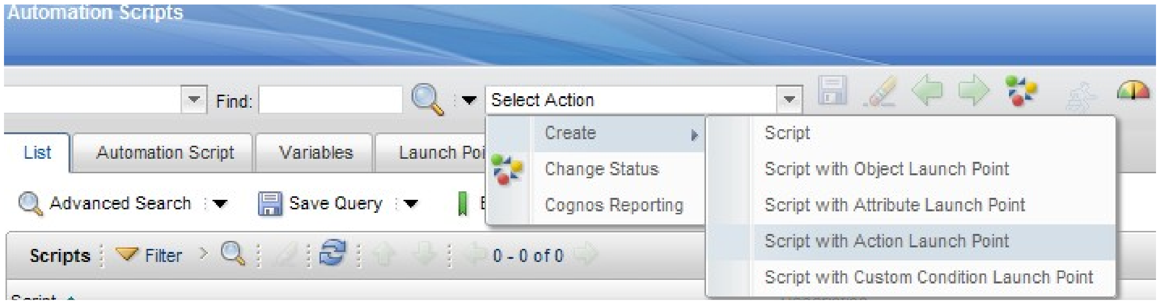
assetnum=:assetnum and siteid=:siteid and metername='IN-PRESSUR'

Similarly we define the other 2 relationships namely **assetmeterop** and **assetmeterp** as below

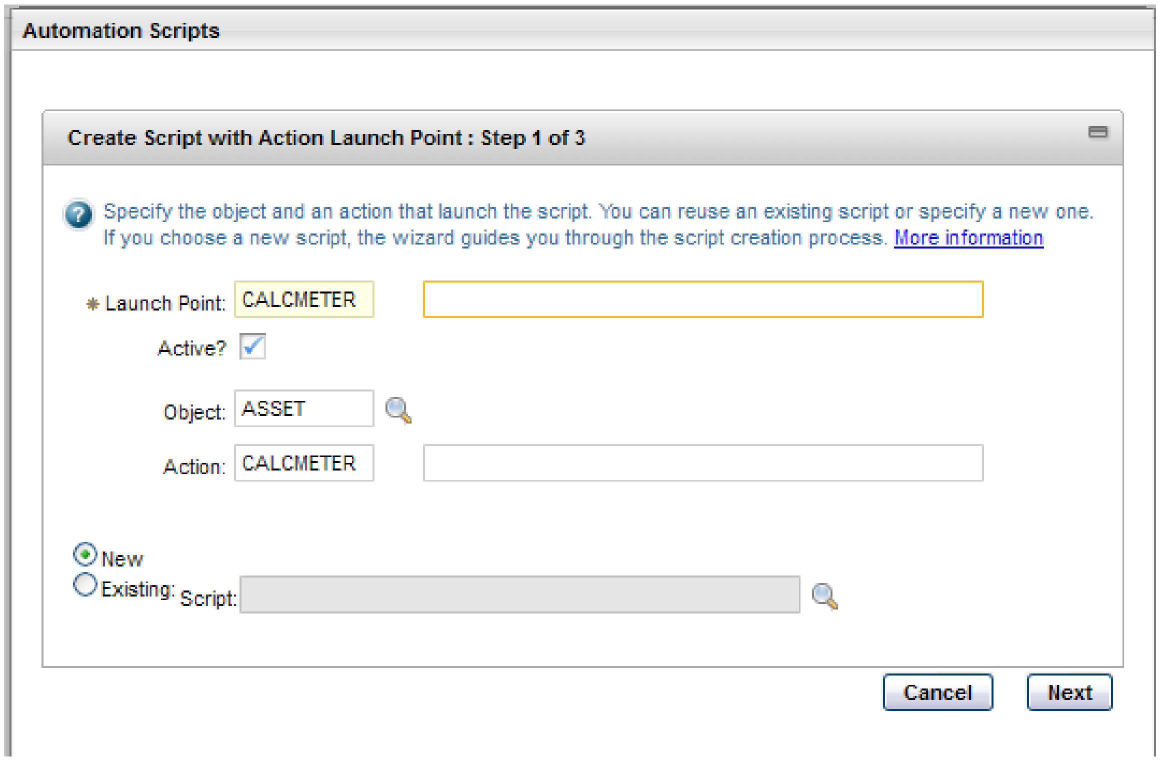
assetnum=:assetnum and siteid=:siteid and metername='O-PRESSUR'

assetnum=:assetnum and siteid=:siteid and metername='PRESSURE'

Next we use the Action Launch Point wizard to define the Action Launch Point.



