



Medidata Rave®

Custom Functions Training Manual 10.0

Medidata Solutions Worldwide

Corporate Office
350 Hudson Street
New York, NY 10014
+1 212 918 1800

Medidata Solutions, Inc. Proprietary – Medidata and Authorized Clients Only. This document contains proprietary information that shall be distributed, routed or made available only within Medidata and its authorized clients, except with written permission of Medidata. February 2015

© Copyright 2015 Medidata Solutions, Inc. All rights reserved

Information in this document relates to Medidata Rave versions 5.6.3, 5.6.4, and 2012.1.0, and is subject to change without notice. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including, but not limited to, photocopying and recording, for any purpose without the express permission of Medidata Solutions, Inc.

Medidata, Medidata Solutions Worldwide, iMedidata, Medidata Balance, Medidata Clinical Cloud, Medidata Coder, Medidata CRO Contractor, Medidata CTMS, Medidata Designer, Medidata Insights, Medidata Grants Manager, Medidata Patient Cloud, Medidata Rave, Medidata Rave Monitor, Medidata Rave Safety Gateway, Medidata Rave Targeted SDV, Medidata University, and their respective logos are trademarks, or registered trademarks, of Medidata Solutions, Inc. All other brands or product names used in this document are trademarks, or registered trademarks, for their respective owners.

Disclaimer: Given that Rave allows users to translate and change text strings, the screens shown in this document are only a representation of the product and may not match the actual Rave interface you are currently using.



Table of Contents

Welcome.....	9
Using This Manual	9
Introduction to Custom Functions.....	11
The Custom Development Process	13
Introduction.....	15
The Rave Object Model	17
Introduction to the Rave Object Model.....	19
Object Model Diagram	20
Object Model Details	21
Rules That Apply to All Arrows in the Diagram	22
Bold Arrow: Definition Relationship	23
Example of Definition Relationship	23
Non-Bold Arrow: Ownership Relationship	25
Example for Arrow 1	25
Example for Arrow 2	26
Using the Object Model to Retrieve Data	27
Common Data Retrieval Tasks	29
Getting a DataPoint on the Same Record	29
Getting a DataPoint on a different DataPage in the same Instance	29
Getting a DataPoint from a different DataPage in a different Instance	30
Deriving a Custom Subject Identifier.....	33
Introduction.....	35
Example: Derived Subject ID.....	36
Sending a Serious Adverse Event Email Notification	49
Introduction.....	51
Example: SAE Email.....	52
Form Cross-Check Custom Functions	69
Introduction.....	71
Example: AE Symptom Required	72
Timespan Custom Functions	79
Introduction.....	80
Example: Diagnosis Date Check.....	81
Duplicate Log Line Functions	87
Introduction.....	88
Example: Duplicate Diagnosis Values	89
Performing Calculations with Custom Functions	95
Introduction.....	97
Example: Body Surface Area Calculation	98
Using the Dynamic Searchlist eCRF Control	103

Introduction	105
Example: Device Lookup	106
Writing Custom Function Code: Common Tasks	117
Looping Through EDC Collections	119
Finding Multiple Copies of a form	119
Example - Multiple Copies of a form	119
Looping Through Records	120
Example of looping Through Records	120
Localized Strings	121
Queries, Comments and Stickies	122
Obtaining CheckID and CheckHash	122
Queries	123
Comments	124
Stickies	124
Fetching Directly From the Database	125
Other Database Support	126
Sending Emails	127
Reusability	128
Within a Custom Function - Breaking a Function into Multiple Methods	128
Within a Draft – Calling One Custom Function from Inside Another	131
Using Architect Loader Across Projects and/or Drafts	132
Scenarios Requiring Multiple Custom Edit Checks	135
Determining When Multiple Checks are Required	137
Example	137
Building Multiple Custom Checks for a Specification	139
Example – CONMED and MEDHISTORY	140
Appendix A: Training Exercises	143
Exercise 1: End of Study Reason	144
Exercise 2: Death Date Timespan	146
Exercise 3: Duplicate log lines	148
Exercise 4: ANC Calculation	149
Exercise 5: SAE Reconciliation	151
Appendix B: Best Practices for Writing Checks	155
Understanding the Wildcard Match	157
How it works (Theory)	157
How it works (Explained)	157
How It Works (Example)	159
How it works (Example Check Steps)	159
Best Practices for Writing Checks	162
Field Edit Checks	162
One to Many DataPoint Comparisons ('Edit Check Hash')	162
Specifying Fields and Folders	163

Specifying Nested Folders	163
Use Data Values Correctly	163
Range and DataPoint Comparison Checks.....	163
Checkboxes	164
Constants as Check Steps	164
Set DataPoint Visible Check Action	164
Add Matrix vs. Merge Matrix Check Actions.....	164
Do not over Qualify	165
Reuse an Edit Check	165
A Single Form Checks the Same Field Across Multiple Folders	165
Multiple Forms Check the Same Field Across a Single Folder	166
Multiple Forms Check the Same Field Across Multiple Folders.....	167
Record Position- Checks between Logs and Standard Fields	168
Record Position	169
Form Repeat Number.....	169
Folder Repeat Number	170
Checks are Log Line Specific.....	170
Logical Record Position (LRP)	170
Qualify Steps and Actions the Same	171
Requires Response and Requires Manual Close	171
Use of HTML – is this still the case?	172
Do not Create Cyclical Derivations.....	172
Hidden Derivations or DataPoints in Edit Check	172
Placement of View Restricted Derivations	172
Derivations Between Log and Standard Fields	172
DaySpan VS AddDay in Derivations	172
Know the Wildcard Match	172
Appendix C: Commonly Used Objects, Properties and Methods	175
Commonly Used Objects, Properties and Methods	177
Searching for Data	180
Custom Function Actions	182
Generic Custom Functions	184
Dynamic Search List	185
Appendix D: The Development Utility	187
Using the Custom Function Development Utility	189
Setting Up the Utility	190
General Configuration of the Development Utility	192
Testing a Custom Function	195
Appendix E: Enhancements to Examples	203
SAE Email	205
Diagnosis Date Check	207
Duplicate Diagnosis.....	209

Welcome

This manual is designed to be a useful and detailed reference for the EDC module in Medidata Rave.

Using This Manual

The layout and format of this manual is intended to be as user friendly as possible. The following legend should help differentiate aspects of the information contained in the training manual:

These bullets show a list of definitions:

-
-

These bullets show a list of options:

-
-

These Numbered Lists Show General Step-by-step Procedures:

- 1.
- 2.

These bullets show a list of options for a step.

- ⇒
- ⇒

Note: Information in these gray boxes is to be noted or critical. Be sure to pay special attention to the instructions or information here.

Introduction to Custom Functions

In most cases, a clinical rule can be enforced by configuring an edit check or derivation through Rave's standard interface, without the use of custom code. For example, a standard edit check can be used to enforce the clinical requirement that the end of study date for a given subject occurs prior to the enrollment date.

Some situations, however, require that C# code be added to an edit check/derivation to achieve the desired functionality.

A custom function is piece of code added to a Medidata Rave Project to extend the functionality of a derivation or edit check.

Some scenarios that require the use of custom functions include:

- ☐ Making sure a field value is unique across all the log lines on a form
- ☐ Deriving a subject ID that includes the site number
- ☐ Performing some kinds of calculations

Setting up any of the scenarios above would involve writing custom function code in C#, adding the custom function to a Project Draft and then referencing the custom function in an edit check or derivation.

The Custom Development Process

- Introduction

Introduction

The process of writing custom functions for Medidata Rave can be broken down into a series of steps. These steps will be used to explain sample exercises in subsequent chapters.

Custom Development Process

Look at the data validation specification to gather information.

- ⇒ Make a list of all objects mentioned in the spec, including all folders, forms, fields, matrices, dictionaries. Note their object identifiers (OIDs).
- ⇒ Identify the business rule to be applied.

Add a new “blank” custom function to the project draft. Its source code should contain only “return true;”

This will serve as a placeholder for the actual code to be filled in later.

Build an edit check or derivation based on the specification. This edit check/derivation:

- ⇒ will serve to trigger execution of the custom function
- ⇒ must reference the custom function to perform the logic
- ⇒ must pass one parameter to the custom function

Write the C# custom function code.

- ⇒ This can be done in an integrated development environment (IDE), such as Microsoft Visual Studio C#.NET in conjunction with the Medidata Custom Function Development Utility.

Paste this code into the Project Draft. This code will replace the “return true;” code that was entered earlier as a placeholder.

Publish a new CRFVersion of the Project Draft, and then push this CRFVersion to a StudySite. For more information on Publishing and Pushing, refer to the Medidata Rave Architect Training Manual.

Add a new Subject on this StudySite, and test whether the business rule has been implemented correctly.

If the business rule has not been implemented correctly, return to steps 3 and 4, modifying edit check/derivation and/or code as necessary, then proceed with steps 5, 6, and 7 again.

A highly efficient complement to the cycle of publish

Function Development Utility, which enables developers to run code directly against a Rave Project without installing the code and testing it directly on the front-end. For more information, please refer to the “The Custom Function Development Utility” section in this manual.

☐push☐test is th

The Rave Object Model

- ❑ Introduction to the Rave Object Model
- ❑ Object Model Diagram
- ❑ Object Model Details
- ❑ Using the Object Model to Retrieve Data
- ❑ Common Data Retrieval Tasks

Introduction to the Rave Object Model

A custom function can only accept one data value as input parameter, so retrieving additional data for the current subject is a common task.

All clinical data in Rave is organized in a hierarchy of related C# objects, known as the Object Model or Core Object Model. To find a piece of information, a custom function must traverse different levels of this structure and access the representative object. This section describes the different kinds of Rave Core Objects, how they are related, and how to work with them in code.

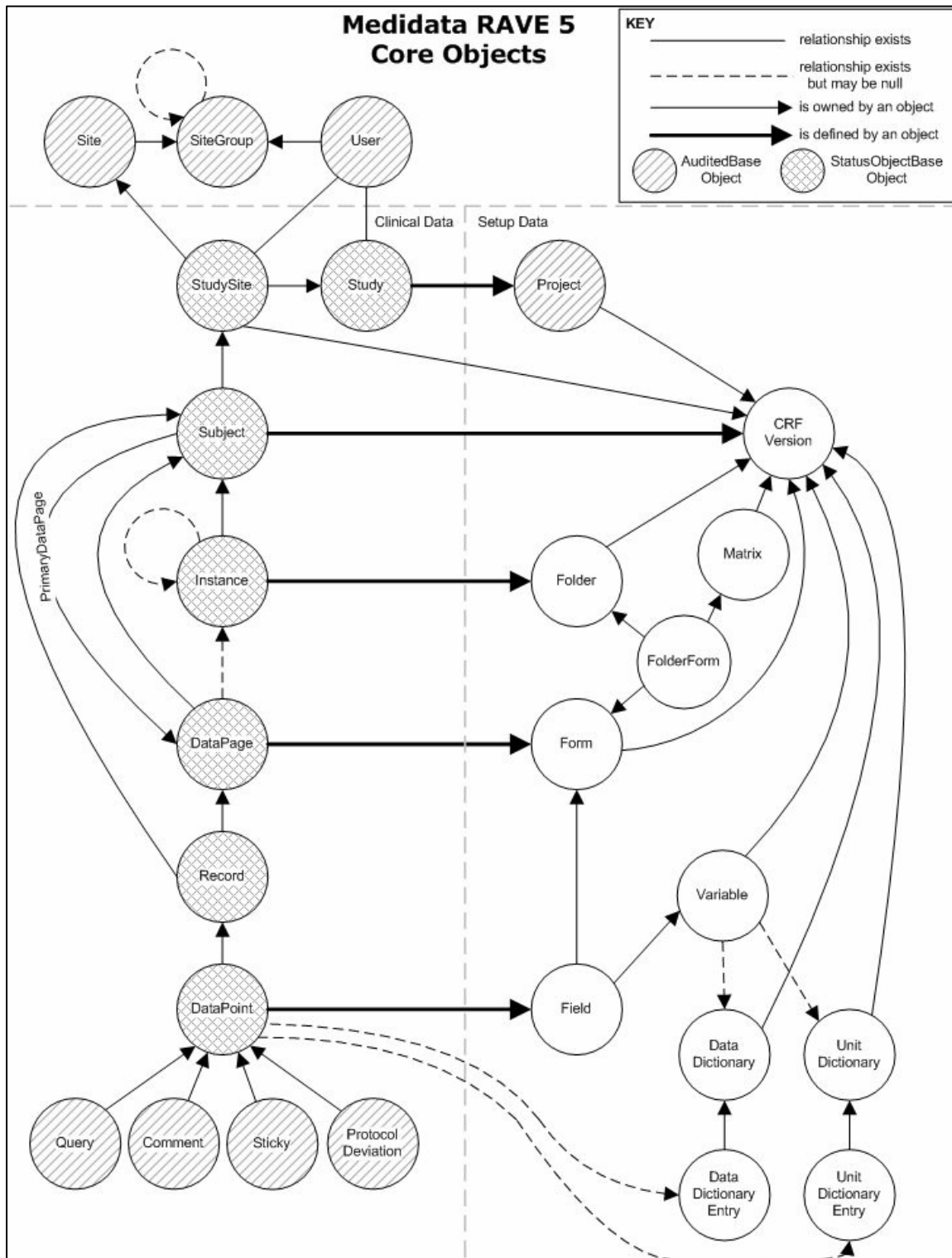
The key concepts are contained in the diagram entitled "Medidata Rave 5 Core Objects." Understanding how to get at subject data requires breaking down the diagram into its principal components, with definitions provided and patterns noted along the way.

The explanation of the diagram has four parts: background information, rules governing arrows, definition relationships, and ownership relationships.

This section concludes with an example showing how to use the diagram and the model it depicts to perform a common task: taking a data value passed in from an edit check and obtaining another piece of data located on the same log line.

Object Model Diagram

The object model diagram below shows the relationships between the Rave Core Objects. Refer to following sections in this manual for additional information.



Object Model Details

To use the object model diagram, assume the following:

- Every circle in the diagram represents a class with the same name.
- Rave provides the custom function developer with predefined classes for real- world clinical objects. There is a Subject class, a Study class, a Query class, a Site class, a User class, etc.
- In code, a CF developer will work with instances of these classes.

Circles on the left side of the dotted line are EDC classes; circles on the right side of the dotted line are Architect classes.

- This is an important distinction.
- Example: outside the context of custom function programming, a 'form' is considered a page where data is entered. The Rave class Form represents the definition of such a page in Architect and all its metadata. Any particular subject's copy of this Form is an instance of DataPage, an entirely different class.
- To understand why two classes are necessary, it is helpful to look at the Created property, one that is available to both Form and DataPage objects.
- One example could be an Adverse Events form. In this case, for the corresponding Form object, Created would return the date and time when the user defined the Adverse Events form in Architect, before adding fields, etc. For an individual subject, the Created property of the representative DataPage would return the date and time that the page's matrix was added to that subject.

For every class, there exists another, plural class.

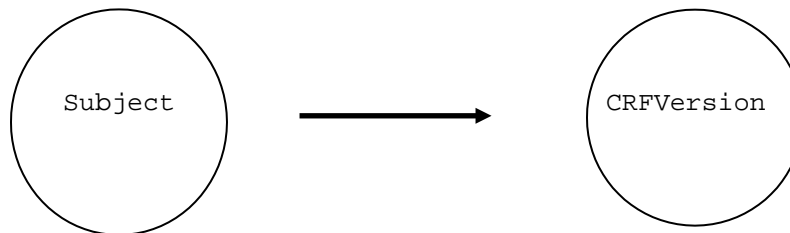
- For example, in addition to `Sticky`, Rave has another class called `Stickies`.
- An instance of `Stickies` represents a collection of `Sticky` objects, and since it implements the Microsoft .NET system interface `ICollection`, it supports resizing, searching, and other features not available with a standard fixed-length array.

Rules That Apply to All Arrows in the Diagram

The following rules apply to all arrows in the diagram, both bold *and* non-bold.

- If two classes are connected by an arrow, **————→** they are related, either by Architect definition or by EDC ownership (parent/child).
- This relationship is contained in a C# property.
- This relationship is many-to-one.

One example is the relationship between Subject and CRFVersion:



Applying the three rules above to this case,

- ❑ A CRFVersion object is said to define all of the instances of Subject on that version.

This makes sense because a CRFVersion is just a collection of setup data that describes how data will be entered for subjects on the front end.

- ❑ Any Subject has a property called 'CRFVersion' that returns the instance of CRFVersion that defines that Subject.

For the sake of example, assume that a custom function has a reference to the current subject called `currentSubject`. This is a variable of type Subject.

To get the CRFVersion of this current subject, use the CRFVersion property:

```
CRFVersion currentVersion =  
    currentSubject.CRFVersion;
```

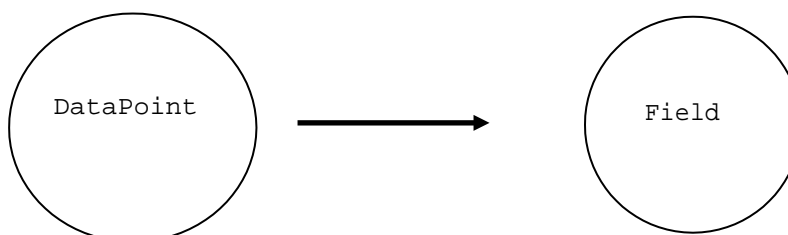
Finally, many instances of the Subject class can exist on a single CRFVersion.

Bold Arrow: Definition Relationship

The following rule applies specifically to **bold** arrows in the diagram:

- ❑ Any bold line (—→) represents a **definition** relationship. It points *from* an EDC object *to* the Architect object that defines it.

DataPoint and **Field** are two classes that share this relationship:



Example of Definition Relationship

All studies provide some place to enter the subject's birthday, and one setup might be a single field BIRTHDT on a single form DEMOG.

Since C# is object-oriented, the developer should ask "What information can I retrieve using the object model?"

For the BIRTHDT "field" mentioned above, there exists only corresponding **Field** object for a given **CRFVersion**, but there can be many **DataPoint** objects. More precisely, there are exactly as many **DataPoint** objects containing birth dates as there are subjects on that particular **CRFVersion**.