Though this wizard would create the Action behind the scene – its the responsibility of the implementer to attach that Action to the escalation, workflow or the UI button/menu. By default the launch point name is used as the name of the Action, but you can modify the value to suit your naming convention. The launch point name need not be the same as the Action name. In the first step of the wizard you would see that the object name is optional, which is in-line with the Maximo Action framework where an Action may or maynot be associated with a Maximo object. In this case however we do want to specify the object as Asset as the Action is specific to the Asset Mbo. Since we are defining a new script we will choose the “New Script” option.


Automation Scripts

Create Script with Action Launch Point : Step 1 of 3

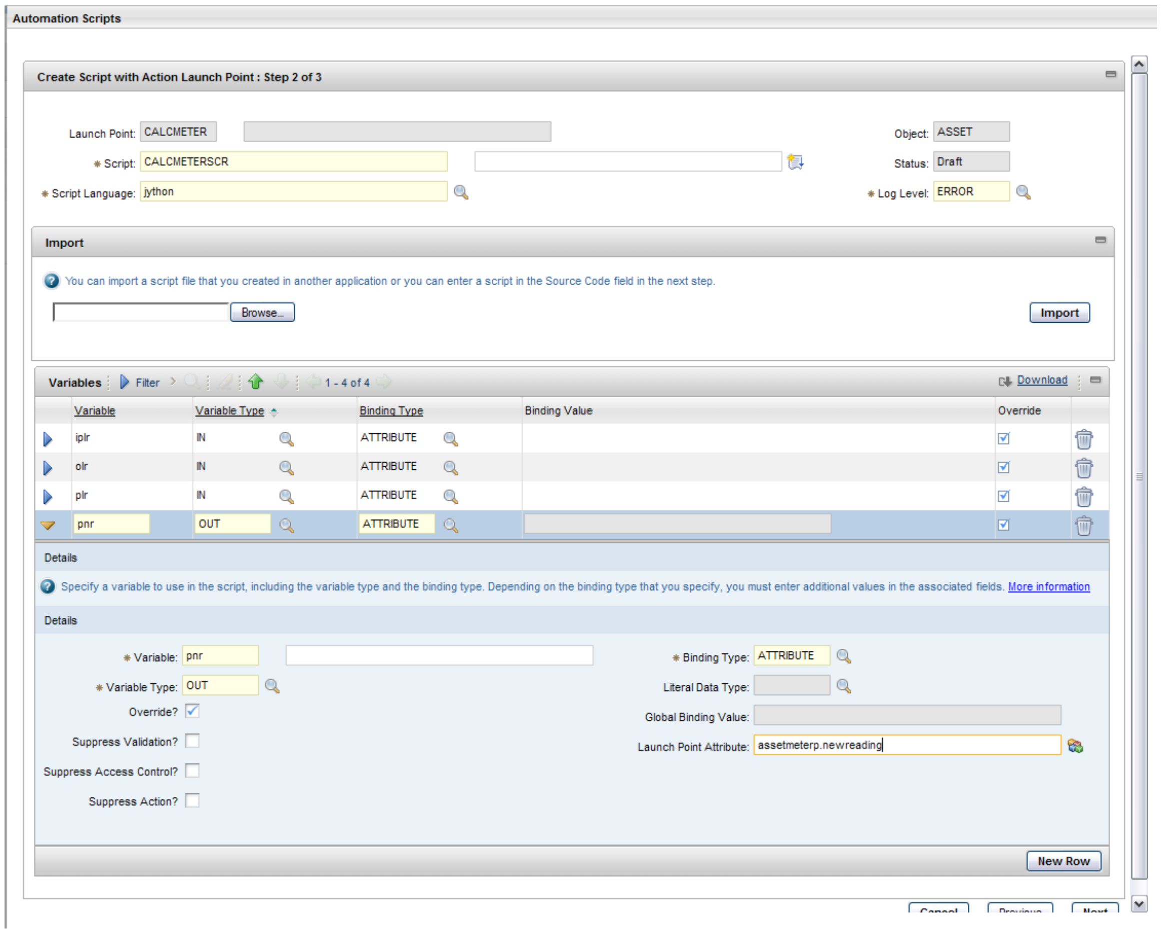
Specify the object and an action that launch the script. You can reuse an existing script or specify a new one. If you choose a new script, the wizard guides you through the script creation process. More information

* Launch Point:	CALCMETER
Active?	✓
Object:	ASSET
Action:	CALCMETER

[✓ New] Existing: Script:

The next page is the bindings page where we are going to define the variables that we intend to use for the script and their bindings. To do this job all we need is the last reading value of the IN-PRESSUR and O-PRESSUR meters and set the calculated value to the new reading attribute of the PRESSURE meter. We however do not want to set the value if the calculated value is the same as the last reading value of the PRESSURE meter as that would generate meter reading history even though the reading never got modified. To check this we would need the last reading value of the PRESSURE meter. So our variable bindings will look like below

Variable name	Variable type	Binding
iplr	IN	assetmeterip.lastreading
olr	IN	assetmeterop.lastreading
plr	IN	assetmeterp.lastreading
pnr	OUT	assetmeterp.newreading



Automation Scripts

Create Script with Action Launch Point : Step 2 of 3

Launch Point:	CALCMETER	Object:	ASSET
* Script:	CALCMETERSCR	Status:	Draft
* Script Language:	jython	* Log Level:	ERROR

Import

You can import a script file that you created in another application or you can enter a script in the Source Code field in the next step.

Variables — 1 - 4 of 4

Variable	Variable Type	Binding Type	Binding Value	Override
iplr	IN	ATTRIBUTE		✓
olr	IN	ATTRIBUTE		✓
plr	IN	ATTRIBUTE		✓
pnr	OUT	ATTRIBUTE		✓

Details

Specify a variable to use in the script, including the variable type and the binding type. Depending on the binding type that you specify, you must enter additional values in the associated fields. More information

Details

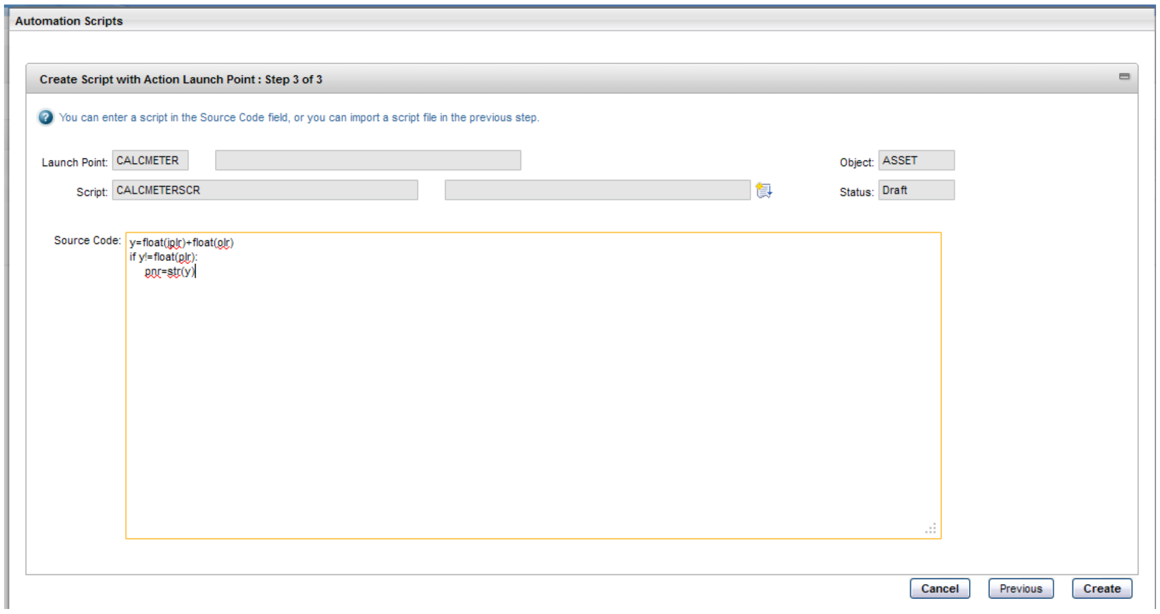
* Variable:	pnr	* Binding Type:	ATTRIBUTE
* Variable Type:	OUT	Literal Data Type:	
Override?	✓	Global Binding Value:	
Suppress Validation?		Launch Point Attribute:	assetmeterp.newreading
Suppress Access Control?			
Suppress Action?			