This is a direct consequence of what a **DataPoint** and a **Field** actually represent.

Note: A **Field** object represents the *definition*, or setup, of an eCRF field in Architect.

The **Field** class defines properties to access relevant characteristics – the field label, length, format, etc.

Note: A **DataPoint** object represents a single piece of clinical data, serving as a container for the value entered into an EDC field.

The **DataPoint** class defines properties to determine, among other things:

- ❑ the data entered
- ❑ its status (locked, nonconformant, queried, etc)
- ❑ if the data is active (false if the data is on an inactivated log line or page)

This class also provides methods to do things like:

- ❑ change the value entered
- ❑ open queries, comments and stickies
- ❑ perform an entry lock, hard lock, or unlock

The general approach for getting subject data involves using both definition relationships – like the one between **DataPoint** and **Field** – *and* ownership relationships. This second kind of connection is discussed in the section immediately following this one. To provide an example before moving forward, a piece of code is presented that doesn't actually retrieve subject data. Instead it simply determines the OID of the defining field for a **DataPoint** passed in from a check action-referenced custom function:


```
ActionFunctionParams afp = ActionFunctionParams.ThisObject;  
DataPoint inputDP = afp.ActionDataPoint;  
Field definingField = inputDP.Field;  
string fieldOID = definingField.OID;
```

Note here that:

- ❑ All **Field** objects (and in fact, all Architect setup objects) have a property **OID**.
- ❑ To get the defining **Field** object for **inputDP**, a property is accessed with that very same name – i.e., **Field**. This is by design, not coincidence.

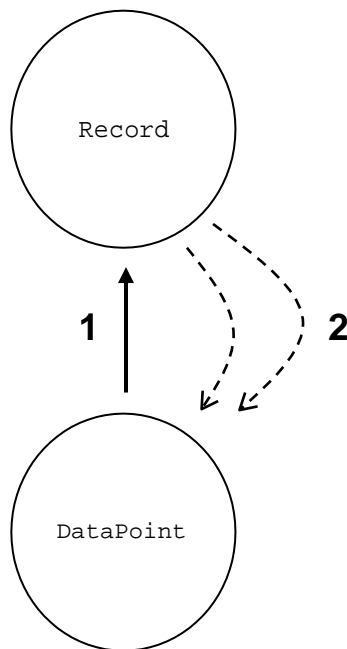
Non-Bold Arrow: Ownership Relationship

The following rule applies specifically to *non-bold* arrows in the diagram:

- ❑ A non-bold line () represents an **ownership** relationship. It points *from* a child EDC object *to* a parent EDC.
- ❑ This is a parent/child relationship in the context of real-world clinical data entry, and does *not* imply object-oriented inheritance.

DataPoint and **Record** are two classes that share this relationship:

- ❑ Many DataPoint objects can be contained within a single instance of a Record.
- ❑ DataPoint does *not* represent a subclass of Record.



Note: To reduce clutter, the dashed lines shown here do NOT appear in the actual object model diagram. However, the relationships they describe are important!

The single arrow labeled **1** is used to get the parent object of something in EDC.

The arrows labeled **2** (implied, but not shown in the full object model diagram) let the developer retrieve all of the children of an EDC object.

Example for Arrow 1

To get the **Record** object that contains a given **DataPoint**, the 'Record' property is used:

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
```

```
//Passed in from DOSE field on CONMED form (log)  
DataPoint dpDose = afp.ActionDataPoint;
```

```
//Get the current Record  
Record recCurrent = dpDose.Record;
```

Example for Arrow 2

The dotted arrows indicated by **2** above mean that programmers can retrieve child objects in the form of the “plural classes” described in the General Information part of this section.

The **'DataPoints'** property is used to get an instance of **DataPoints**.

This will contain all of the child **DataPoint** objects inside **recCurrent** - not only the one for the DOSE field, but ones for all the fields – e.g., FREQUENCY, TIME, and MEDNAME.

```
//Get all of the DataPoints inside recurrent  
DataPoints allDPs = recCurrent.DataPoints;
```

Using the Object Model to Retrieve Data

To get subject data, it is necessary to take advantage of the definition *and* ownership relationships in the object model.

Example:

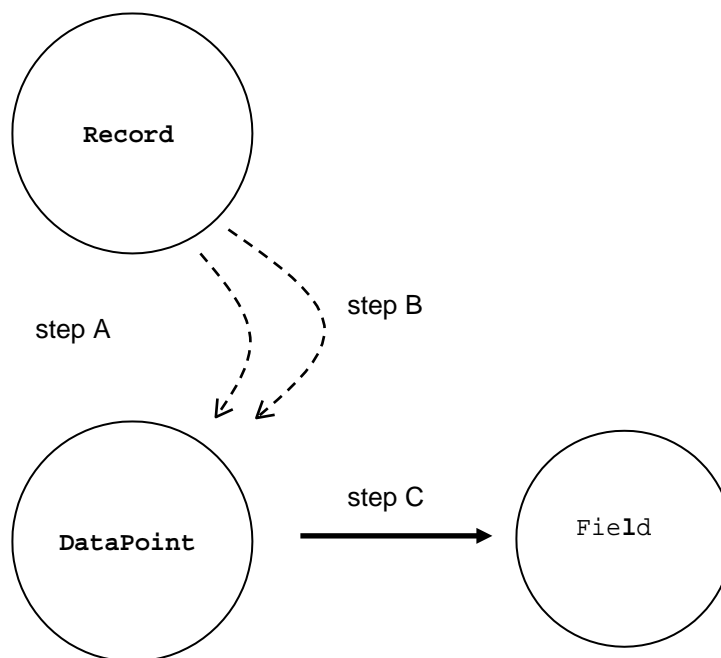
A Data Point defined by the DOSE field is passed to a custom function that needs to retrieve the FREQUENCY **DataPoint** on the same **Record**.

To accomplish this, the function must:

- ❑ Get the parent **Record** of the DOSE DataPoint (step A)
- ❑ Get the **DataPoints** object containing all of the children of this **Record** (step B)
- ❑ Pick out the desired **DataPoint** from this collection (step C)

```
DataPoint dpDose = afp.ActionDataPoint; Record  
recCurrent = dpDose.Record; //Step A  
DataPoints dpsAll = recCurrent.DataPoints; //Step B  
//Step C  
DataPoint dpFrequency = dpsAll.FindByFieldOID("FREQUENCY");
```

The example above illustrated how to start with one **DataPoint** object and get a different one on the same **Record**.



starting point = dpDose

finish = dpFrequency

The approach used above can be applied to the general case of starting with *any* type of EDC object – **DataPoint**, **Record**, **DataPage**, **Instance**, or even **Subject** – and navigating through the different levels of the “tree” to any other EDC object.

This general strategy mirrors the steps A, B and C followed above:

- ❑ Navigate up the tree as far as necessary, then back down, to retrieve a collection of EDC objects – the “plural class” mentioned earlier
- ❑ Select the desired EDC object from this collection by using one of three methods:
 - FindByFieldOID (to find a DataPoint)
 - FindByFormOID (to find a DataPage)
 - FindByFolderOID (to find an Instance)

Common Data Retrieval Tasks

The following scenarios appear very frequently when retrieving data:

Getting a DataPoint on the Same Record

See the DOSE DataPoint example in the “Using the Object Model to Get Data” section, above.

Getting a DataPoint on a different DataPage in the same Instance

This situation occurs when a custom function needs to retrieve data from a different page than the one containing the data value passed in by the edit check. For example, checking a subject's gender when a pregnancy date is entered.

Assumptions:

- ❑ The custom function is passed a DataPoint whose defining Field has an OID of "PREG_DATE".
- ❑ The goal is to retrieve the DataPoint whose defining Field has OID "GENDER" on a DataPage whose defining Form has OID "DEMOG".
- ❑ Also, assume both DataPages are subject level pages.
- ❑ Finally, assume Male = 1 as DataDictionary value.

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;

//Get the DataPoint passed in from the edit check. DataPoint
dpPregDate = afp.ActionDataPoint;

//Get a reference to the current subject.
Subject currentSubject = dpPregDate.Record.Subject;

//Now get all DataPages for this subject. DataPages
allPages = currentSubject.DataPages;

//Now search for a DataPage with Form OID "DEMOG". DataPage
dpgDEMOG = allPages.FindByFormOID("DEMOG");

if (dpgDEMOG != null && dpgDEMOG.Active)
{
    //Use .MasterRecord to get the standard record Record
    masterRecord = dpgDEMOG.MasterRecord;

    //Now search for a DataPoint with Field OID "GENDER". DataPoint dpGender
    =
    masterRecord.DataPoints.FindByFieldOID("GENDER");
    //Do something with the found DataPoint
}
return null;
```

Getting a DataPoint from a different DataPage in a different Instance

This is similar to the situation above, except additionally, the page containing the desired data is in a different Instance than the one containing the data value passed in from the edit check.

Assumptions:

- ❑ this custom function is run as a check action that is passed the SYSTOLIC from the VITALS form inside of the SCREENING folder
- ❑ the function needs to obtain the HEARTRATE from the ECG form (non-log) inside of the BASELINE folder

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
DataPoint dpSystolic = afp.ActionDataPoint;

Instance instBaseLine =
    dpSystolic.Record.Subject.Instances.FindByFolderOID
        ("BASELINE");

DataPage dpgECG;
if (instBaseLine != null && instBaseLine.Active)
{
    dpgECG =
        instBaseLine.DataPages.FindByFormOID("ECG");
    DataPoint dpHeartRate;
    if (dpgECG != null && dpgECG.Active)
    {
        dpHeartRate =
            dpgECG.MasterRecord.DataPoints.
                FindByFieldOID("HEARTRATE");

        //Do some work
    }
}
return null;
```


Deriving a Custom Subject Identifier

- ❑ Introduction
- ❑ Example: Derived Subject ID

Introduction

This section presents an example of a custom function that derives a subject identifier from the site number, subject's initials and subject's birth date.

This custom function is developed to be used with a derivation.

Example: Derived Subject ID

One example of a derivation requiring the use of a custom function is the case of a derived subject identifier that contains the site number and the subject number in the StudySite; neither is available as standard derivation steps on the Rave front-end. (Please see the Medidata Rave Architect Training Manual for a list of standard derivation steps).

A sample specification for this example follows. After the sample specification, the example is set up and explained by going through the steps of the Custom Development Process outlined in the previous section.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input type="checkbox"/> Edit Check <input checked="" type="checkbox"/> Derivation
Custom Function Name:	Subject ID
Short Description:	Derive the SUBJID field and set the Subject Name property.

IF a Derivation, please provide General Derivation Requirements:	
Field(s) to be derived:	SUBJID Field on SUBJID Form
Type of Derivation:	<input type="checkbox"/> N/A <input type="checkbox"/> Calculation <input checked="" type="checkbox"/> Data Concatenation <input type="checkbox"/> Other (Specify):

Elements involved in the Custom Function:			
Folder OID(s):			
Form OID:	SUBJID		
Form Name:	Subject ID		
Field OID(s)	Pretext	Is Log?	Action
SUBJINI	Subject Initials	<input type="checkbox"/>	None
BIRTHDT	Birth Date	<input type="checkbox"/>	None
SUBJID	Subject Identifier	<input type="checkbox"/>	Set Value

Business Rules:

SUBJID Field Should Be Derived As Follows:

$\text{SUBJID} = \{\text{Site Number}\} - \{\text{Subject Initials}\} - \{\text{Birth Date}\}$

The subject name property should be set to this derived SUBJID value.

Special Instructions

The Subject Initials are to be obtained from the SUBJINI field.

The Birth Date is to be obtained from the BIRTHDT field.

The Site Number, is not a field, but a property of a site, to be obtained in the custom function through the Rave Object Model.

The Birth Date should be padded on the left with zero to ensure it is always 11 digits.

For Example, if Birth Date is submitted as "1 Jan 1950", Subject Identifier should be derived as "001-ABC-01 Jan 1950".

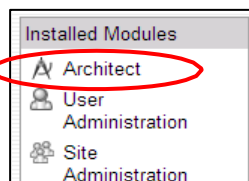
To set up the derivation and custom function:

Look at the specification to gather information.

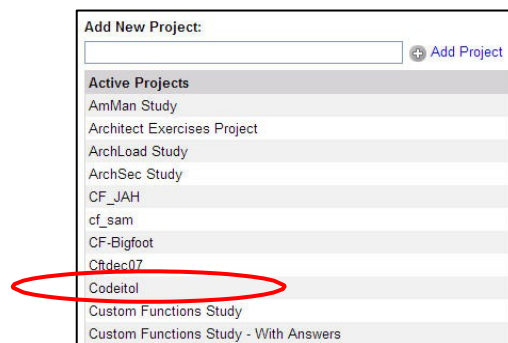
- ⇒ All three fields mentioned, SUBJINI, BIRTHDT and SUBJID are on the SUBJID form.
- ⇒ The site number needs to be retrieved.
- ⇒ Site number, initials and birth date must be concatenated, and this value should be derived into SUBJID.
- ⇒ The Subject Name should also be set to this derived value.

Add a new “blank” custom function to the project draft.

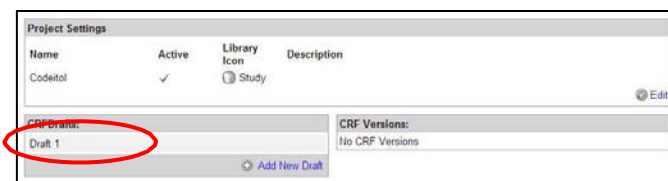
- ⇒ From the Rave homepage, navigate to the Architect Module by clicking on the appropriate link on the sidebar.



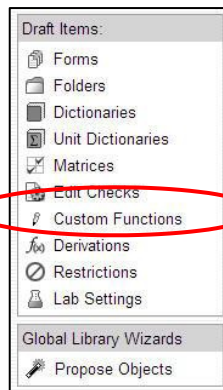
- ⇒ Select the appropriate Project from the list of Active Projects.



- ⇒ Select the **CRFDraft** where the custom function will be installed.



- ⇒ Select **Custom Functions** from the list of Draft Items.



⇒ Click Add Custom Function.

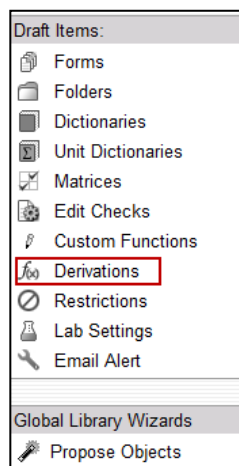
A screenshot of the 'Add Custom Function' dialog box. It has a 'Show All' checkbox, a 'Name' field, a 'Language' dropdown, and an 'Edit' button. The 'Add Custom Function' button is circled in red.

⇒ Enter "**Subject ID**" for the **Name** and `return true;` in the source code box below it. Then click **Update**. This will be just a placeholder for the custom function, so the derivation can be built. The actual code you'll submit later.

A screenshot of the 'Add Custom Function' dialog box. The 'Name' field contains 'Subject ID' and the 'Source Code' box contains 'return true;'. The 'Update' button is circled in red.

Build a derivation based on the specification.

⇒ On the left side of the screen, select **Derivations** from the list of Draft Items.



⇒ Click Add Derivation.

The screenshot shows the top of the Derivations table. The 'Add Derivation' button, represented by a plus icon, is circled in red. Other buttons like 'Bypass During Migration', 'Active', 'Apply To Variable', 'Edit', and 'Derivation Steps' are also visible.

⇒ For the **Name** of the derivation, enter "**Subject ID**". Leave **Bypass During Migration** unchecked, and leave **Active** checked. For the **Apply To Variable**, specify the SUBJID field on the Subject ID form. Put "0" for Record Position.

This screenshot shows the configuration for the 'Subject ID' derivation. The 'Name' is 'Subject ID'. 'Bypass During Migration' is unchecked, and 'Active' is checked. Under 'Apply To Variable', the 'Form' is 'Subject ID', the 'Field' is 'SUBJID', and the 'Variable' is 'SUBJID'. The 'Rec. Pos.' is set to '0'. The 'Form Repeat Number' and 'Folder Repeat Number' are blank. 'Apply To All Folders' and 'Apply To All Fields' are unchecked.

⇒ Leave the **Form Repeat**, and **Folder Repeat** boxes blank. Also leave **Apply to All Folders** and **Apply to All Fields** unchecked. Assume all of this in all subsequent examples unless otherwise stated. Then, click **Update**.

This screenshot is similar to the previous one, but the 'Update' button, represented by a checkmark icon, is circled in red.

⇒ Click the detail arrow for **Derivation Steps**.

The screenshot shows the Derivations table with one row for 'Subject ID'. The 'Derivation Steps' button, represented by a play icon, is circled in red.

⇒ Now, all fields, on which the derivation depends, have to be put into the derivation steps. Click Add Derivation Step.

The screenshot shows a table with columns 'Type', 'Step', and 'Edit'. The 'Add Derivation Step' button, represented by a plus icon, is circled in red.

⇒ For Type, select Data Value.

Type	Step	Edit
<div> <div>Constant</div> <div>Data Value</div> <div>Constant</div> <div>Step Function</div> <div>Custom Function</div> </div>	<div>Value</div> <div>0</div> <div>Format</div> <div>0</div>	<div> <input checked="" type="checkbox"/> Update <input type="checkbox"/> Cancel <input type="checkbox"/> Delete </div>

⇒ Select **Subject ID** form, **BIRTHDT** field, "0 for Record Position and **DataPoint** for Data Value.

Step									
Folder	Form	Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number	Logical Record Position	Record Position	Data Value
...	Subject ID	BIRTHDT	BIRTHDT	0			None		Data Point
<input type="checkbox"/> Apply To All Folders <input type="checkbox"/> Apply To All Fields									

⇒ Click Update

									Edit
Form	Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number	Logical Record Position	Record Position	Data Value	
Subject ID	BIRTHDT	BIRTHDT	0			None		Data Point	
<input type="checkbox"/> Apply To All Fields									<input checked="" type="checkbox"/> Update

⇒ Click Add Derivation Step to add another field.

Type	Step	Edit
<div> <div>▲▼</div> <div>Data Value (Data Point)</div> <div>+</div> <div>Add Derivation Step</div> </div>	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None	<input checked="" type="checkbox"/> Update

⇒ For Type, select Data Value.

Type	Step
Data Value (Data Point)	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None
<div> <div>Data Value</div> <div>Data Value</div> <div>Constant</div> <div>Step Function</div> <div>Custom Function</div> </div>	<div>Folder</div> <div>Form</div> <div>Field</div> <div>...</div> <div>...</div> <div>...</div> <div> <input type="checkbox"/> Apply To All Folders <input type="checkbox"/> Apply To All Fields </div>

⇒ Fill out the details of this step as following, and click Update.

Step									
...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None									
Folder	Form	Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number	Logical Record Position	Record	Data Value
...	Subject ID	SUBJINI	SUBJINI	0			None		Data Point
<input type="checkbox"/> Apply To All Folders <input type="checkbox"/> Apply To All Fields									

⇒ Click Add Derivation Step.

Type	Step	Edit
Data Value (Data Point)	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None	
Data Value (Data Point)	...>Subject ID>SUBJINI>SUBJINI>0>...>...>None	
Add Derivation Step		

⇒ For the **Type** of this step, select **Custom Function**.

Type	Step	Edit
Data Value (Data Point)	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None	
Data Value (Data Point)	...>Subject ID>SUBJINI>SUBJINI>0>...>...>None	
<div> <div>Constant</div> <div> Data Value Constant Step Function Custom Function </div> </div>	<div>Value</div> <div>0</div> <div>Format</div> <div>0</div>	<input checked="" type="checkbox"/> Update <input type="checkbox"/> Cancel <input type="checkbox"/> Delete

⇒ For the custom function **Name**, select **Subject ID**, and for **Input**, select **From Previous Step**. Then click **Update**.

Type	Step	Edit
Data Value (Data Point)	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None	
Data Value (Data Point)	...>Subject ID>SUBJINI>SUBJINI>0>...>...>None	
Custom Function	<div>Name</div> <div>Subject ID</div> <div>Input</div> <div>From previous Step</div>	<input checked="" type="checkbox"/> Update

⇒ This is the completed derivation:

Type	Step
Data Value (Data Point)	...>Subject ID>BIRTHDT>BIRTHDT>0>...>...>None
Data Value (Data Point)	...>Subject ID>SUBJINI>SUBJINI>0>...>...>None
Custom Function (From previous Step)	Subject ID
Add Derivation Step	

Write the C# custom function code.