The iplr [IN-PRESSUR meters last reading], olr [O-PRESSUR meters last reading] and plr [PRESSURE meters last reading] are all of type IN as we just need those values to calculate the pnr [PRESSURE meters last reading]. Note the pnr is of type OUT as we are going to set it back to the PRESSURE meter mbo.

Next page is the script code and it will look as below.

```
y=float(iplr)+float(olr)
if y!=float(plr):
    pnr=str(y)
```

As you notice here y is a local variable to the script.



Automation Scripts

Create Script with Action Launch Point : Step 3 of 3

You can enter a script in the Source Code field, or you can import a script file in the previous step.

Launch Point: CALCMETER

Object: ASSET

Script: CALCMETERSCR

Status: Draft

Source Code:

```
y=float(iplr)+float(olr)
if y!=float(plr):
   pnr=str(y)
```

As you must have figured out – the if check in the 2nd line takes care of not updating the pnr value if the calculated value is the same as the plr [PRESSURE meters last reading]. Note this calculation was implemented as a mere addition just as an example. In real implementations it can be any complicated mathematical calculation as needed for your business case and only limited by the mathematical support provided by the scripting language of your choice.

Now we are not done yet as we need to associate this Action to an Escalation. Our next step is to create an escalation which will only apply to those Assets which have all those 3 meters. We use the escalation condition to implement that sifting functionality. The SQL condition for the above case will look like below.

```
exists (select assetnum from assetmeter where metername='IN-PRESSUR' and
assetnum=asset.assetnum and siteid=asset.siteid) and exists (select assetnum from assetmeter
where metername='O-PRESSUR' and assetnum=asset.assetnum and siteid=asset.siteid) and
exists (select assetnum from assetmeter where metername='PRESSURE' and
assetnum=asset.assetnum and siteid=asset.siteid)
```

Next we select the Action for this escalation – the name of the Action is the same as the launch point name [unless you had modified it in the step 1 of the wizard]. After we activate the escalation our job is done – the escalation executes the scripted Action for all those Assets with the 3 meters and the modified Asset meter readings are saved and committed by the escalation framework.

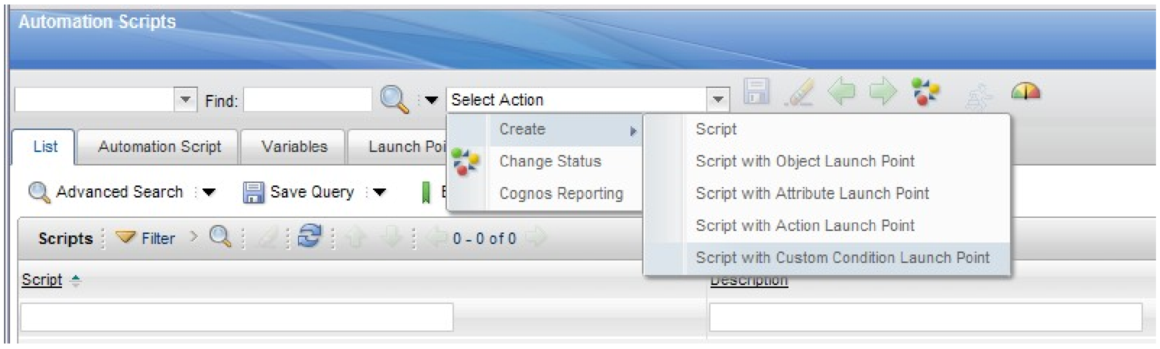
An important thing to note here is if you had chosen to not attach the Action to a Maximo object – the step 2 of the wizard [variables and bindings] will not let you bind a variable to a Mbo attribute. You can however use the literal, system property and maxvar variable binding types. A use case for that might arise when you write a generic Action that say invokes a service [like a Web service] which is not specific to a Mbo or you intend to do the script code based on direct usage of the Mbo apis and hence will not need the mbo attribute bindings.