```
//Get SUBJINI DataPoint passed in from Rave derivation
DataPoint dpSUBJINI = (DataPoint)ThisObject;

//Get the reference to the current Subject
Subject currentSubject = dpSUBJINI.Record.Subject;

//Find the datapoint of the Birth Date field
DataPoint dpBIRTHDT =
    dpSUBJINI.Record.DataPoints.FindByFieldOID("BIRTHDT");

//Get the Birth Date data string and pad left with zeros
//so it is always 11 characters long
string strBIRTHDT = dpBIRTHDT.Data.PadLeft(11, '0');

//Get number of the site to which subject belongs
string strSiteNumber = currentSubject.StudySite.Site.Number;

//Concatenate all values to create the subject identifier
string strSubjectID =
    strSiteNumber + "-" + dpSUBJINI.Data + "-" + strBIRTHDT;

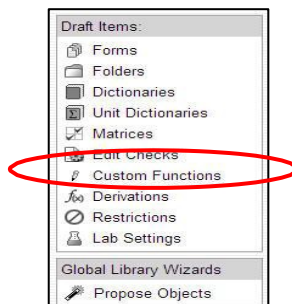
//Set "Name" property of current Subject,
//which will be shown on front-end.
currentSubject.Name = strSubjectID;

//The derivation will assign this returned value to the SUBJID field
return strSubjectID;
```

Note: For custom functions used in derivations and in edit check steps, the passed parameter is DataPoint. For custom functions used in edit check actions, the passed parameter is ActionFunctionParams, with the exception of "Set DynamicSearchList" action, for which the parameter is DynamicSearchParams.

Paste the code into the Project Draft.

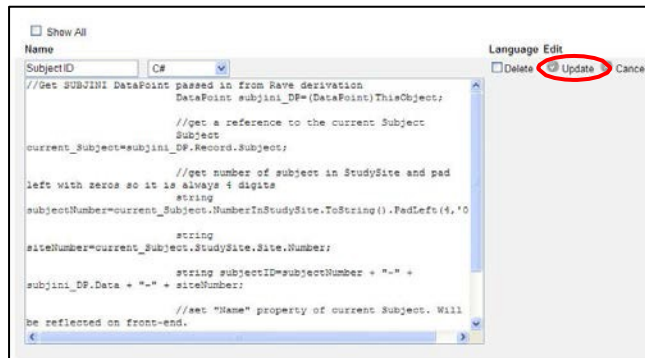
- ⇒ Copy the code above.
- ⇒ From the list of **Draft Items**, select **Custom Functions**.



⇒ Click the **Edit** pencil next to the **Subject ID** custom function.



⇒ Paste the code into the source code window, and then click **Update**.



Publish a new CRFVersion of the Project Draft, and then push this CRFVersion to a StudySite. For more information on Publishing and Pushing, refer to the Medidata Rave Architect Training Manual.

Add a new Subject on this StudySite, and test whether the business rule has been implemented correctly.

- ⇒ Navigate to the Rave home page.
- ⇒ Select appropriate Study and Site. Refer to the Medidata Rave EDC Training Manual for more information on navigating to Studies and Sites.
- ⇒ Click Add Subject.



- ⇒ Fill out **Subject Initials** and **Birth Date** fields. Then click **Save** to create the subject.

Subject Status: New

Subject: **New Subject**
Page: **Subject ID**

Subject Initials: ABC

Birth Date: 1 Jan 1950

Subject Identifier: [Icons]

[Printable Version](#) [Icon Key](#)

CRF Version 2130 - Page Generated: 23 Dec 2010 14:14:33 Eastern Standard Time

Save **Cancel**

- ⇒ Click OK on the pop-up window to confirm the subject entry.

Subject Status: New

Subj
Pag

The page at <https://training1.mdsol.com> says:

Are you sure you wish to save this subject?

OK **Cancel**

Birth Date: 1 Jan 1950

Subject Identifier: [Icons]

[Printable Version](#) [Icon Key](#)

CRF Version 2130 - Page Generated: 23 Dec 2010 14:14:33 Eastern Standard Time

Save **Cancel**

- ⇒ Note the last tab on the right above the subject page is now updated to the new subject name.
- ⇒ To verify that this subject name matches the SUBJID field, click "Subject ID" link to navigate to the primary form.

001-ABC-01 Jan 1950

Adverse Events

EOS / Withdrawal

Screening

Labs

Death Summary

Subject ID

Visit	Date

- ⇒ Confirm that the derivation and custom function are working by matching the **Subject Identifier** field with the subject name displayed in the tab bar.

The screenshot displays the 'Subject ID' page in the Medidata Rave interface. The top navigation bar includes 'Training', 'Test Site', and a tab labeled '001-ABC-01 Jan 1950' which is circled in red. Below the navigation bar, the 'Subject Status' is 'Enrolled'. The main content area shows the subject details: 'Subject: 001-ABC-01 Jan 1950' and 'Page: Subject ID'. A table lists the subject's information:

Subject Initials	ABC	✓						
Birth Date	1 Jan 1950	✓						
Subject Identifier	001-ABC-01 Jan 1950	✓						

The 'Subject Identifier' value '001-ABC-01 Jan 1950' is circled in red, matching the tab label. At the bottom, there are links for 'Printable Version', 'View PDF', and 'Icon Key', along with 'Save' and 'Cancel' buttons. The footer indicates 'CRF Version 2130 - Page Generated: 23 Dec 2010 14:30:00 Eastern Standard Time'.

Sending a Serious Adverse Event Email Notification

- ❑ Introduction
- ❑ Example: SAE Email

Introduction

The previous section presented an example showing how to use a custom function in a derivation.

This section covers a sample custom function that is installed as part of an edit check.

There are two ways to use a custom function in an edit check:

- ❑ Add the function as a **check step** – the “If” part of a Rave edit check
- ❑ Add the function as a **check action** – the “Then” part

The example code in this section is added as a **check action**.

Example: SAE Email

It is common to send an email notification when a serious adverse event is logged in Rave. This requires a custom function.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	SAE Email
Short Description:	Send an email when a serious adverse event is logged.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input type="checkbox"/> Yes <input type="checkbox"/> No <input checked="" type="checkbox"/> N/A
Manual Close Query Required?	<input type="checkbox"/> Yes <input type="checkbox"/> No <input checked="" type="checkbox"/> N/A
Marking Group(s)	N/A
Please specify action message (if applicable): N/A	

Elements involved in the Custom Function:			
Folder OID(s):	AE		
Form OID:	AE		
Form Name:	Adverse Event Log		
Field OID(s)	Pretext	Is Log?	Action
SERIOUS	Is this a serious AE?	<input checked="" type="checkbox"/>	Other Send email when this field is checked.

Business Rules:

When “Is this a serious AE?” is checked, following email should be sent:

Subject: SAE Added! Site: {site number}, Subject: {subject name}

Body:

A new serious AE has been recorded by {login name}.

{List labels and values of all fields on this log line, except for “Is this a serious AE?”}

If any field changes on log line where “Is this a serious AE?” is checked, Follow-Up email should be sent:

Subject: SAE Updated! Site: {site number}, Subject: {subject name}

Body:

An existing serious AE has been updated by {login name}. The following fields have been updated:

{List labels and values of all fields which has been updated on this log line}

Use the following email addresses:

From: rave@mdsol.com

To:

If the subject is in the DEV environment, the email should go to
user_dev@mdsol.com.

If the subject is in the PROD environment, the email should go to
user_prod@mdsol.com.

If the subject is in any other environment, the email should go to
user_any@mdsol.com.

Note: To simplify this custom function for training purpose, whenever field “Is this a serious AE?” is re-submitted as checked, consider it as a new AE. No email should be sent on inactivation, downgrade or upgrade.

To set up the edit check and custom function:

Look at the specification to gather information.

- ⇒ All fields are on the AE form in the AE folder.
- ⇒ If the AE is *new*, all field values on the current AE record need to be retrieved. If the AE is *updated* – e.g., if the date is changed – only the values for updated fields need to be retrieved. The site number and subject name also need to be retrieved.
- ⇒ The exact text of the email will differ depending on whether the AE is new or updated.
- ⇒ The intended recipient depends on what Rave environment the current subject is in.

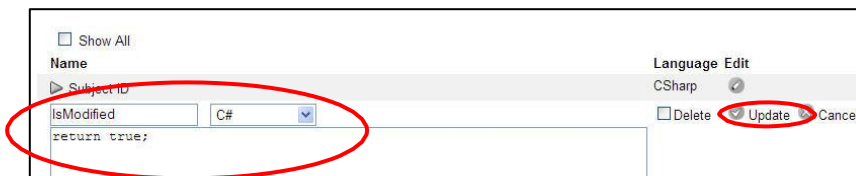
Add a new “blank” custom function to the project draft.

- ⇒ From the Rave homepage, navigate to the Architect Module by clicking on the appropriate link on the sidebar.
- ⇒ Select the appropriate Project from the list of Active Projects.
- ⇒ Select the **CRFDraft** where the custom function will be installed.
- ⇒ Select **Custom Functions** from the list of Draft Items.
- ⇒ Click Add Custom Function.



Note: The first custom function added as part of this exercise is not the custom function that will send an email, but a special custom function, *IsModified*. This custom function *need only be installed once per project draft*.

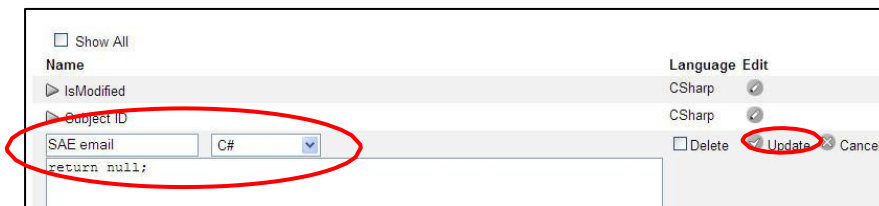
- ⇒ For the **Name**, enter **IsModified**. In the source code text box below, enter `return true;`. Then click **Update**.



⇒ Click **Add Custom Function**.

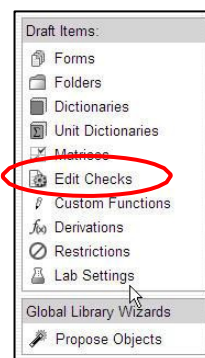


⇒ For the **Name**, enter **SAE Email**. In the source code text box below, enter `return null;`. Then click **Update**.

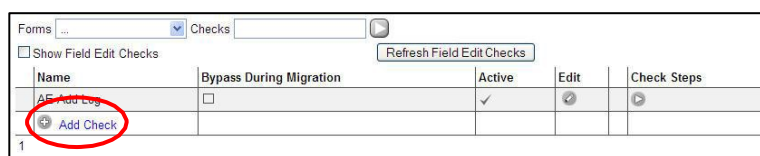


Create an edit check based on the specification.

⇒ Click **Edit Checks** from the **Draft Items** sidebar on the left side of the screen.



⇒ Click **Add Check**.



- ⇒ For **Name**, enter **SAE Email**. Leave **Bypass During Migration** unchecked, and leave **Active** checked. Then click **Update**.

Name	Bypass During Migration	Active	Edit	Check Steps
SAE Add Log	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
SAE email	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Update Cancel Delete	

- ⇒ Click the detail arrow to create **Check Steps**.

Name	Bypass During Migration	Active	Edit	Check Steps
SAE email	<input type="checkbox"/>	<input checked="" type="checkbox"/>		

- ⇒ Click **Add Check Step**.

Go Back

Check Steps

Type	Step	Edit
Add Check Step		

Check Actions

Data Point	Action	Edit
Add Check Action		

- ⇒ For the **Type**, select **Data Value**.

Go Back

Check Steps

Type	Step	Edit
Constant Data Value Constant Check Function Custom Function	Value: 0 Format: 0	Update Cancel Delete

Check Actions

Data Point	Action	Edit
Add Check Action		

- ⇒ Specify the **SERIOUS** field on the **Adverse Events Log** form. Leave the **Folder** blank.

Check Steps						
Type	Step					
	Folder	Form	Field	Variable	Rec. Pos.	
Data Value	...	Adverse Event Log	SERIOUS	YES_NO		
<input type="checkbox"/> Apply To All Folders <input type="checkbox"/> Apply To All Fields						
Check Actions						
Data Point	Action			Edit		
Add Check Action						

- ⇒ For the type of the Data Value, select **Data Point**. Then click **Update**.

Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number	Logical Record Position	Data Value	Edit
Log	SERIOUS	YES_NO			None	Standard Value	<input checked="" type="radio"/> Update <input type="radio"/> Cancel <input type="radio"/> Delete
Fields							<input checked="" type="radio"/> Data Point

- ⇒ Click **Add Check Step**.

Check Steps			
Type	Step		
Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None		
Add Check Step			
Check Actions			
Data Point	Action		Edit
Add Check Action			

- ⇒ For the Type, select **Custom Function**. For the **Name**, select **IsModified**. For **Input**, select **From previous Step**. Then click **Update**.

Check Steps			
Type	Step		
Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None		
Custom Function	Name: IsModified	Input: From previous Step	<input checked="" type="radio"/> Update <input type="radio"/> Cancel <input type="radio"/> Delete
Check Actions			
Data Point	Action		Edit
Add Check Action			

⇒ Click **Add Check Step**.

Type	Step	Edit
Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	
Custom Function (From previous Step)	IsModified	

Add Check Step

⇒ For the **Type**, select **Data Value**. Then specify the **AESYMP** field on the **Adverse Events Log** form. Leave the **Folder** blank.

Type	Step
Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None
Custom Function (From previous Step)	IsModified

Folder	Form	Field	Variable	Rec. Pos.
	Adverse Event Log	AESYMP	AESYMP	

Add Check Action


⇒ For the type of **Data Value**, select **Data Point**. Then click **Update**.


Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number	Logical Record Position	Data Value	Action	Edit
Adverse Event Log	AESYMP	AESYMP			None	Standard Value	Update	

Data Point

⇒ Click **Add Check Step**.



Invalid Edit Check Entered


Check Steps			
	Type	Step	Edit
▲▼	Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	⌕
<div>  Add Check Step </div>			

Check Actions		
Data Point	Action	Edit
<div>  Add Check Action </div>		

⇒ Select **Custom Function** for the **Type** of check step. For **Name**, select **IsModified**, and for **Input**, select **From Previous Step**. Then click **Update**.


Invalid Edit Check Entered

Check Steps			
	Type	Step	Edit
▲▼	Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	⌕
<div> <div> Custom Function ▼ </div> <div> Name IsModified </div> <div> Input From previous Step </div> <div>  Update Cancel </div> <div>  Delete </div> </div>			

Check Actions		
Data Point	Action	Edit
<div>  Add Check Action </div>		

⇒ Follow the same pattern for the **STARTDT**, **ENDDATE**, and **GRADE** fields, alternating with one field, then one **IsModified** custom function. This is the result of adding all fields:

Invalid Edit Check Entered

Check Steps			
	Type	Step	Edit
▲▼	Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>STARTDT>DATE>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>ENDDT>DATE>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
▲▼	Data Point (Data Point)	...>Adverse Event Log>GRADE>GRADE>...>...>None	⌕
▲▼	Custom Function (From previous Step)	IsModified	⌕
<div>  Add Check Step </div>			

⇒ Click **Add Check Step**.

Invalid Edit Check Entered

Check Steps			
	Type	Step	Edit
▲▼	Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	✓
▲▼	Custom Function (From previous Step)	IsModified	✓
▲▼	Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	✓
▲▼	Custom Function (From previous Step)	IsModified	✓
▲▼	Data Point (Data Point)	...>Adverse Event Log>STARTDT>DATE>...>...>None	✓
▲▼	Custom Function (From previous Step)	IsModified	✓
▲▼	Data Point (Data Point)	...>Adverse Event Log>ENDDT>DATE>...>...>None	✓
▲▼	Custom Function (From previous Step)	IsModified	✓
▲▼	Data Point (Data Point)	...>Adverse Event Log>GRADE>GRADE>...>...>None	✓
▲▼	Custom Function (From previous Step)	IsModified	✓
+	Add Check Step		

⇒ For the type, select **Check Function**, select **Or**, then click **Update**.

Invalid Edit Check Entered

Check Steps			
Type	Step	Edit	
Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	✓	
Custom Function (From previous Step)	IsModified	✓	
Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	✓	
Custom Function (From previous Step)	IsModified	✓	
Data Point (Data Point)	...>Adverse Event Log>STARTDT>DATE>...>...>None	✓	
Custom Function (From previous Step)	IsModified	✓	
Data Point (Data Point)	...>Adverse Event Log>ENDDT>DATE>...>...>None	✓	
Custom Function (From previous Step)	IsModified	✓	
Data Point (Data Point)	...>Adverse Event Log>GRADE>GRADE>...>...>None	✓	
Custom Function (From previous Step)	IsModified	✓	
Check Function ▼	Or ▼	✓ Update	Cancel Delete

Check Actions

Data Point	Action	Edit
+	Add Check Action	

⇒ Repeat the last step, adding three additional Or check functions. The result is this:

If **SERIOUS** in Adverse Event Log IsModified Or **AESYMP** in Adverse Event Log IsModified Or **STARTDT** in Adverse Event Log IsModified Or **ENDDT** in Adverse Event Log IsModified Or **GRADE** in Adverse Event Log IsModified then... (No Actions Have Been Selected)

Check Steps			
	Type	Step	Edit
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>SERIOUS>YES_NO>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>AESYMP>AESYMP>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>STARTDT>DATE>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>ENDDT>DATE>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>GRADE>GRADE>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
Add Check Step			

⇒ Click **Add Check Action**.

▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Data Point (Data Point)	...>Adverse Event Log>GRADE>GRADE>...>...>None	✓
▲ ▼	Custom Function (From previous Step)	IsModified	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
▲ ▼	Check Function	Or	✓
Add Check Step			
Check Actions			
▲ ▼	Data Point	Action	Edit
Add Check Action			

⇒ For **Data Point**, select the **SERIOUS** field on the **Adverse Events Log** form. Leave the **Folder** blank.

Check Actions						
Data Point						
Folder	Form	Field	Variable	Rec. Pos.	Form Repeat Number	Folder Repeat Number
...	Adverse Event Log	SERIOUS	YES_NO			
<input type="checkbox"/> Apply To All Folders <input type="checkbox"/> Apply To All Fields						

⇒ For the **Action Type**, select **Custom Function**. For the custom function **Name**, choose **SAE Email**. Then click **Update**.

Action		Edit	
Action Type	Custom Function	SAE email	
		<input checked="" type="radio"/> Update Cancel	<input type="checkbox"/> Delete

Write the C# custom function code.

```

const string YES = "1";

//Get SERIOUS DataPoint passed in from Rave
ActionFunctionParams afp =
(ActionFunctionParams)ThisObject; DataPoint dpSerious =
afp.ActionDataPoint;

//Only continue if this is a serious AE if
(dpSerious.Data == YES)
{
    string subject, body, to;
    const string from = "rave@mdsol.com";
    Subject currentSubject = dpSerious.Record.Subject;

    //Use string.Format to create a subject
    //with the Subject Name and Site Number if
    (dpSerious.IsObjectChanged) //New
    {
        subject = String.Format(@"SAE Added! Site: {0},
                                Subject: {1}", currentSubject.StudySite.Site.Number,
                                dpSerious.Record.Subject.Name);
        body = BodyForNewAE(dpSerious);
    }
    else //Updated
    {
        subject = String.Format(@"SAE Updated! Site: {0},
                                Subject: {1}",
                                currentSubject.StudySite.Site.Number,
                                dpSerious.Record.Subject.Name);
        body = BodyForUpdatedAE(dpSerious);
    }

    //Chose recipient based on current environment
    switch (currentSubject.StudySite.Study.Environment.ToUpper())
    {
        case "DEV":
            to = "user_dev@mdsol.com"; break;
        case "PROD":
            to = "user_prod@mdsol.com"; break;
        default:
            to = "user_any@mdsol.com"; break;
    }
    Message.SendEmail(to, from, subject, body);
}
return null;
}

//Composes the body of the email message for NEW serious AEs.
//Lists all fields and their values of the current AE log line
string BodyForNewAE (DataPoint dpSerious)
{

```

```

    string body = "A new serious AE has been recorded by "
        + dpSerious.Interaction.TrueUser.Login
        + Environment.NewLine;

    foreach (DataPoint dp in dpSerious.Record.DataPoints)
    {
        if (dp.Field.OID == "SERIOUS")
            continue;
        body += string.Format("{0}: {1}",
                               dp.Field.PreText, dp.UserValue());
        body += Environment.NewLine;
    }
    return body;
}

//Composes the body of the email message for UPDATED serious AEs.
//Lists only updated fields and their values of the current AE log line
string BodyForUpdatedAE (DataPoint dpSerious)
{
    string body = "An existing serious AE has been updated by "
        + dpSerious.Interaction.TrueUser.Login
        + Environment.NewLine;
    body += "The following fields have been
    updated:"

    Environment.NewLine;
    foreach (DataPoint dp in
    dpSerious.Record.DataPoints)
    {
        if (dp.IsObjectChanged)
        {
            body += string.Format("{0}: {1}", dp.Field.PreText,
                                   dp.UserValue());
            body += Environment.NewLine;
        }
    }

    return body
}

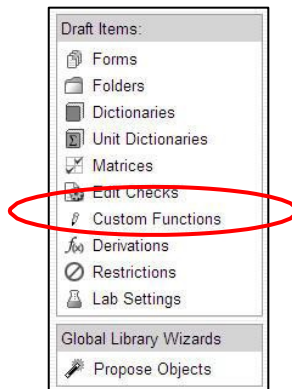
```

In order for a custom function containing multiple methods to work, it must be pasted into Rave as shown in the dashed-rectangle above.

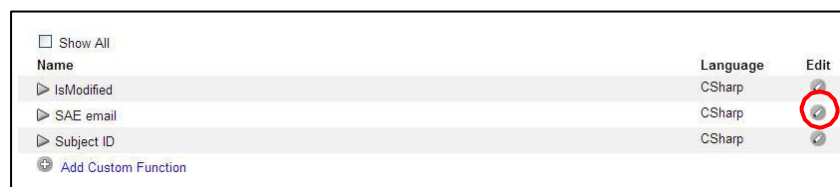
Note: Custom Function Development utility is not fully integrated with Rave, therefore some properties dependent on user interaction will not return expected results during debugging. Since an actual user interaction will not occur, `IsObjectChanged` would return false and `dpSerious.Interaction.TrueUser` would return null. To debug a code containing these properties, a developer should temporary replace them with hard coded values.

Paste this code into the Project Draft.

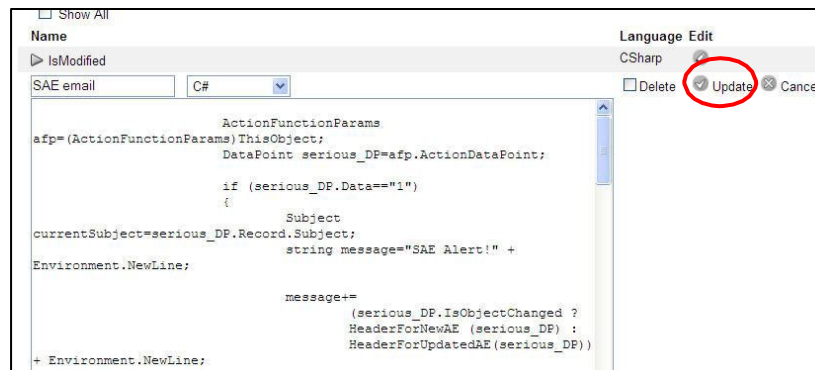
- ⇒ Copy the code written inside whatever editor is being used.
- ⇒ From the list of **Draft Items**, select **Custom Functions**.



- ⇒ Click the **Edit** pencil next to the SAE Email custom function.



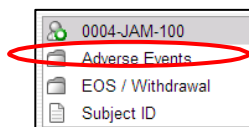
- ⇒ Paste the code into the source code window, and then click **Update**.



Apply the changes by overwriting existing latest CRF Version. For more information on Publishing, Pushing and Overwriting, refer to the Medidata Rave Architect Training Manual.

Use the subject created earlier (in "Derived Subject ID" example) to test whether the business rule for this custom function has been implemented correctly.

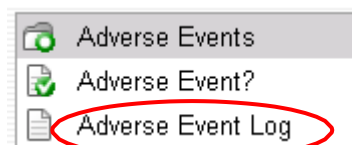
- ⇒ On the sidebar, click the **Adverse Events** folder.



- ⇒ Within **Adverse Events** folder, in the **Adverse Events** form, answer **Yes** to "Were there any adverse events?" Then click **Save**.

 A screenshot of a web form titled 'Page: Adverse Event? - Adverse Events'. It contains a question 'Were there any adverse events?' with radio buttons for 'Yes' and 'No'. The 'Yes' radio button is selected and circled in red. At the bottom right, there are 'Save' and 'Cancel' buttons, with the 'Save' button also circled in red.

- ⇒ On the sidebar, click on the **Adverse Event Log** that has just been added.



- ⇒ Enter values for **Symptom**, **Onset Date**, **End Date**, and **Grade**. Make sure to check **Is this a serious AE?** field. Then click **Save**.

 A screenshot of a web form titled 'Page: Adverse Event Log - Adverse Events'. It contains several input fields: 'Symptom' (with 'Headache' entered), 'Onset Date' (1 Mar 2007), 'End Date' (2 Mar 2007), 'Is this a serious AE?' (checked), and 'Grade' (Severe). At the bottom right, there are 'Save' and 'Cancel' buttons, with the 'Save' button circled in red.

- ⇒ If the edit check and custom function are developed correctly, an email similar to the following is sent:

From: rave@mdsol.com [mailto:rave@mdsol.com]

Sent: Sun 12/30/2007 4:25 PM

To: user_dev@mdsol.com

Subject: SAE Added! Site: 101, Subject: 001-ABC-02 Jan 1950

A new serious AE has been recorded by user_dev
Symptom: Headache

Onset Date: 1 MAR 2007

End Date: 2 MAR 2007 Grade: Severe

Form Cross-Check Custom Functions

- ❑ Introduction
- ❑ Example: AE Symptom Required

Introduction

It is common for edit checks to include fields from different forms. For cases involving a simple comparison between such fields, a standard edit check is sufficient and no custom function is needed. An example of such a case is a check that ensures the enrollment date on a screening form is greater than the birth date on a demography form.

One different type of cross-form edit check, however, always requires the use of a custom function. This type of check requires that if a user enters a particular value on one non-log form, that there be at least one entry of a particular value on a different, log form.

The example in this section involves two forms:

- ❑ An “Adverse Events Yes/No” *non-log* form with a single field asking whether or not there were any adverse events for the subject
- ❑ An “Adverse Event Log” *log* form with several log fields, among them a field for recording the symptom

The example requires that if the field on the Adverse Events Yes/No has a response of “yes”, at least one log line on Adverse Events Log form is submitted with a non-blank response in the symptom field.

The walkthrough and solution to the sample specification are important for two reasons:

- ❑ Checks that look for *at least one* instance of a response on a log form require a custom to look through each log line one by one.
- ❑ This one custom function must be called not from one, but from *two* different edit checks. This is because, as a consequence of the Rave edit check engine, one edit check alone will not run the custom function in the case that the Adverse Events log form has not yet been submitted.

Example: AE Symptom Required

This sample custom function is monitoring a relationship between two forms, which may not both exist at the same time. Therefore two edit checks are required.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	AE Symptom Required
Short Description:	If 'were there any adverse events' is marked as 'yes', there must be at least one AE logged with a symptom.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable): <i>AE Y/N is answered 'Yes', but no AE log symptoms have been completed.</i>	

Elements involved in the Custom Function:			
Folder OID(s):	AE		
Form OID:	AEYN		
Form Name:	Adverse Event?		
Field OID(s)	Pretext	Is Log?	Action
AEYN	Were there any adverse events?	<input type="checkbox"/>	Open Query
Folder OID(s):	AE		
Form OID:	AE		
Form Name:	Adverse Event Log		
Field OID(s)	Pretext	Is Log?	Action
AESYMP	Symptom	<input checked="" type="checkbox"/>	None

Business Rules:

If the AEYN field on the AEYN form is marked as 'yes', then there must be at least one *active* log line on the AE form with the SYMPTOM field entered.

If there is not, then open a query on the AEYN field.

To set up the two edit checks calling one custom function:

Look at the specification to gather information.

- ⇒ One field is the AEYN field on the AEYN form in the AE folder.
- ⇒ The other is the AESYMP field on the AE form, also in the AE folder.
- ⇒ If the AEYN = yes, then there must be at least one log line with AESYMP completed.

Add a new "blank" custom function to the project draft as described in previous sections. It should be called **"AE Symptom Required"**.

Create two edit checks:

AE Symptom Required 1

If AESYMP in Adverse Event Log in Adverse Events IsModified then... execute the "AE Symptom Required" custom function			
Check Steps			
	Type	Step	Edit
	Data Point (Data Point)	Adverse Events>Adverse Event Log>AESYMP>AESYMP>...>...>None	
	Custom Function (From previous Step)	IsModified	
	Add Check Step		
Check Actions			
	Data Point	Action	Edit
	Adverse Events>Adverse Event?>AEYN>AEYN>...>...>None	Custom Function:AE Symptom Required	
	Add Check Action		

AE Symptom Required 2

If AEYN in Adverse Event? in Adverse Events IsModified then... execute the "AE Symptom Required" custom function			
Check Steps			
	Type	Step	Edit
	Data Point (Data Point)	Adverse Events>Adverse Event?>AEYN>AEYN>...>...>None	
	Custom Function (From previous Step)	IsModified	
	Add Check Step		
Check Actions			
	Data Point	Action	Edit
	Adverse Events>Adverse Event?>AEYN>AEYN>...>...>None	Custom Function:AE Symptom Required	
	Add Check Action		

Note: There is no need to specify record position here. That is because this edit check will not contain both non-log and log forms - only non-log. However, it's recommended to always specify record position as 0 for standard fields.

⇒ Write the C# custom function code.

```
//To decide whether any AEs have been logged, look for at least one
//DataPoint matching this OID path that has complete data.
const string REQUIRED_AE_FIELD_OID = "AESYMP";
const string AE_FORM_OID = "AE"; const string AE_FOLDER_OID = "AE";
const string QUERY_TEXT = @"AE Y/N is answered 'Yes', but no AE log
                           symptoms have been completed.";

const int MARKING_GROUP_ID = 1;
const bool ANSWER_ON_CHANGE = false;
const bool CLOSE_ON_CHANGE = false;

//Get the AEYN DataPoint passed in from Rave
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
DataPoint dpAEYN = afp.ActionDataPoint;

bool openQuery = false;

if (dpAEYN.Data == YES)
{
    openQuery = true;

    Subject current_Subject = dpAEYN.Record.Subject;
    DataPoints dpsAESYMP = CustomFunction.FetchAllDataPointsForOIDPath(
        REQUIRED_AE_FIELD_OID, AE_FORM_OID,
        AE_FOLDER_OID, current_Subject);

    //Look for one completed AE symptom and stop looking when found
    for (int i=0; i<dpsAESYMP.Count; i++)
    {
        //Inactivated log lines shouldn't count as valid AEs
        if (dpsAESYMP[i].Active && dpsAESYMP[i].EntryStatus
            == EntryStatusEnum.EnteredComplete)
        {
            openQuery = false;
            break;
        }
    }
}