Condition Launch Points

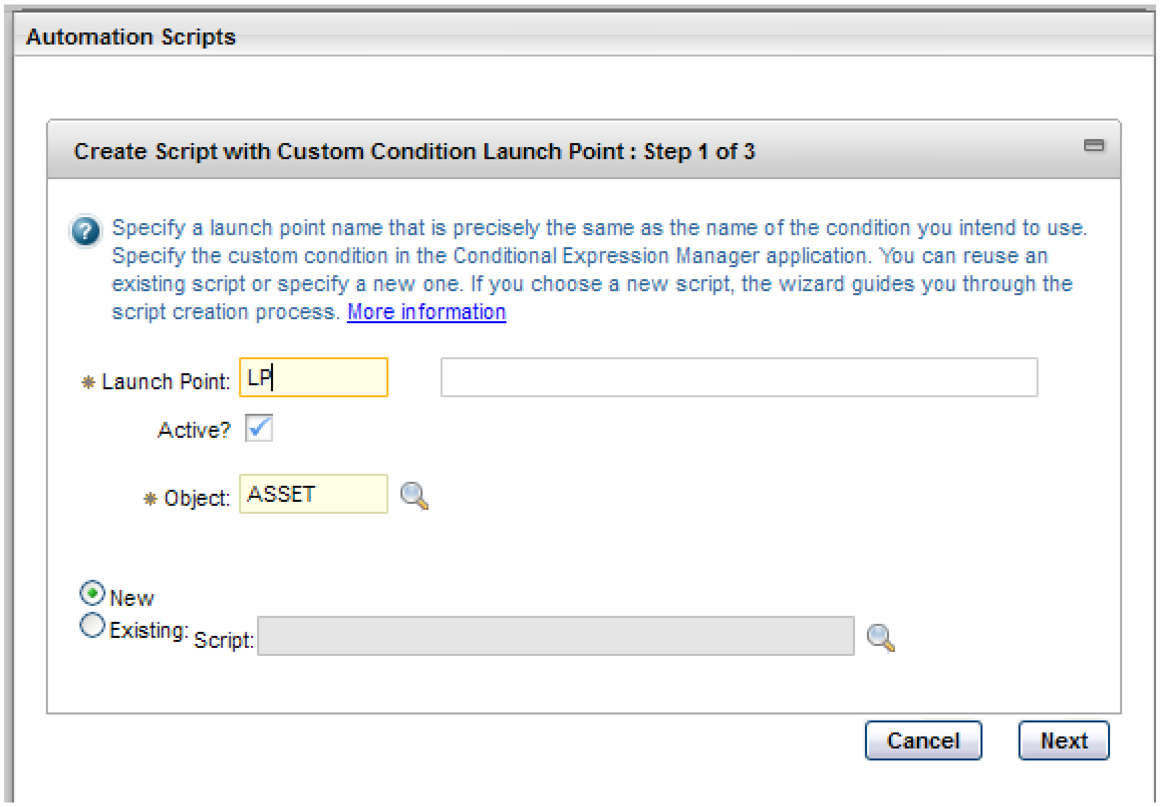
We all know about Maximo conditions which are [based on the javacc parser] used in Workflows, Conditional Uis etc. One aspect of custom conditions lets you write up a condition using Java code in case the condition is complicated enough to be encoded using the javacc based condition grammar. This launch point lets you avoid that Java coding and enables you to attach a scripted condition to these Maximo components [Workflows/Conditional UIs].

Lets take an example to understand it better. The use case is to add a condition to the workflow that would change the status of Asset from not ready to operating if the Asset has spareparts quantity total as greater than 10 and the asset vendor is not null.

As before we would launch the wizard for the condition launch point from the Autoscript application [List Tab → drop down actions → Create → Create Script with Custom Condition Launch Point] and like other's this is also a 3 step process.



Step 1 is ofcourse defining the launch point. As we can see this is an entirely new script.


Automation Scripts

Create Script with Custom Condition Launch Point : Step 1 of 3

Specify a launch point name that is precisely the same as the name of the condition you intend to use. Specify the custom condition in the Conditional Expression Manager application. You can reuse an existing script or specify a new one. If you choose a new script, the wizard guides you through the script creation process. More information

* Launch Point: LP

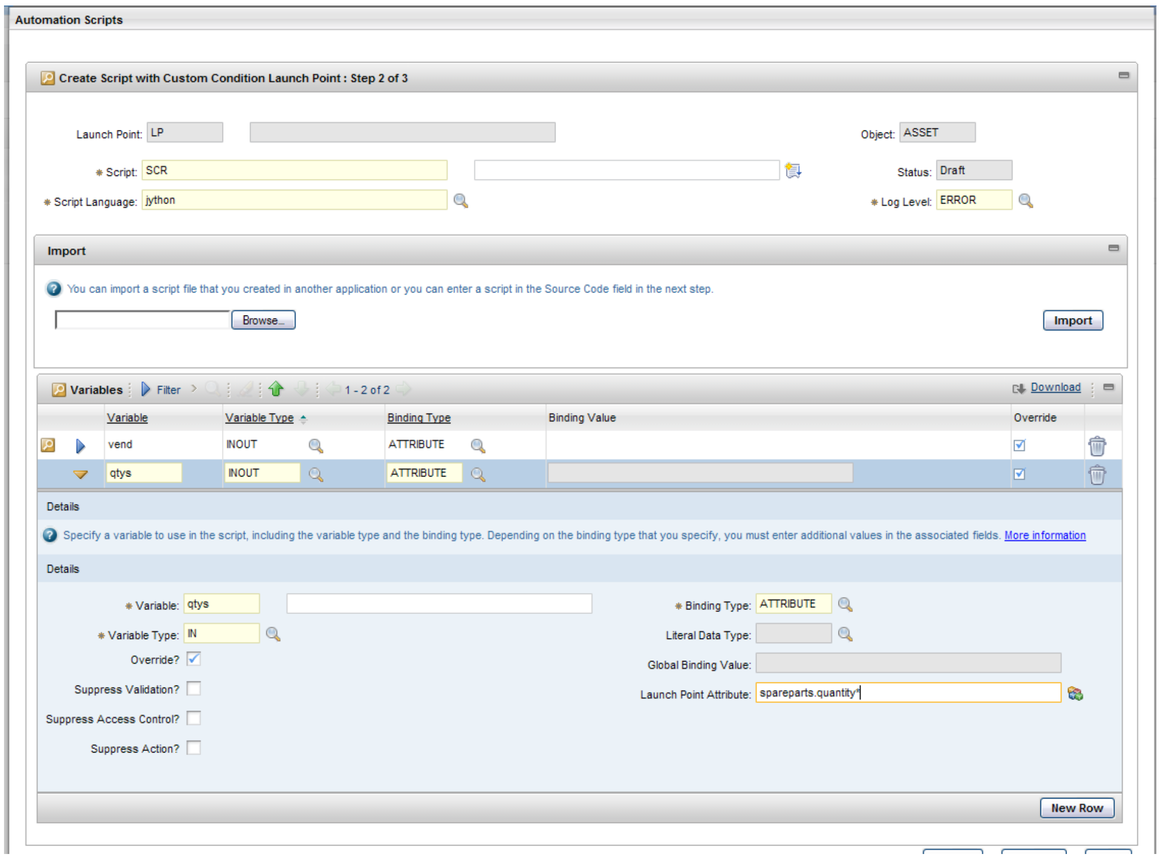
Active? ✓

* Object: ASSET

[✓ New] Existing: Script:

Step 2 will be to define and bind the variables. This script will involve 2 variables as listed below

Variable name	Variable type	Binding
vend	IN	vendor
qtys	IN	sparepart.quantity*


Automation Scripts
Create Script with Custom Condition Launch Point : Step 2 of 3

Launch Point: LP
Object: ASSET
Script: SCR
Status: Draft
Script Language: jython
Log Level: ERROR

You can import a script file that you created in another application or you can enter a script in the Source Code field in the next step.