//Open or close query as necessary
CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID,
    ANSWER_ON_CHANGE, CLOSE_ON_CHANGE,
    dpAEYN, openQuery, afp.CheckID, afp.CheckHash);

return null;
```

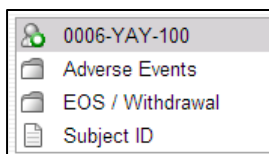
Paste the code into the Project Draft.

- ⇒ On the Draft Items sidebar, click Custom Functions.
- ⇒ Click the **Edit Pencil** for the **AE Symptom Required** custom function.
- ⇒ Paste the custom function into the source code text box. Then click **Update**.

Apply the changes by overwriting existing latest CRF Version. For more information on Publishing, Pushing and Overwriting, refer to the Medidata Rave Architect Training Manual.

Test whether the business rule for this custom function has been implemented correctly.

- ⇒ Navigate to the Rave home page.
- ⇒ Select appropriate Study and Site. Refer to the EDC Training Manual for more information on navigating to Studies and Sites.
- ⇒ Select the subject created earlier or a create a new subject.
- ⇒ Click on the **Adverse Events** folder on the sidebar.



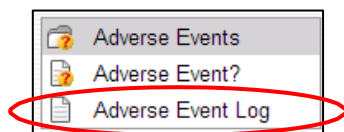
- ⇒ On the **Adverse Events?** form, respond to "Were there any adverse events?" with "yes". Then click **Save**.

A screenshot of the 'Adverse Events?' form. At the top, it says 'Subject: 0006-YAY-100' and 'Page: Adverse Event? - Adverse Events'. The main question is 'Were there any adverse events?'. Below it, there are two radio buttons: 'Yes' (which is selected and circled in red) and 'No'. At the bottom right, there are 'Save' and 'Cancel' buttons. The 'Save' button is also circled in red. There are also links for 'Printable Version', 'View PDF', and 'Icon Key'.

- ⇒ A query is opened on the AEYN field.

A screenshot of the 'Adverse Events?' form after submission. At the top, it says 'Your form has been successfully submitted.' Below that, it says 'Subject: 0008-YAY-100' and 'Page: Adverse Event? - Adverse Events'. The main question is 'Were there any adverse events?'. Below it, there is a red error message: '? AE Y/N is answered "Yes", but no AE log symptoms have been completed. Opened To Site from System (01 Jan 2008)'. To the right of the error message, there is a 'Data Entry Error' dropdown menu and two radio buttons: 'Yes' (selected) and 'No'. At the bottom right, there are 'Save' and 'Cancel' buttons. There are also links for 'Printable Version', 'View PDF', and 'Icon Key'.

- ⇒ Notice also that a new form, the **Adverse Event Log**, has been added to the **Adverse Events** folder by an unrelated standard check. Click the link to this new form on the sidebar.



- ⇒ Fill out the first line of the Adverse Event Log, making sure to enter something in the **Symptom** field. Then click **Save**.

Subject: 0008-YAY-100
Page: Adverse Event Log - Adverse Events

Currently viewing line 1 of 1.
Click here to return to "Complete View".

Symptom: headache

Onset Date: 01 Dec 2007

End Date: 01 Dec 2007

Is this a serious AE?: No

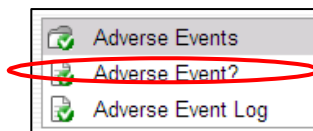
Grade: Mild

Printable Version View PDF Icon Key

CRF Version 302 - Page Generated: 01 Jan 2008 03:28:34 Greenwich Standard Time

Save Cancel

- ⇒ Click the link to the Adverse Event? form on the sidebar. Notice that the status icon for this form has switched from query to complete.



- ⇒ The query on the **AEYN** field has been closed because there is now at least one complete symptom field on the AE log form.

Subject: 0008-YAY-100
Page: Adverse Event? - Adverse Events

Were there any adverse events? Yes

Printable Version View PDF Icon Key

CRF Version 302 - Page Generated: 01 Jan 2008 03:37:30 Greenwich Standard Time

Save Cancel

Timespan Custom Functions

- ❑ Introduction
- ❑ Example: Diagnosis Date vs. Enrollment Date

Introduction

One very common type of edit check is one that compares two date fields.

Imagine that a study has a non-log enrollment form with an enrollment date field, as well as a medical history *log* form with fields for specifying both a diagnosis and the date the diagnosis was made. For this setup, it makes sense to enforce a rule that any diagnosis entered on the log form must have been made before the enrollment date. This way, the study would only consider the impact of the study drug on pre-existing conditions.

Suppose it's also necessary to ensure that diagnosis date falls within a range of 180 days prior to the enrollment date.

$0 \text{ days} < \text{Enrollment Date} - \text{Diagnosis Date} \leq 180 \text{ days}.$

The example in this section shows how to create a custom function to enforce this kind of requirement, utilizing dates and time span.

Example: Diagnosis Date Check

This sample custom function is implemented as an edit check action.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	Diagnosis Date Check
Short Description:	All diagnosis dates must come before the enrollment date, but not more than 180 days prior.

IF an EDIT CHECK, please provide General Edit Check Requirements:			
Response Required?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	<input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	<input type="checkbox"/> N/A
Marking Group(s)	1		
Please specify action message (if applicable): <i>Diagnosis date must be before enrollment date, but not more than 180 days prior.</i>			

Elements involved in the Custom Function:			
Folder OID(s):	SCRN		
Form OID:	ENRL		
Form Name:	Enrollment		
Field OID(s)	Pretext	Is Log?	Action
ENRLDT	Enrollment Date	<input type="checkbox"/>	None
Folder OID(s):	SCRN		
Form OID:	MEDHX		
Form Name:	Medical History		
Field OID(s)	Pretext	Is Log?	Action
MEDHX_DT	Diagnosis	<input checked="" type="checkbox"/>	Open Query

Business Rules:
For any active log line, MEDHX_DT must be less then ENRL_DT but not more than 180 days prior. Otherwise, a query should be opened at MEDHX_DT on that log line.

To set up edit check and custom function:

Look at the specification to gather information.

- ⇒ One field is the ENRLDT field on ENRL *non-log* form in the SCRn folder.
- ⇒ The other field is the MEDHX_DT field on the MEDHX *log* form, also in the SCRn folder.
- ⇒ All instances of the MEDHX_DT field must be at least one day and at most 180 days prior to the ENRLDT field.

Add a new "blank" custom function to the project draft as described in previous sections. It should be called **"Diagnosis Date Check"**.

Create edit check **"Diagnosis Date Check"**:

[Go Back](#)

If ENRLDT in Enrollment in Screening with record position 0 IsModified Or MEDHX_DT in Medical History in Screening IsModified then... execute the "Diagnosis Date Check" custom function

Check Steps

	Type	Step	Edit
▲▼	Data Point (Data Point)	Screening>Enrollment>ENRLDT>ENRLDT>0>...>...>None	
▲▼	Custom Function (From previous Step)	IsModified	
▲▼	Data Point (Data Point)	Screening>Medical History>MEDHX_DT>MEDHX_DT>...>...>None	
▲▼	Custom Function (From previous Step)	IsModified	
▲▼	Check Function	Or	
+ Add Check Step			

Check Actions

Data Point	Action	Edit
Screening>Medical History>MEDHX_DT>MEDHX_DT>...>...>...	Custom Function:Diagnosis Date Check	
+ Add Check Action		

Write the C# custom function code.

```
//This is the OID path for the enrollment date
const string FIELD_OID = "ENRLDT";
const string FORM_OID = "ENRL";
const string FOLDER_OID = "SCRN";
const string QUERY_TEXT = @"Diagnosis date must be before enrollment
                             date, but not more than 180 days
                             prior.";
const int MARKING_GROUP_ID = 1;
const bool ANSWER_ON_CHANGE = false;
const bool CLOSE_ON_CHANGE = false;

//Get the MEDHX_DT field passed in from Rave
ActionFunctionParams afp = (ActionFunctionParams)ThisObject; DataPoint
dpDiagnosis = afp.ActionDataPoint;

//Initially, we don't plan to open a query
bool openQuery = false;

//Only obtain the enrollment date if the diagnosis passed in is a valid date
if (dpDiagnosis.StandardValue() is DateTime)
{
    //Get a reference to the current subject
    Subject current_Subject = dpDiagnosis.Record.Subject;

    //Fetch enrollment date
    DataPoints dpsENRLDT =
        CustomFunction.FetchAllDataPointsForOIDPath(
            FIELD_OID, FORM_OID, FOLDER_OID, current_Subject);

    //Don't continue unless the enrollment is a valid date
    if (dpsENRLDT.Count > 0 && dpsENRLDT[0].StandardValue() is DateTime)
    {
        //Perform casts directly from StandardValue()
        //instead of using Convert.ToDateTime

        DateTime enrollment_DT = (DateTime)dpsENRLDT[0].StandardValue();
        DateTime diagnosis_DT = (DateTime)dpDiagnosis.StandardValue();

        //Set openQuery to true if the dates are outside
        //the appropriate timespan
        openQuery = !(diagnosis_DT < enrollment_DT &&
            diagnosis_DT >= enrollment_DT.Subtract(
                new TimeSpan(180, 0, 0, 0, 0)));
    }
}

//Open or close the query as necessary
CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID,
    ANSWER_ON_CHANGE, CLOSE_ON_CHANGE,
    dpDiagnosis, openQuery,
    afp.CheckID, afp.CheckHash);

return null;
```

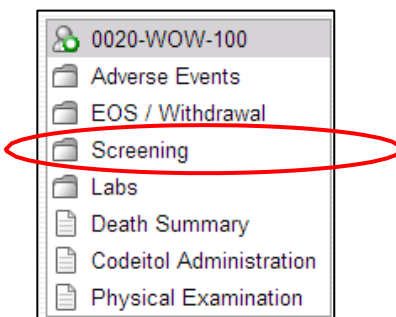
Note: This edit check (EC) and custom function (CF) example are for training purpose only. This CF relies on the fact that its EC will execute for EVERY log line. This is very inefficient, especially because `FetchAllDataPointsForOIDPath()` is used, which will be executed multiple times. To ensure that EC executes only once, it must be split into two checks (to avoid mixing log and standard fields in EC steps), standard field should be used in the action, and the CF must be updated accordingly.

Paste this code into the Project Draft.

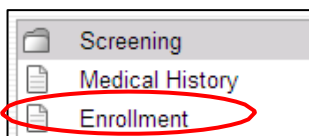
Apply the changes by overwriting existing latest CRF Version. For more information on Publishing, Pushing and Overwriting, refer to the Medidata Rave Architect Training Manual.

Test whether the business rule for this custom function has been implemented correctly.

- ⇒ Navigate to the Rave home page.
- ⇒ Select appropriate Study and Site. Refer to the EDC Training Manual for more information on navigating to Studies and Sites.
- ⇒ Select the subject created earlier or a create a new subject.
- ⇒ Click on the **Screening** folder on the sidebar.



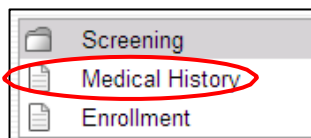
- ⇒ Click on the link to the **Enrollment** form.



- ⇒ Enter an **Enrollment Date** of Jan 1 2000, then click **Save**.

A screenshot of the Medidata Rave Enrollment form. The form has a header with 'Subject: 0020-WOW-100' and 'Page: Enrollment - Screening'. Below the header, there are two date fields: 'Enrollment Date' and 'Informed Consent Date'. The 'Enrollment Date' field is filled with '01 Jan 2000' and is highlighted with a red oval. The 'Informed Consent Date' field is empty. At the bottom right of the form, there are two buttons: 'Save' (highlighted with a red oval) and 'Cancel'. The footer of the form contains links for 'Printable Version', 'View PDF', and 'Icon Key', and a footer note: 'CRF Version 739 - Page Generated: 22 Feb 2008 11:43:41 Greenwich Standard Time'.

⇒ Click the link to the **Medical History** form.



⇒ Enter any value for **Diagnosis or Procedure**. Then enter a **Date** of Jan 1999. Then click **Save**.

Subject: 0020-WOW-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory	01 Jan 1999	

Add a new Log line Inactivate

Printable Version View PDF Icon Key
CRF Version 739 - Page Generated: 22 Feb 2008 11:47:05 Greenwich Standard Time

Save Cancel

⇒ A query has opened on the date field:

Subject: 0020-WOW-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory	01 Jan 1999	

Approximate Month and Year of diagnosis of Procedure, if available
? Diagnosis date must be before enrollment date, but not more than 180 days prior.
Opened To Site from System (22 Feb 2008)

Add a new Log line Inactivate

Printable Version View PDF Icon Key
CRF Version 739 - Page Generated: 22 Feb 2008 13:49:24 Greenwich Standard Time

Save Cancel

⇒ Change the **Date** to **Dec 1999**, then click **Save**. The query closes because the diagnosis is now within 180 days of the enrollment.

Subject: 0020-WOW-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory	01 DEC 1999	

Add a new Log line Inactivate

Printable Version View PDF Icon Key
CRF Version 739 - Page Generated: 22 Feb 2008 13:51:31 Greenwich Standard Time

Save Cancel

Duplicate Log Line Functions

- ❑ Introduction
- ❑ Example: Duplicate Diagnosis Values

Introduction

It is often necessary to make sure that for a particular log field, all values entered are unique, so that no log line has the same value for that field as any other log line.

This must be done with a custom function.

Example: Duplicate Diagnosis Values

This example custom function will be set up to run as a check action.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	Duplicate diagnosis log lines
Short Description:	No two active diagnosis log lines can have identical values for the MEDHX_DIAG field.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable): <i>Duplicate diagnosis record. Please correct.</i>	

Elements involved in the Custom Function:			
Folder OID(s):	SCRN		
Form OID:	MEDHX		
Form Name:	Medical History		
Field OID(s)	Pretext	Is Log?	Action
MEDHX_DT	Diagnosis or Procedure	<input checked="" type="checkbox"/>	Open Query

Business Rules:
Two log lines should be considered identical if they have the same values for the MEDHX_DIAG field (not case sensitive).
Inactive log lines should be ignored.
When 2 or more lines are identical, open a query on all identical log lines but the first log line (i.e., smallest recordposition).
The query should open on the MEDHX_DIAG field.

To set up the edit check and custom function:

Look at the specification to gather information.

- ⇒ The only field involve here is the MEDHX_DIAG field on the MEDHX form in the Screening folder.
- ⇒ There can be no duplicate values for this log field

Add a new "blank" custom function to the project draft as described in previous sections. It should be called **"Duplicate Diagnosis"**.

Create edit check **"Duplicate Diagnosis"**:

If MEDHX_DIAG in Medical History in Screening IsPresent then... execute the "Duplicate Diagnosis" custom function

Check Steps

	Type	Step	Edit
⬇	Data Point (Data Point)	Screening>Medical History>MEDHX_DIAG>MEDHX_DIAG>...>...>...>None	✎
⬇	Check Function	IsPresent	✎
+ Add Check Step			

Check Actions

Data Point	Action	Edit
Screening>Medical History>MEDHX_INDIC>INDIC>0>...>...	Custom Function:Duplicate Diagnosis	✎
+ Add Check Action		

Note: Even though standard field in the check action is not required for the custom function logic, it's presence in the edit check is necessary to ensure that edit check will be executed whenever a log line is inactivated, as there always will be a reference to at least one standard field, which cannot be inactivated.

Write the custom function code.

```
//Initialize parameters
const string QUERY_TEXT = "Duplicate diagnosis. Please, correct.";
const int MARKING_GROUP_ID = 1;
const bool ANSWER_ON_CHANGE = false; const bool CLOSE_ON_CHANGE = false;
const string fieldOID = "MEDHX_DIAG";

//Get the INDIC field passed in from Rave
ActionFunctionParams afp = ActionFunctionParams(ThisObject);
DataPoint dpIndic = afp.ActionDataPoint;

//Get a reference to all Records on the Med History page
Records allRecords = dpIndic.Record.DataPage.Records;

//Order the records by their record position
allRecords = GetSortedRecords(allRecords);

bool openQuery = false;
DataPoint i_dp = null; //for outer loop
DataPoint j_dp = null; //for inner loop

//i>1 because allRecords also contains master record
for (int i = allRecords.Count-1; i>1; i--)
{
    if (allRecords[i].Active) //Skip inactive records
    {
        i_dp = allRecords[i].DataPoints.FindByFieldOID(fieldOID);

        for (int j = i-1; j>0; j--)
        {
            if (allRecords[j].Active)
            {
                j_dp =
                    allRecords[j].DataPoints.FindByFieldOID(fieldOID);
                if (string.Compare(i_dp.Data, j_dp.Data, true) == 0)
                {
                    openQuery = true;
                    break;
                    //Open query only on this current duplicate.
                    //For other duplicates, queries will be opened
                    //in following iterations.
                }
            }
        }

        //Open or close query as necessary
        CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID,
                                           ANSWER_ON_CHANGE, CLOSE_ON_CHANGE, i_dp,
                                           openQuery, afp.CheckID, afp.CheckHash);

        openQuery = false;
    }
}
return null;
}
```

```
//Only works when unSortedRecords contains ALL records of SAME datapage
Records GetSortedRecords (Records unSortedRecords)
{
    //Sort Records in the collection by record position.
    Record[] tmpRecords = new Record[unSortedRecords.Count];
    for (int i=0; i<unSortedRecords.Count; i++)
    {
        Record record = unSortedRecords[i];
        tmpRecords[record.RecordPosition] = record;
    }

    //Restore the original collection data type
    Records sortedRecords = new Records();
    for (int i=0; i<tmpRecords.Length; i++)
    {
        sortedRecords.Add(tmpRecords[i]);
    }

    return sortedRecords;
}
```

In order for a custom function containing multiple methods to work, it must be pasted into Rave as shown in the dashed-rectangle above.

Paste this code into the Project Draft as described earlier.

- ⇒ On the Draft Items sidebar, click Custom Functions.
- ⇒ Click the **Edit Pencil** for the **Duplicate Diagnosis** custom function.
- ⇒ Paste the custom function into the source code text box. Then click **Update**.

Apply the changes by overwriting existing latest CRF Version. For more information on Publishing, Pushing and Overwriting, refer to the Medidata Rave Architect Training Manual.

Test whether the business rule for this custom function has been implemented correctly.

- ⇒ Navigate to the Rave home page.
- ⇒ Select appropriate Study and Site. Refer to the EDC Training Manual for more information on navigating to Studies and Sites.
- ⇒ Select the subject created earlier or a create a new subject.
- ⇒ Navigate to the **Medical History** form inside the Screening Folder.
- ⇒ Enter a **Diagnosis of Respiratory**, then click **Save**.

Subject: 0012-MMM-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory		

Add a new Log line Inactivate

Printable Version View PDF Icon Key

CRF Version 306 - Page Generated: 08 Jan 2008 20:43:53 Greenwich Standard Time

Save Cancel

- ⇒ Enter a second **Diagnosis of Respiratory**, then click **Save**.

Subject: 0012-MMM-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory		
2	Respiratory		

Add a new Log line Inactivate

Printable Version View PDF Icon Key

CRF Version 306 - Page Generated: 08 Jan 2008 20:46:32 Greenwich Standard Time

Save Cancel

- ⇒ A query opens on the second log line, as expected.

Subject: 0012-MMM-100
Page: Medical History - Screening

Does the subject have a known history of an abnormality, disease or surgery?

#	Diagnosis or Procedure	Date	Status
1	Respiratory		
2	Data Entry Error Respiratory		

Diagnosis or Procedure
? Duplicate diagnosis
Opened To Site from System (08 Jan 2008)

Add a new Log line Inactivate

Printable Version View PDF Icon Key

CRF Version 306 - Page Generated: 08 Jan 2008 20:49:37 Greenwich Standard Time

Save Cancel

Performing Calculations with Custom Functions

- ❑ Introduction
- ❑ Example: Body Surface Area Calculation

Introduction

A custom function is necessary when performing a calculation that uses a math function not found as a standard derivation step or edit check step. Examples of these functions include raising a number to a particular power and taking a logarithm.

This section shows how to write a custom function that calculates body surface area (BSA). A custom function is needed here because the formula involves taking a square root.

The sample presented uses a custom function configured as an edit check action.

Example: Body Surface Area Calculation

Consider the following specification for a body surface area calculation.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	BSA
Short Description:	Populate the BSA field using the values of HEIGHT and WEIGHT with the formula provided in the Business Rules section below.

Elements involved in the Custom Function:			
Folder OID(s):			
Form OID:	EXAM		
Form Name:	Physical Examination		
Field OID(s)	Pretext	Is Log?	Action
HEIGHT	Height	<input type="checkbox"/>	None
WEIGHT	Weight	<input type="checkbox"/>	None
BSA	Body Surface Area	<input type="checkbox"/>	Set Value Set Visible

Business Rules:
<p>Populate the BSA field based on the values of the HEIGHT and WEIGHT fields using this formula:</p> $BSA (m^2) = ((HEIGHT (cm) * WEIGHT (kg)) / 3600) ^ {1/2}$ <p>This derived BSA field should be recalculated whenever the HEIGHT or WEIGHT values are changed.</p> <p>If either HEIGHT or WEIGHT is left blank, the BSA should not be calculated.</p> <p>Make Body Surface Area field visible.</p>

To set up the edit check and custom function:

- ⇒ Look at the specification to gather information.
- ⇒ All the fields mentioned are on the EXAM form.
- ⇒ The BSA field should be populated from the HEIGHT and WEIGHT fields using the formula
- ⇒ $BSA (m^2) = ((HEIGHT (cm) * WEIGHT (kg)) / 3600) ^ {1/2}$.
- ⇒ Even though the BSA field will be derived, an edit check will be used rather than a derivation. BSA will be populated through custom function code placed in a check action.

Add a new "blank" custom function to the project draft as described in previous sections. It should be called **"Body Surface Area"**.

Create edit check **"Body Surface Area"**:

if HEIGHT in Physical Examination with record position 0 IsModified Or WEIGHT in Physical Examination with record position 0 IsModified then... execute the "Body Surface Area" custom function			
Check Steps			
	Type	Step	Edit
⊕	Data Point (Data Point)	...>Physical Examination>HEIGHT>HEIGHT>0>...>...>None	⊙
⊕	Custom Function (From previous Step)	IsModified	⊙
⊕	Data Point (Data Point)	...>Physical Examination>WEIGHT>WEIGHT>0>...>...>None	⊙
⊕	Custom Function (From previous Step)	IsModified	⊙
⊕	Check Function	Or	⊙
⊕	Add Check Step		
Check Actions			
Data Point	Action	Edit	
...>Physical Examination>HEIGHT>HEIGHT>...>...>None	Custom Function:Body Surface Area	⊙	
⊕	Add Check Action		

Note: Form and field OIDs should be specified. Folder OID should be left blank because EXAM is a subject-level form. Refer to the Wildcarding section of Appendix B for more information.

- ⇒ Write the custom function code. The following source code should be used for this example:

```
const string WEIGHT_FIELD_OID = "WEIGHT";
const string BSA_FIELD_OID = "BSA";

ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
DataPoint dpHeight = afp.ActionDataPoint;

//Obtain the WEIGHT and BSA DataPoints
DataPoint dpWeight =
dpHeight.Record.DataPoints.FindByFieldOID(WEIGHT_FIELD_OID);
DataPoint dpBSA = dpHeight.Record.DataPoints.FindByFieldOID(BSA_FIELD_OID);
dpBSA.IsVisible = true;

double height = 0, weight = 0;
double bsa = double.NaN;
try
{
    height = Convert.ToDouble(dpHeight.StandardValue());
    weight = Convert.ToDouble(dpWeight.StandardValue());
}
catch
{
    //Nothing
}
//Calculate result only if both arguments are valid.
if (height > 0 && weight > 0)
{
    bsa = Math.Round(Math.Sqrt((height * weight) / 3600), 2);
}

dpBSA.UnFreeze(); //Unfreeze the BSA field

//If bsa calculated successfully, use the calculated value,
//otherwise clear the BSA field
string strBSA = (Double.IsNaN(bsa) ? string.Empty : bsa.ToString());
dpBSA.Enter(strBSA, null, 0);

dpBSA Freeze(); //Freeze the BSA field to prevent user data entry.
return null;
```

- ⇒ This ActionFunctionParams object contains the HEIGHT DataPoint passed in from Rave, and is accessed through the ActionDataPoint property

Note: To obtain a Data Value passed to a check action-referenced custom function, cast ThisObject to an ActionFunctionParams object and access its ActionDataPoint.

- ⇒ The body of the code obtains the WEIGHT DataPoint from the parent Record of the HEIGHT DataPoint passed in from Rave.
- ⇒ The entered value of these DataPoint objects is accessed through the Data property, which returns a string.
- ⇒ If the string values are successfully converted to double values, the BSA is calculated, and rounded to two decimal places. Then the enter method is used to put this calculated value inside the dpBSA DataPoint.
- ⇒ If the conversion is not successful, the code inputs blank data ("", or string.empty) to dpBSA.
- ⇒ The code returns an arbitrary object.

Test whether the business rule for this custom function has been implemented correctly.

- ⇒ Navigate to the home page.
- ⇒ Select appropriate Study and Site. Refer to the EDC Training Manual for more information on navigating to Studies and Sites.
- ⇒ Select the subject created earlier or a create a new subject.
- ⇒ Navigate to the **Physical Examination** form. This form is not inside a folder
- ⇒ Provide values for **Height** and **Weight**, then click **Save**.

Subject: New Subject
Page: Physical Examination

Height	182	cm		
Weight	80	kg		

Printable Version View PDF Icon Key
CRF Version 95 - Page Generated: 06 Dec 2007 16:02:18 Greenwich Standard Time

Save Cancel

- ⇒ The body surface area field has been made visible and populated. Click on the **Edit** pencil of the **Weight** field.

Subject: New Subject
Page: Physical Examination

Height	182	cm		
Weight	80	kg		
Body Surface Area	2.01	m^2		

Printable Version View PDF Icon Key
CRF Version 97 - Page Generated: 06 Dec 2007 16:11:45 Greenwich Standard Time

Save Cancel

- ⇒ Clear the **Weight** field and click **Save**, to test what happens if a value that is required for your calculation is not available.

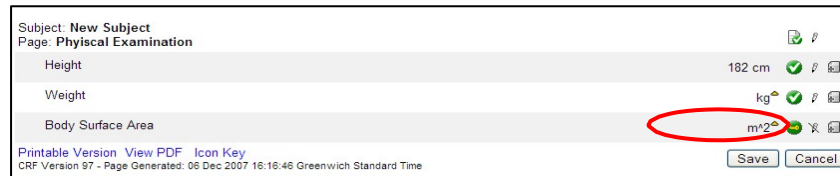
Subject: New Subject
Page: Physical Examination

Height	182	cm		
Weight	Data Entry Error	kg		
Body Surface Area	2.01	m^2		

Printable Version View PDF Icon Key
CRF Version 97 - Page Generated: 06 Dec 2007 16:15:15 Greenwich Standard Time

Save Cancel

- ⇒ The custom function has cleared out the **Body Surface Area** field as expected.



The screenshot displays the 'Physical Examination' page for a 'New Subject'. It lists three fields: Height (182 cm), Weight (kg), and Body Surface Area (m²). The 'Body Surface Area' field is circled in red, indicating it has been cleared. Each field has a green checkmark icon to its right. At the bottom, there are links for 'Printable Version', 'View PDF', and 'Icon Key', along with a footer note: 'CRF Version 97 - Page Generated: 06 Dec 2007 16:16:46 Greenwich Standard Time'. 'Save' and 'Cancel' buttons are also present.

Subject: New Subject	
Page: Physical Examination	
Height	182 cm
Weight	kg
Body Surface Area	m²

Printable Version View PDF Icon Key

CRF Version 97 - Page Generated: 06 Dec 2007 16:16:46 Greenwich Standard Time

Save Cancel

Using the Dynamic Searchlist eCRF Control

- ❑ Introduction
- ❑ Example: Device Name Lookup

Introduction

One standard control eCRF control available in Rave's Architect Form Designer is the Searchlist control. Like a simple drop-down list control, the Searchlist is connected to a dictionary and lets the user select one dictionary entry from the list. The difference is that the SearchList control gives the user the option of typing in the name of the item they want to select, which narrows down the list of available items with each keystroke the user makes, on-the-fly. This speeds up data entry, and is particularly helpful when selecting from longer lists.

Rave 5.6.3 introduces a new control that builds on this functionality, the **Dynamic SearchList**. This special type of list control is populated with items provided at run-time by a custom function which supplies a specific set of list items based on a previously entered data from one or more other fields.

Also new in Rave 5.6.3 is a third class of custom functions, written in SQL rather than C# or VB, created expressly for populating Dynamic SearchLists.

The example in this section shows how to offer the user a list of medical devices matching the device type previously selected.

Example: Device Lookup

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	Device name Lookup
Short Description:	Populate the DEVICE dynamic searchlist field according to the user's selection in the DEVICETYPE field.

Elements involved in the Custom Function:			
Folder OID(s):			
Form OID:	SURGERY		
Form Name:	Surgery		
Field OID(s)	Pretext	Is Log?	Action
DEVICETYPE	Device Type	<input type="checkbox"/>	None
DEVICE	Device Name	<input type="checkbox"/>	Other Populated this Dynamic Searchlist based on the user's entry in the DEVICETYPE field and the table presented in the Business Rules below.

Business Rules:		
The following table should be used to associate device names with device types. The entries in the TypeCode column match the entries in the Device Type data dictionary , already installed on the draft and used by the DEVICETYPE field.		
TypeCode	DeviceCode	DeviceName
C	CATH STD	Standard Occlusion Balloon Catheter
C	CATH STER	Sterling Balloon Dilation Catheter
C	CATH PERIPH	Peripheral Cutting Balloon Device
C	CATH ANGIO	Imager II Angiographic Catheter
S	STENT DYN	Dynamic (Y) Stent
S	STENT POLY	PolyFlex Airway Stent
S	STENT TRACH	Ultraflex Tracheobronchial Stent System
L	COIL GDC	GDC Detachable Coil
L	COIL MATRIX	Matrix2 Detachable Coil

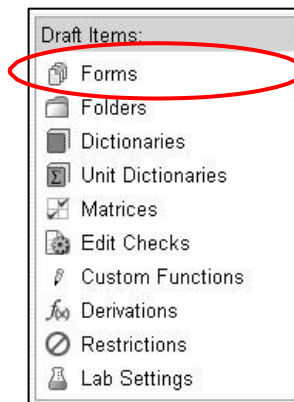
To set up the edit check and custom function

Look at the specification to gather information.

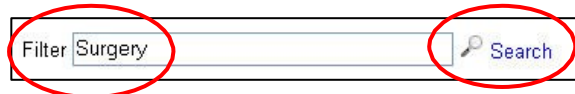
- ⇒ There are two fields involved: DEVICETYPE and DEVICE. Both are on the SURGERY form.
- ⇒ The DEVICE list needs to be filled based on what the user entered in the DEVICETYPE field.

Ensure that the dynamic list field is a Dynamic SearchList control..

- ⇒ Go into the Architect Module to get a closer look at the form.
- ⇒ From the **Draft Items** on the left side of the page, click **Forms**.



- ⇒ Search for the **Surgery** form.



- ⇒ Click **Fields**.

Filter <input type="text" value="Surgery"/> <input type="button" value="Search"/>												
Order	Form Name	OID	Help Text	#Fields	Active	Other Visit	Log Direction	Save Confirm	Redirect	Signature Required	Edit	Fields
	Surgery	SURGERY		3	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input checked="" type="checkbox"/>	No Link	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
	Add Form											

- ⇒ Click on the **Edit Pencil** for the DEVICE field.

Surgery					Preview
Num	Name	Label	Format	Active	
	Surgery date	Surgery Date	dd MMM yyyy	✓	
	Device Type	Device Type	\$2	✓	
	Device	Device	\$100	✓	

- ⇒ On the right side of the screen, change the **Control Type** to **Dynamic SearchList**.

Surgery

Control Type: Text

Accept files with extensions: CheckBox

Lab Analyte: DateTime

Default Value: DropDownList

SAS Label: Dynamic SearchList

SAS Format: File Upload

LongText

RadioButton

RadioButton (Vertical)

SearchList

Signature

Text

- ⇒ **Save** the field.

Save Cancel Go Back Delete

Add a new “blank” SQL custom function to the project draft as described in previous sections. It should be called Lookup Device Type.

Create edit check “**Lookup Device Type**”:

Lookup Device Type Update Cancel Delete

- ⇒ Add the following check steps as described in previous sections, using **IsPresent** instead of IsModified. Note, that both, DEVICETYPE and DEVICE are required to be in the edit check steps.

Check Steps				
	Type	Step	Edit	
+	Data Value	...>Surgery>Device Type>DEVICETYPE>...>...>None	✓	
+	Check Function	IsPresent	✓	
+	Data Value	...>Surgery>Device>DEVICE>...>...>None	✓	
+	Check Function	IsPresent	✓	
+	Check Function	And	✓	
+	Add Check Step			

⇒ Add a new check action, and for the **Data Point**, select the **DEVICETYPE** field.

Check Actions

Data Point

Folder: ... Form: Surgery Field: Device Variable: DEVICETYPE

☐ Apply To All Folders ☐ Apply To All Fields

Rec. Pos. Form Repeat Number Folder Repeat Number

⇒ For the Action Type, select SetDynamicSearchList, and for the Custom Function, select Lookup Device Type. Then click Update.

Action

Action Type: Set DynamicSearchList Custom Function: Lookup Device Type

⇒ The completed edit check looks like this:

If Device Type in Surgery IsPresent And Device in Surgery IsPresent then ... execute the "Lookup Device Type" custom function as a DynamicSearchList on field Device in Surgery

Check Steps

	Type	Step	Edit
1	Data Value	...>Surgery>Device Type>DEVICETYPE>...>...>None	⚙
2	Check Function	IsPresent	⚙
3	Data Value	...>Surgery>Device>DEVICE>...>...>None	⚙
4	Check Function	IsPresent	⚙
5	Check Function	And	⚙

[Add Check Step](#)

Check Actions

Data Point	Action	Edit
...>Surgery>Device>DEVICE>...>...>...	Set DynamicSearchList Lookup Device Type	⚙

[Add Check Action](#)

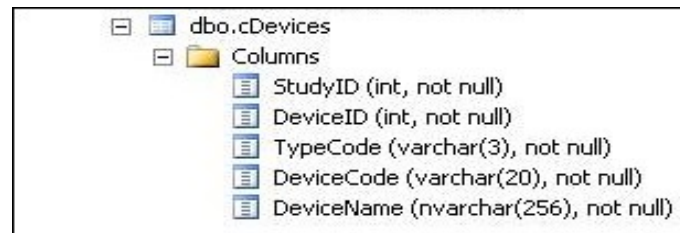
Custom Functions for Dynamic Search List can be implemented using C# or SQL.

	Advantage	Disadvantage
CF	<ul style="list-style-type: none">• Not necessary to install/update custom tables in database.• Can build the list of values on the fly from current data in any forms/folders.	<ul style="list-style-type: none">• Dictionary values depend on CRF version. Any change to the values requires CRF migration.• The list of all possible dictionary values may not fit in 8000 character limit of a Custom Function.
SQL	<ul style="list-style-type: none">• Reusable. Can use the same data in a custom table for more than one study.	<ul style="list-style-type: none">• Must have the knowledge of database tables and structure.• If custom table is necessary, it must be installed in database.

Here is an example of Dynamic SearchList custom function in SQL:

Add the custom SQL table cDevices to the target Rave database.

⇒ The table structure is:



Column	Data Type	Nullability
StudyID	int	not null
DeviceID	int	not null
TypeCode	varchar(3)	not null
DeviceCode	varchar(20)	not null
DeviceName	nvarchar(256)	not null

⇒ The contents of the table should match the data in the Business Rules section of the specification:

	StudyID	DeviceID	TypeCode	DeviceCode	DeviceName
1	317	1	C	CATH STD	Standard Occlusion Balloon Catheter
2	317	2	C	CATH STER	Sterling Balloon Dilation Catheter
3	317	3	C	CATH PERIPH	Peripheral Cutting Balloon Device
4	317	4	C	CATH ANGIO	Imager II Angiographic Catheter
5	317	5	S	STENT DYN	Dynamic (Y) Stent
6	317	6	S	STENT POLY	PolyFlex Airway Stent
7	317	7	S	STENT TRACH	UltraFlex Tracheobronchial Stent System
8	317	8	CL	COIL GDC	GDC Detachable Coil
9	317	9	CL	COIL MATRIX	Matrix2 Detachable Coil
10	318	10	C	CATH STD	Standard Occlusion Balloon Catheter
11	318	11	C	CATH STER	Sterling Balloon Dilation Catheter
12	318	12	C	CATH PERIPH	Peripheral Cutting Balloon Device
13	318	13	C	CATH ANGIO	Imager II Angiographic Catheter
14	318	14	S	STENT DYN	Dynamic (Y) Stent
15	318	15	S	STENT POLY	PolyFlex Airway Stent
16	318	16	S	STENT TRACH	UltraFlex Tracheobronchial Stent System
17	318	17	CL	COIL GDC	GDC Detachable Coil
18	318	18	CL	COIL MATRIX	Matrix2 Detachable Coil

Note: This table should be installed once on the database of the target URL by a qualified DBA. Make sure to create an index on TypeCode column and to populate StudyID with an actual ID of the study of this custom function. For the training class, this table has already been installed.

Write the custom function code and paste into Architect as described in previous sections. Select "SQL" as a language of this function.

```
SELECT DeviceName, DeviceCode FROM cDevices cd
INNER JOIN StudySites sts on sts.StudyID = cd.StudyID
INNER JOIN Subjects sb on sb.StudySiteID = sts.StudySiteID
WHERE TypeCode = '{text='datapoint.standardvalue' fieldoid='deviceType'}'
AND sb.SubjectID = '{subjectid}'
```

⇒ The first line specifies the columns to retrieve and return to Rave.

Note: Dynamic SearchList custom function must return two columns, as in a dictionary. The first column must contain User Data Strings (values to be displayed to the user), and the second column must contain Coded Data.

⇒ Inner joins and SubjectID are required to filter the cDevices rows by StudyID.

⇒ The forth line matches the TypeCode column to the standard value of the the DEVICETYPE field.

Note: In this case, datapoint.standardvalue was specified since the reference field, DEVICETYPE, is attached to a data dictionary. Also, only the fieldoid was specified because both DEVICETYPE and DEVICE are on the same form. For different situations, use one of the other options below.

⇒ Other options:

```
--same form, standardvalue
WHERE TypeCode =
    '{text = 'datapoint.standardvalue' fieldoid = 'myFieldOID'}'

--same form , uservalue
WHERE TypeCode =
    '{text = 'datapoint.uservalue' fieldoid = 'myFieldOID'}'

--different form, same folder, uservalue
WHERE TypeCode =
    '{text = 'datapoint.uservalue' formoid = 'myFormOID' fieldoid =
    'myFieldOID'}'

--different form, different folder, standardvalue
WHERE TypeCode =
    '{text = 'datapoint.standardvalue' folderoid = 'myFolderOID' formoid
    = 'myFormOID' fieldoid = 'myFieldOID'}'
```

Apply the changes by overwriting existing latest CRF Version. For more information on Publishing, Pushing and Overwriting, refer to the Medidata Rave Architect Training Manual.

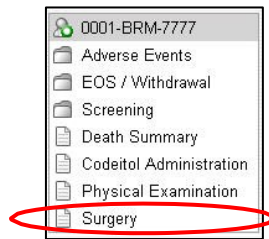
Test whether the business rule for this custom function has been implemented correctly.

⇒ Navigate to the home page.

⇒ Select appropriate Study and Site. Refer to the EDC Training Manual for more information on navigating to Studies and Sites.

⇒ Select the subject created earlier or a create a new subject.

⇒ Navigate to the **Surgery** form on the sidebar.



⇒ For the **Device Type**, select **Catheter**. Then continue to the **Device Field** and click the down arrow. Only Catheter device names will appear.

A screenshot of the 'Surgery' form for patient 0001-BRM-7777. The 'Surgery Date' field is empty. The 'Device Type' field has radio buttons for 'Catheter', 'Coil', and 'Stent'. The 'Catheter' radio button is selected and circled in red. Below it, the 'Device' field has a dropdown arrow circled in red. The device list shows: Imager II Angiographic Catheter, Peripheral Cutting Balloon Device, Standard Occlusion Balloon Catheter, and Sterling Balloon Dilation Catheter. There are 'Save' and 'Cancel' buttons at the bottom right.

⇒ Now click the **Coil** radio button for the **Device Type**. The **Device** list is cleared. Click the arrow again, and a list of coils is displayed.

A screenshot of the 'Surgery' form for patient 0001-BRM-7777. The 'Device Type' field has radio buttons for 'Catheter', 'Coil', and 'Stent'. The 'Coil' radio button is selected and circled in red. Below it, the 'Device' field has a dropdown arrow circled in red. The device list shows: GDC Detachable Coil and Munit2 Detachable Coil. There are 'Save' and 'Cancel' buttons at the bottom right.

Here is an example of Dynamic SearchList custom function in C#:

```
DataPoint dpDSL = ((DynamicSearchParams)ThisObject).DataPoint;
DataPoint dpDeviceType =
dpDSL.Record.DataPoints.FindByFieldOID("DEVICETYPE");
string code = (CustomFunction.DataPointIsEmpty(dpDeviceType)) ?
               string.Empty : dpDeviceType.CodedValue().ToString();
KeyValueCollection devices = new KeyValueCollection();

switch (code)
{
    case "C":
        devices.Add(new KeyValue("CATH STD",
                                "Standard Occlusion Balloon
                                Catheter")); devices.Add(new KeyValue("CATH STER",
                                "Sterling Balloon Dilation Catheter"));
        devices.Add(new KeyValue("CATH PERIPH",
                                "Peripheral Cutting Balloon
                                Device")); devices.Add(new KeyValue("CATH ANGIO",
                                "Imager II Angiographic Catheter"));
        break;
    case "S":
        devices.Add(new KeyValue("STENT DYN", "Dynamic (Y) Stent"));
        devices.Add(new KeyValue("STENT POLY", "PolyFlex Airway
        Stent")); devices.Add(new KeyValue("STENT TRACH",
        "UltraFlex Tracheobronchial Stent System"));
        break;
    case "CL":
        devices.Add(new KeyValue("COIL GDC", "GDC Detachable Coil"));
        devices.Add(new KeyValue("COIL MATRIX",
                                "Matrix2 Detachable Coil"));
        break;
}
return devices;
```

In this C# custom function, the dynamic searchlist values are added programmatically, depending on the value of DEVICETYPE DataPoint.

Writing Custom Function Code: Common Tasks

- ❑ Looping Through EDC Collections
- ❑ Localized Strings
- ❑ Queries, Comments and Stickies
- ❑ Fetching Directly From The Database
- ❑ Other Database Support
- ❑ Sending Emails
- ❑ Reusability

Looping Through EDC Collections

For every class in the Object Model that represents something in EDC, there exists another class that represents a collection of objects of that type. Several situations arise often and necessitate scanning through the objects in such a collection, one at a time.

Finding Multiple Copies of a form

The previous chapter introduced **FindByFormOID**, a method that retrieves a **DataPage** from a collection based on the OID of its defining Form.

The **FindByFormOID** method returns **one DataPage**. If there are multiple **DataPage** objects defined by the specified form, one is returned at random.

When working with different copies of the same form, a different approach is used.

Example - Multiple Copies of a form

A subject can have multiple visit forms.

Assume:

- ❑ this custom function is run as a check action and takes some generic data value as an input
- ❑ the visit Forms are located in the WEEK1 Folder
- ❑ the visit date Field has OID "VISITDATE"

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
DataPoint dpInput = afp.ActionDataPoint;

Subject currentSubject = dpInput.Record.Subject;
Instance instWeek1 =
    currentSubject.Instances.FindByFolderOID("WEEK1");

DataPage dpgTemp; DataPoint dpThisVisitDate;
for (int i=0; i<instWeek1.DataPages.Count; i++)
{
    dpgTemp = instWeek1.DataPages[i];
    if (dpdTemp.Form.OID == "VISIT")
    {
        dpThisVisitDate = dpdTemp.MasterRecord.
            DataPoints.FindByFieldOID("VISITDATE");
        //Now do something with this visit DataPoint
        break;
    }
}
return null;
```

Here, a **for** loop steps through the **DataPage** objects inside the current subject's WEEK1 Instance. For every **DataPage** in the collection, the function tests whether the **DataPage** is defined by a Form with OID VISIT. If so, then the function attempts to retrieve the visit date **DataPoint** from the page's master Record.

Similarly, a `for` loop can be used to work with multiple Instances defined by the same folder. This could be helpful if each visit were contained in a separate 'Basic Visit' folder, rather than grouped in a folder with other visits by week, as in the example above.

Looping Through Records

`for` loops can also scan through log lines on a page to search for a particular data value.

Example of looping Through Records

Based on some user input, a custom function needs to verify the presence of an AE marked as 'Serious'.

Assume:

- ❑ this custom function is run as a check action and takes some non-specific data value as an input
- ❑ the adverse events form has OID 'AE'
- ❑ the adverse events form has a field 'Serious?' with a coded value of '1' equal to a user value of 'Yes'

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;  
DataPoint dpInput = afp.ActionDataPoint;
```

```
bool isSeriousAeFound = false;
```

```
DataPage dpgAE =  
dpInput.Record.Subject.DataPages.FindByFormOID("AE");
```

```
if (dpdAE != null && dpdAE.Active)  
{  
    DataPoint dpIsSerious;  
    for (int i=0; i<dpdAE.Records.Count; i++)  
    {  
        dpIsSerious =  
        dpdAE.Records[i].DataPoints.  
        FindByFieldOID("SERIOUS");  
        if (dpIsSerious.Active && dpIsSerious.Data == "1")  
        {  
            isSeriousAeFound = true;  
            break;  
        }  
    }  
}
```

```
if (isSeriousAeFound) { //Do some work }  
return null;
```

Localized Strings

All text for Rave's queries, comments and stickies is localized, and when placing one of these marking items, custom functions must provide an internal localized string ID rather than the string literal.

One option for working with localized marking items in custom functions is to add the necessary message in the Translation Workbench module and note the value listed in the "Key" column – this is the string ID.

Another way is to take advantage of the following functions:

```
int Localization.AddDataString(string defaultLocaleText)
```

- ❑ Adds a new localized string to the system with the given default locale text. Returns the string ID of the newly created localized string.

```
void Localization.UpdateLocalDataString(int stringID, string  
                                       localizedString, string locale)
```

- ❑ Edits the translation for a given string in the specified locale. If the locale is omitted, the default locale is used.

```
int Localization.FindDataString(string localizedString, string locale)
```

- ❑ Retrieves the ID of the localized string with the specified translation in the given locale. If the locale is omitted, the default locale is used.

Queries, Comments and Stickies

Rave's CustomFunction class provides static methods for managing a DataPoint's queries, comments and stickies (collectively known as marking items).

To ensure that a custom function does not remove marking items that it didn't itself place, Rave requires that developers

- ❑ give information about the edit check that called the custom function when placing a new marking item, in effect labelling it
- ❑ specify this same information when removing a marking item. For the marking item to be successfully removed, its label must match the edit check specified.

This information consists of two parts:

- ❑ **CheckID** is the unique internal identifier for the edit check.
- ❑ **CheckHash** is an encrypted string of all the unique identifiers for the DataPoints that populated the edit check.

Providing Rave with the current CheckID value tells Rave what edit check called the custom function.

CheckHash is specified in order to distinguish between situations where one edit check could be triggered by DataPoints on different log Records.

Obtaining CheckID and CheckHash

All methods for managing marking items require CheckID and CheckHash values. To get references to these two identifiers, extract them from the instance of ActionFunctionParams that is passed to all custom functions configured as check actions.

```
ActionFunctionParams afp = (ActionFunctionParams)ThisObject;  
  
//Need these two values to open/close queries  
int checked = afp.CheckID;  
string checkHash = afp.CheckHash;
```

Queries

This method is used to open and close queries on DataPoints.

```
CustomFunction.PerformQueryAction  
(string QueryText, int MarkingGroupID,  
bool AnswerOnChange, bool CloseOnChange,  
DataPoint queryDP, bool Condition,  
int checkID, string checkHash)
```

Parameters:

- **QueryText**: Text of the Query to place or close.
- **MarkingGroupID**: ID of marking group that will be able to view the query, as defined in Rave's Configuration Module. Never 0.
- **AnswerOnChange**: Corresponds to the **opposite** of "Require Response" from Rave's standard query action.
Example: Specifying true for AnswerOnChange is equivalent to not requiring a response for the query.
- **CloseOnChange**: Corresponds to the **opposite** of "Require Manual Close" from Rave's standard query action.
Example: Specifying true for AnswerOnChange is equivalent to **not** requiring manual close for the query.
- **queryDP**: The DataPoint on which to open/close the query.
- **Condition**: Specifies whether to open a new query or close an existing one.
- **checkID**: Internal identifier of the edit check that called the current custom function. (optional)
- **checkHash**: Hash value passed from the edit check through the ActionFunctionParams structure. (optional)

Comments

This method is used to add and remove comments from DataPoints.

```
CustomFunction.PerformCommentAction(  
    int CommentTextId, DataPoint commentDP,  
    bool Condition, int checkID, string checkHash)
```

Parameters:

- **CommentTextId:** Localized string ID of the query text.
- **commentDP:** The DataPoint on which to place the comment.
- **Condition:** Specifies whether to open a new query or close an existing one.
- **checkID:** Internal identifier of the edit check that called the current custom function.
- **checkHash:** Hash value passed from the edit check through the ActionFunctionParams structure.

Stickies

This method is used to add and remove stickies from DataPoints.

```
CustomFunction.PerformStickyAction(  
    int StickyTextID, int MarkingGroupID,  
    DataPoint stickyDP, bool Condition,  
    int checkID, string checkHash)
```

Parameters:

- **StickyTextId:** Text of comment to be placed.
- **MarkingGroupID:** ID of marking group that will be able to view the sticky, as defined in Rave's Configuration Module. Never 0.
- **stickyDP:** The DataPoint on which to place the sticky.
- **checkID:** Internal identifier of the edit check that called the current custom function. (optional)
- **checkHash:** Hash value passed from the edit check through the ActionFunctionParams structure. (optional)

Fetching Directly From the Database

The following methods can be used to retrieve data values directly from the database without parsing the object hierarchy.

This method returns a collection of **DataPoints** that match a specified OID path.

```
CustomFunction.FetchAllDataPointsForOIDPath  
    (string fieldOID, string formOID, string folderOID,  
     Subject currentSubject)
```

Other Database Support

The following method executes a stored procedure against the Rave database and returns a DataSet object.

```
DataSet CustomFunction.Database.ExecuteDataSet(  
    String dataSourceHint, string procedureName,  
    object[] parameters)
```

Parameters

- **dataSourceHint:** ConnectionSetting shortcut string. Specify null for default value.
- **procedureName:** The name of the stored procedure to execute. Must begin with 'csp'.
- **parameters:** Array of parameter values.

The following method executes a Direct-SQL command against the Rave database and returns the number of rows affected.

```
int CustomFunction.Database.ExecuteNonQuery(  
    String dataSourceHint, string procedureName,  
    object[] parameters)
```

Parameters:

- **dataSourceHint:** ConnectionSetting shortcut string. Specify null for default value.
- **procedureName:** The name of the stored procedure to execute. Must begin with 'csp'.
- **parameters:** Array of parameter values.

The following method executes a stored procedure against the Rave database and returns a scalar object.

```
object CustomFunction.Database.ExecuteScalar(  
    String dataSourceHint, string procedureName,  
    object[] parameters)
```

Parameters:

- **dataSourceHint:** ConnectionSetting shortcut string. Specify null for default value.
- **procedureName:** The name of the stored procedure to execute. Must begin with 'csp'.
- **parameters:** Array of parameter values.

Sending Emails

The following method sends an email without requiring confirmation.

```
Message.SendEmail (
    string To, string From, string Subject,
    string Body, string CC, string BCC,
    [int MessageID, MessageUrgencyEnum urgency])
```

Parameters:

- **To:** Semicolon delimited recipient list.
- **From:** Email sender.
- **Subject:** Email title.
- **Body:** Email contents.
- **CC:** Semicolon delimited CC recipient list.
- **BCC:** Semicolon delimited BCC recipient list.

The following method sends an email requiring confirmation.

```
Message.SendEmailWithConfirmation(
    string To, string From, string Subject,
    string Body, string CC, string BCC,
    [int MessageID, MessageUrgencyEnum urgency])
```

Parameters:

- **To:** Semicolon delimited recipient list.
- **From:** Email sender.
- **Subject:** Email title.
- **Body:** Email contents.
- **CC:** Semicolon delimited CC recipient list.
- **BCC:** Semicolon delimited BCC recipient list.

Note: Provided email addresses need to be fully qualified and valid. For example, "user@mdsol.com" is fully qualified - it contains user name, the "@" sign and the domain "mdsol.com". "user", "user@" or "user@mdsol" are not fully qualified email addresses.

Reusability

Reuse of code streamlines development. There are several approaches:

Within a Custom Function - Breaking a Function into Multiple Methods

A regular block of code pasted into Rave's Custom Function module – such as the one boxed in below – is executed as a single method, **Eval**:

```
public object Eval (object ThisObject)
{
    [-----]
    ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
    DataPoint inputDP = afp.ActionDataPoint;
    //Do some work
    return null;
    [-----]
}
```

Breaking a custom function into multiple methods is a good idea when a piece of code is used again and again. The code on the next page defines a method **GetAllPages** that looks inside a given Instance and returns all of the DataPages that match a given Form OID. It is reused for finding lab DataPages inside of a lab Instance, and visit DataPages within week 1 and week 2 **Instances**.


```

public object Eval (object ThisObject)
{
    ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
    DataPoint inputDP = afp.ActionDataPoint;

    Instance labInstance =
        inputDP.Record.Subject.Instances.FindByFolderOID("LABS");

    Instance week1Instance =
        inputDP.Record.Subject.Instances.FindByFolderOID("WEEK1");

    Instance week2Instance =
        inputDP.Record.Subject.Instances.FindByFolderOID("WEEK2");

    DataPages allLabPages =
        GetDataPagesInInstance(labInstance, "LABFORM");

    DataPages week1VisitPages =
        GetDataPagesInInstance (week1Instance, "VISIT");

    DataPages week2VisitPages =
        GetDataPagesInInstance (week2Instance, "VISIT");

    //Do some work with the lab pages
    return null;
}

//Returns all of the DataPages in a given Instance
//that match a particular Form OID
public DataPages GetDataPagesInInstance
    (Instance instance, string formOID)
{
    DataPages matchingPages = new DataPages();
    for (int i=0; i<instance.DataPages.Count; i++)
    {
        if (instance.DataPages[i].Form.OID == formOID)
        {
            matchingPages.Add(instance.DataPages[i]);
        }
    }
    return allPages;
}

```

Paste boxed part into Rave

In order for a custom function containing multiple methods to work, it must be pasted into Rave as shown in the dashed-rectangle above. In general, the selection to be copied/pasted should start directly after the opening brace of the first method, **Eval**, and end directly before the closing brace of the last method – in this case, **GetAllPages**.

Note: The example above also shows how to use the `new` constructor to create an empty collection of EDC objects. The code:

```
DataPages allPages = new DataPages();
```

inside of the `GetAllPages` method creates a new instance of `DataPages` object with a count of zero. Existing `DataPage` objects can then be added to this collection.

Within a Draft – Calling One Custom Function from Inside Another

Creating helper methods like `GetDataPagesInInstance` above is one way to avoid typing the same code over and over.

Another way is by putting the reusable piece into a separate custom function entirely. In the code below, `GetDataPagesInInstance` has been set up as an entirely new function that can then be called from within the original function.