Variable	Variable Type	Binding Type	Binding Value	Override
vend	INOUT	ATTRIBUTE		✓
qtys	INOUT	ATTRIBUTE		✓

Specify a variable to use in the script, including the variable type and the binding type. Depending on the binding type that you specify, you must enter additional values in the associated fields. More information

Variable: qtys
Binding Type: ATTRIBUTE
Variable Type: IN
Literal Data Type:
Override? ✓
Global Binding Value:
Suppress Validation?
Launch Point Attribute: spareparts.quantity*
Suppress Access Control?
Suppress Action?

Step 3 would be to define the script. The script would look like as below

```
if vend is not None and qtys is not None and sum(qtys)>10:
        evalresult=true
```

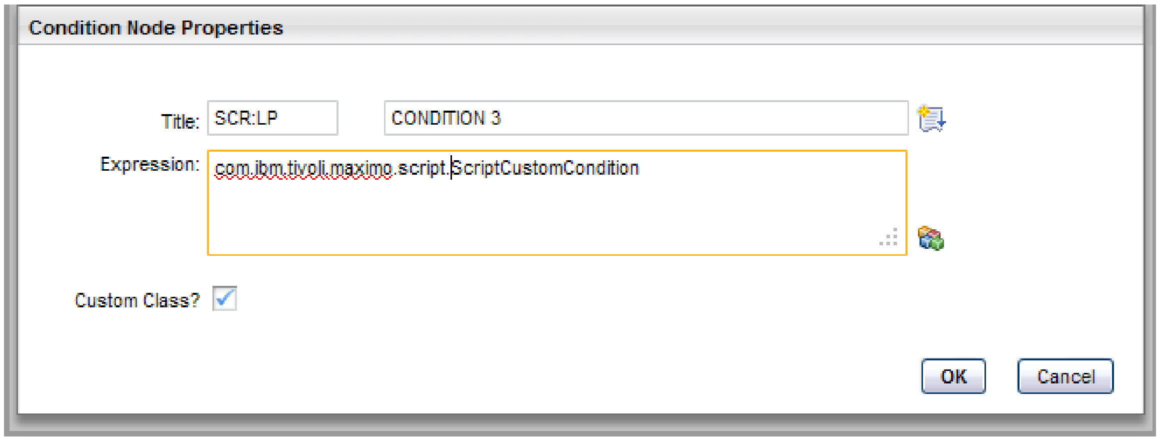
This evalresult as explained earlier is an implicit variable which carries the boolean result of the condition evaluation. Its predefined and is always there for condition launch points.

At the end of the wizard process when we submit the design – it would create a script with the logic as above. But the work is not done yet as we now have to attach the script to the actual condition which unfortunately is still a manual process. The steps are listed below:

1. create a condition node in the workflow designer.
2. Set the title of the condition node to <scriptname>:<launchpointname> where script name and the launch point name would point to the script+launch point pair just created.
3. Bring up the the condition node properties dialog [as shown below in the screen shot] to set the condition type as custom and set the custom Java class to com.ibm.tivoli.maximo.script.ScriptCustomCondition which is the predefined

proxy for the scripted conditions.

4. Save and activate the workflow.



Condition Node Properties

Title: SCR:LP CONDITION 3

Expression: com.ibm.tivoli.maximo.script.ScriptCustomCondition

Custom Class? ✓

One thing to note here is the title field value. This value is mapped to the WFNODE Mbo's title attribute which has a limit of 10 characters. And as you can see that the title is holding a pointer to the script launch point pair by appending the script name and the launch point name with the ":" as separator. Each of script name and launch point name can be 20 characters [out of the box setting]. So we do have a length issue here and at this point there is not much we can do but to keep the names of the script and launch point limited to 4 characters at maximum. Also if the script has one and only launch point we can just omit the launch point name from the title and just do with the script name. If all these have made you wonder why it was done this way as opposed to keeping an entry in the WFNODE table for the script name and the launch point name the answer is pretty simple – we just did not want to modify any existing Maximo artifacts.

The condition in the example was trivial and is meant to demonstrate the "how to do custom conditions using scripting" aspect. As you must have figured out – we can harness all the powers of scripting in this launch point and do all complicated evaluations needed to come out with the boolean result [evalresult] for the evaluation.

Activating and Deactivating scripts

Scripts can be activated or deactivated from the launchpoint tab of the scripting application. A script that is deactivated will not be invoked by Maximo. For example if a Attribute Launch point is deactivated – the script for that launchpoint will not be invoked when say the value of that mbo attribute changes. This is a very useful tool for debugging when you run into some trouble with the application behaviour and want to take the script out of the equation temporarily just to test the vanilla application devoid of all the customization. One thing to note here is the autoscript status attribute value has no significance as to how the scripting framework will [or will not] invoke the script. So a script in the draft state is treated the same as a script in “Active” state.

Debugging Scripts

By default all script related logging is done via the autoscript logger. Every script can be configured at different log levels – DEBUG,INFO,ERROR etc. The default setup for any script is at ERROR which can be changed from management application or at the time of creation [in the 2nd step of the wizard].

If we were to just debug the script below

```
y=float(iplr)+float(olr)
if y!=float(plr):
    pnr=str(y)
```

We can put some debug statements like

```
print “iplr=”+iplr
print “olr=”+olr
y=float(iplr)+float(olr)
print “y=”+y
if y!=float(plr):
    pnr=str(y)
    print “pnr=”+pnr
```

Next we need to make sure that the log level for the autoscript logger is set to the log level of the script. For example set both of them to INFO.

This will result in the print statements to show up in your systemout log. If required, the log statements produced by this logger can be re-directed to a dedicated log file holding only script-related log statements. Note that the syntax of the print statement depends upon which language the script is being written with. Also if the 'autoscript' logger is set to ERROR level logs only, then the print statements inside the automation script will not be written out to log file.

The scripting framework logs cover script loading, initialization, execution time and so forth. To see the values being passed received by script variables, set the autoscript logger to 'DEBUG', apply the logging settings and run the script configuration. Variable values should be output to the log file or system console. As a general rule of thumb we can set the autoscript logger to INFO during script development/debugging time and set inidvidual scripts log level to INFO too which will push the script specific print statements to the log file.

Below is the list of log information automatically generated at the DEBUG level from the scripting framework.

1. Launch point name and script name that are about to be executed
2. Script execution time (as time elapsed between start and end of script execution)
3. Application name, logged in use name, MBO name, MBO unique ID values always injected as implicit data to the script - script author may or may not use these
4. Variable values passed into the script based on variable bindings (variables that are sourced from MBO attribute, MAXVAR, system property or literal)

To redirect your scripting logs to a separate log file follow the steps below.

1. Go to Logging app.
2. Click Manage Appenders from the Select Action menu.
3. Click New Row in the Manage Appenders popup dialog.
4. Fill in Details as:
    - Appender = ScriptingOnly [or any appropriate name you choose]
    - Appender Implementation Class = psdi.util.logging.MXFileAppender [you can select this from the value list]
    - File Name = autoscript.log [or any appropriate file name you choose]
    - Accept all other defaults.
5. Save the new appender by clicking OK.
6. Locate 'autoscript' logger in the Root Loggers section of the application.
7. Click the Manage Appenders icon to the right of Appenders field in the Details section for 'autoscript' logger.
8. Select only the appender you created earlier (ScriptingOnly), and de-select any other appender previously associated with this logger.
9. Click OK to save the new association. From the Select Action menu click 'Apply' for the log settings to take effect.