```
public class MainCustomFunction
{
    public object Eval (object ThisObject)
    {
        ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
        DataPoint inputDP = afp.ActionDataPoint;

        Instance labInstance =
            inputDP.Record.Subject.Instances.FindByFolderOID("LABS");

        DataPages allLabPages = (DataPages)
            CustomFunction.PerformCustomFunction(
                "GetDataPagesInInstance",
                inputDP.Field.CRFVersion.ID,
                new object[]{labInstance, "LABFORM"});

        //Now do some work with the lab pages
        return null;
    }
}

public class GetDataPagesInInstance
{
    public object Eval (object ThisObject)
    {
        //This CF is GetDataPagesInInstance, on CRFVersion 100
        //ThisObject is an array
        //ThisObject[0] = Instance to search
        //ThisObject[1] = Form OID to match
        object[] parameters = (object[])ThisObject;
        Instance instance = (Instance)parameters[0];
        string formOID = (string)(parameters[1]);
        DataPages matchingPages = new DataPages();

        for (int i=0; i<instance.DataPages.Count; i++)
        {
            if (instance.DataPages[i].Form.OID == formOID)
                matchingPages.Add(instance.DataPages[i]);
        }
        return matchingPages;
    }
}
```

The following method is used to execute the second function from within the first:

```
CustomFunction.PerformCustomFunction(  
    string functionName,  
    int CRFVersionID,  
    object[] params)
```

params is the list of parameters passed to the second custom function.

Using Architect Loader Across Projects and/or Drafts

`CustomFunction.PerformCustomFunction` can call any function that lives in the same `CRFDraft`. To take advantage of code saved in other drafts, the Architect Loader must be used to import the helper custom function(s) into the current `CRFDraft`.

For more information, refer to the Medidata Rave Architect or Medidata Rave Architect Loader Training Manuals.

Scenarios Requiring Multiple Custom Edit Checks

- ❑ Determining When Multiple Checks are Required
- ❑ Building Multiple Custom Checks for a Specification

Determining When Multiple Checks are Required

Examples in previous chapters have presented custom functions that are triggered by one edit check. At times, enforcing a new clinical rule will require two or more custom edit checks.

The following specification is used as an example in the procedure outlined below.

Example

For every MED_NAME field on the CONMED log form (subject level), make sure that there is at least one corresponding DIAGNOSIS field on the MEDHISTORY log form (also subject level). If this not the case for a given MED_NAME field, raise a query on it with the following text: "No corresponding diagnosis found on the Medical History Form".

Note: In this procedure, a DataPage is said to "exist" if it is present for the subject and has been submitted at least once.

How to determine if more than one custom check is needed:

If the custom function has only one trigger, or has multiple triggers that are all on the same DataPage, then only one check is required, and the following steps are unnecessary. Otherwise, proceed to step 2.

For each of the triggers, there are two possibilities: either its parent DataPage exists (as defined above), or it does not. List out all combinations of these possibilities.

For the example situation above, there are four possibilities:

- ⇒ Both the CONMED and MEDHISTORY forms have been submitted.
- ⇒ Only the CONMED form has been submitted.
- ⇒ Only the MEDHISTORY form has been submitted.
- ⇒ Neither form has been submitted. (Can be ignored.)

Identify which possibilities represent conditions under which the custom function should execute.

Note: If more than one of these possibilities represents a valid runtime situation, one edit check is needed for each valid possibility.

The example above requires two edit checks because the code that enforces the rule should run under the first two conditions listed above. A query should fire when:

- ⇒ none of the existing MEDHISTORY log lines contains an appropriate DIAGNOSIS value, or
- ⇒ the MEDHISTORY page has not been added for the subject and then submitted at least once.

Building Multiple Custom Checks for a Specification

If multiple custom checks are required, please follow the steps below.

To build the appropriate custom checks when more than one is needed (as determined above):

Put the core logic of the clinical rule into a single custom function that will be referenced as a check action.

Review all of the possibilities from step 2 of the process described previously, entitled “How to determine if more than one custom check is needed.”

For each possibility where it makes sense for the custom function to run, write one edit check that:

- ⇒ references the custom function as a check action, AND
- ⇒ contains check steps that detect changes in any data values on pages that exist in that possibility, using **AlwaysTrue**.

In all of the checks, the custom function should be passed the data value that is present in the check steps of all of the checks.

Example – CONMED and MEDHISTORY

A custom function should be created that checks for the appropriate diagnosis values.

Then, two edit checks should be created, one for each possible combination of pages that could exist for the subject:

- ❑ Both the CONMED and MEDHISTORY Data Pages exist.

Go Back

If DIAGNOSIS in Medical History AlwaysTrue Or MEDNAME in Concomitant Medications AlwaysTrue then... execute the "Example_CF" custom function

Check Steps

Type	Step	Edit
Data Point (Data Point)	...>Medical History>DIAGNOSIS>DIAGNOSIS>...>...>None	
Custom Function (From previous Step)	AlwaysTrue	
Data Point (Data Point)	...>Concomitant Medications>MEDNAME>MEDNAME>...>...>None	
Custom Function (From previous Step)	AlwaysTrue	
Check Function	Or	
Add Check Step		

Check Actions

Data Point	Action	Edit
...>Concomitant Medications>MEDNAME>MEDNAME>...>...>...	Custom Function:Example_CF	
Add Check Action		

- ❑ Only the CONMED DataPage exists.

Go Back

If MEDNAME in Concomitant Medications AlwaysTrue then... execute the "Example_CF" custom function

Check Steps

Type	Step	Edit
Data Point (Data Point)	...>Concomitant Medications>MEDNAME>MEDNAME>...>...>None	
Custom Function (From previous Step)	AlwaysTrue	
Add Check Step		

Check Actions

Data Point	Action	Edit
...>Concomitant Medications>MEDNAME>MEDNAME>...>...>...	Custom Function:Example_CF	
Add Check Action		

In both cases, MEDNAME is passed to the Example_CF custom function.

This makes sense. In both of the scenarios where the custom function should run, DataPoints on CONMED (like MEDNAME) should exist, but MEDNAME need not exist, since MEDHISTORY might not be present and/or submitted for the subject in question.

Appendix A: Training Exercises

The five exercises in this section are to be completed as part of a live Medidata Rave training session.

Each presents a specification detailing a business rule that should be implemented by creating a custom function and one or more edit checks as necessary.

Exercise 1: End of Study Reason

This exercise looks at a business rule that requires using two edit checks and one custom function to scan login lines and look for at least one adverse event marked with the maximum available severity.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	EOS Reason
Short Description:	If the reason for termination is death, then there must be at least one AE log line with a grade of fatal.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable): <i>Death is listed as reason for termination, but there are no adverse events logged with 'Grade' marked as 'Fatal'.</i>	

Elements involved in the Custom Function:			
Folder OID(s):	EOS		
Form OID:	EOS		
Form Name:	End of Study		
Field OID(s)	Pretext	Is Log?	Action
ENRLDT	If no, specify the main reason why not	<input type="checkbox"/>	Open Query
Folder OID(s):	AE		
Form OID:	AE		
Form Name:	Adverse Event Log		
Field OID(s)	Pretext	Is Log?	Action
MEDHX_DT	Grade	<input checked="" type="checkbox"/>	Open Query

Business Rules:

If REASON is marked as 'Death', then there needs to be at least one *active* AE log line with a GRADE value of 'Fatal'.

If there is not, open a query on the REASON field.

Special Instructions:

This custom function should run - and open a query if necessary – even when the AE form has not been submitted.

When looking through log lines for a GRADE of 'Fatal', the custom function should only look at active log lines.

Exercise 2: Death Date Timespan

The solution to this exercise uses three edit checks and one custom function to make sure two fields are within a set number of days of one another.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	Death Date Timespan
Short Description:	If there is a last dose of the study drug within 8 weeks of death, there must be at least one AE log line with a grade of fatal.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable): <i>When there is a last dose within 8 weeks of the death date, there must be at least one AE logged with grade = fatal.</i>	

Elements involved in the Custom Function:			
Folder OID(s):			
Form OID:	DEATH		
Form Name:	Death Summary		
Field OID(s)	Pretext	Is Log?	Action
DEATHDT	Death Date	<input type="checkbox"/>	Open Query
Folder OID(s):	AE		
Form OID:	AE		
Form Name:	Adverse Event Log		
Field OID(s)	Pretext	Is Log?	Action
GRADE	Grade	<input checked="" type="checkbox"/>	None

Folder OID(s):			
Form OID:	SDRUG		
Form Name:	Codeitol Administration		
Field OID(s)	Pretext	Is Log?	Action
LDOSE	Date of Last Dose	<input checked="" type="checkbox"/>	None

Business Rules:

if there is at least one log line on the SDRUG form where LDOSE is within 8 weeks of a valid date entered for DEATHDT, then there *must* be at least one AE log line with GRADE = fatal.

Special Instructions:

Only active log lines should count when looking for LDOSE and GRADE values.

This custom function should run even when the AE form is not submitted, but the SDRUG and DEATH forms must already have been submitted.

Exercise 3: Duplicate log lines

This exercise focuses on making sure no two lines on a log form have the same exact values for all fields.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	Duplicate Diagnosis Log Lines
Short Description:	No two active diagnosis log lines can have identical values for all fields.

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Manual Close Query Required?	<input checked="" type="checkbox"/> Yes <input type="checkbox"/> No <input type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable): <i>Duplicate diagnosis record. Please correct.</i>	

Elements involved in the Custom Function:			
Folder OID(s):	SCRN		
Form OID:	MEDHX		
Form Name:	Medical History		
Field OID(s)	Pretext	Is Log?	Action
MEDHX_DIAG	Diagnosis or Procedure	<input checked="" type="checkbox"/>	Open Query
[all fields]	[all fields]	<input checked="" type="checkbox"/>	None

Business Rules:
Two log lines should be considered identical if all field values on the two lines are identical.
When 2 or more lines are identical, open a query on all identical log lines but the first log line (i.e., smallest recordposition).
The query should open on the MEDHX_DIAG field.

Exercise 4: ANC Calculation

This exercise focuses on doing calculations and populating a DataPoint from within a custom function.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	ANC Calculation
Short Description:	Populate the absolute neutrophil count based on the WBC count, the % segmented neutrophils (mature neutrophils) and the % bands (almost mature neutrophils).

Elements involved in the Custom Function:			
Folder OID(s):	LABS		
Form OID:	HEMLAB		
Form Name:	Hematology		
Field OID(s)	Pretext	Is Log?	Action
ANC	Absolute Neutrophil Count	<input type="checkbox"/>	Set Visible Set Value
WBC	White Blood Cell Count	<input type="checkbox"/>	None
NEUTROPHILS	Neutrophils	<input type="checkbox"/>	None
SEGS	Segmented Neutrophils (fully mature)	<input type="checkbox"/>	None
BANDS	Bands (Almost mature)	<input type="checkbox"/>	None

Business Rules:

Calculate and derive ANC as follows:

If NEUTROPHILS is not blank, then $ANC = WBC * NEUTROPHILS$.

Else, If both SEGS and BANDS are not blank, then $ANC = WBC * (SEGS + BANDS)$

If any of the fields involved (WBC, NEUTROPHILS, SEGS, BANDS) are non conformant or the value is outside of analyte range ANC should be populated blank.

Once ANC is derived, make this field visible. Note:

- ANC and WBC are cell counts with units 10^3 cells/microliter
- NEUTROPHILS indicates the % of white blood cells that are neutrophils
- SEGS indicates the % of white blood cells that are *fully mature* neutrophils
- BANDS indicates the % of white blood cells are *immature* neutrophils.
- $NEUTROPHILS = SEGS + BANDS$, so the user will enter either NEUTROPHILS
- **OR** (SEGS and BANDS).
- An edit check has been already installed to inform the user of this rule above by opening a query in case the user mistakenly enters both NEUTROPHILS and SEGS/BANDS.

Exercise 5: SAE Reconciliation

Exercise 5 shows to how to automate a very common and time-consuming process – reconciliation of AE and SAE forms.

General Requirements	
URL:	Training1.mdsol.com
Project Name:	Training
Type:	<input checked="" type="checkbox"/> Edit Check <input type="checkbox"/> Derivation
Custom Function Name:	SAE Reconciliation
Short Description:	<p>If an adverse event is Serious, add the SAEMAT matrix and auto-populate the new SAE page with the serious AE description and the serious AE Date. If the user changes these values in the AE page, update the related SAE page.</p> <p>Prevent the user from editing the SAE description and start date.</p> <p>If the AE is no longer serious, inactivate the related SAE.</p>

IF an EDIT CHECK, please provide General Edit Check Requirements:	
Response Required?	<input type="checkbox"/> Yes <input type="checkbox"/> No <input checked="" type="checkbox"/> N/A
Manual Close Query Required?	<input type="checkbox"/> Yes <input type="checkbox"/> No <input checked="" type="checkbox"/> N/A
Marking Group(s)	1
Please specify action message (if applicable):	

Elements involved in the Custom Function:			
Folder OID(s):	AE		
Form OID:	AE		
Form Name:	Adverse Event Log		
Field OID(s)	Pretext	Is Log?	Action
SERIOUS	Is this a serious AE?	<input checked="" type="checkbox"/>	None
AESYMP	Symptom	<input checked="" type="checkbox"/>	None
STARTDT	Onset Date	<input checked="" type="checkbox"/>	None

Folder OID(s):	SAE		
Form OID:	SAE		
Form Name:	Serious Adverse Event		
Field OID(s)	Pretext	Is Log?	Action
SAEDATE	Date of Serious Adverse Event	<input type="checkbox"/>	Set Value
AEDESC	Serious Adverse Event Symptom	<input type="checkbox"/>	Set Value
AENUM	N/A (Hidden field. Value is record position of related AE record)	<input type="checkbox"/>	Set Value

Business Rules:

if SERIOUS = yes

THEN

Find SAE datapage with AENUM = RecordPosition of AE.

If SAE not found, add new SAEMAT matrix, find the new datapage and set AENUM.

set SAEDATE = to STARTDT and set AEDESC = to AESYMP.

ELSE if SERIOUS = no

find SAE datapage with AENUM = RecordPosition of AE

If found, inactivate it

else do nothing.

Appendix B: Best Practices for Writing Checks

- ❑ Understanding the Wildcard Match
- ❑ Best Practices for Writing Checks

Understanding the Wildcard Match

It is important to understand the **wildcard match** before looking at best practices for writing standard edit checks.

When leaving any information blank while specifying a DataPoint in the edit check or derivation engine, this is called 'wildcarding,' but it is match wildcarding. Match wildcards differ from 'true wildcarding' in that any blank information will fill down to all check steps and actions based on the DataPoint currently being submitted. This includes folders, forms, record position, form repeat number, and folder repeat number.

How it works (Theory)

When a DataPoint changes, the edit check engine looks for all checks and derivations that are affected by that DataPoint by determining which steps in the check or derivation the DataPoint matches.

What happens next depends on how this match is made. When matching the check or derivation step, if the DataPoint is a match because the Folder specified in the check step is a wildcard, then the engine goes through all the check or derivation steps and fills in all folder wildcards with the current Instance of the DataPoint (Folder). The engine does not fill in folders that are already specified. A second scenario can occur when the folder specified in the check or derivation step and the DataPoint come from the same Folder. In this case, the engine does not resolve the wildcards in the other check or derivation steps. The engine goes through the remaining steps and for each one finds the DataPoints that matches it; clones the check or derivation for each DataPoint; and repeats the above process.

How it works (Explained)

When a data point is initially entered or modified, the edit check engine will find all edit checks that contain a check step related to the data point entered or modified.

Once the check engine finds all the edit checks related to the entered or modified data point, it will iterate through each edit check and find all possible scenarios for that edit check.

(e.g. If a non-log field is compared to a log field, which has three log entries, there will be a edit check checking each log line having 3 edit checks total.)

Data value check steps are qualified in the edit check depending on what is or is not qualified in the data value check step that triggers the edit check.

Note: The check step that triggers the edit check will be referred to as the trigger step from here on out.

There are five variables for a data value check step which are Folder > Form > Field > Variable > Record Position

If a variable is not qualified for the trigger step, the system will determine what that current value is for the trigger step and fill it in.

... > **Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0**

Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0

Since the Folder variable was not qualified in the trigger step and required the system to qualify it, all other data value check steps that do not have the folder qualified will be populated with the same folder value as the trigger step.

Since the trigger step was populated with "Day -1", all other data value check step missing a folder will have the folder variable qualified with "Day -1".

Example:

... > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Day -1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

This concept applies to all 5 variables in a data value check step. One edit check will be evaluated which is:

Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0

vs.

Day -1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

If a variable is qualified for the trigger step, the system will NOT assume the same value for all the other data value check steps with the same variable missing.

Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0

The system will find every instance with the specified variables for the data value check steps that exists.

Example – (assume our Medical History form exists in three folders which are "Screening", "Day -1", and "Day 1"):

... > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Day -1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Screening > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Day 1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Three different edit checks will be evaluated and they are:

Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0 vs. Day -1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0 vs. Screening > Medical History > DIAGC_MH:MH > DIAGC_MH > 0


Day -1 > Medical History > NA_CHECKBOX:MH > NA_CHECKBOX > 0 vs. Day 1 > Medical History > DIAGC_MH:MH > DIAGC_MH > 0

How It Works (Example)

- Medical history form with one standard field combined with four fields which are part of a log form.
- We want an edit check that fires when the "Not Applicable" field, the standard field, is NOT checked and the "Prior History of Condition/Diagnosis" is empty for all active log lines.

Subject: **10006 NTR**
Page: **Day -1 - Medical History (1)**

Not Applicable - check if there is no data to record. ☐

 Log form line 1 of 1
Click here to view complete log...

Prior History of Condition/Diagnosis

Onset Date

Ongoing?* ☐

Specify

How it works (Example Check Steps)

We want an edit check that fires when the "Not Applicable" field, the standard field, is NOT checked and the "Prior History of Condition/Diagnosis" is empty for all active log lines.

Check steps for edit check are listed below

- Two Data Value check steps
- Both Data Value check step do not have the Folder qualified
- One Data Value check step does not have the record position qualified

Type	Step
Data Value	...Medical History>NA_CHECKBOX:MH>NA_CHECKBOX>0
Constant	1 (1)
Check Function	IsNotEqualTo
Data Value	...Medical History>DIAGC_MH:MH>DIAGC_MH>...
Check Function	IsEmpty
Check Function	And

Scenario 1

Assumptions:

- Medical History form is only in a folder called "Day -1"
- The form is initially submitted with the "Not Applicable" field checked and the "Prior History of Condition/Diagnosis" field empty for log line 1.
- A second log line is added and submitted with all the fields blank.
- The "Not Applicable" field is modified and submitted as unchecked
- The two data value check steps are set up as follows:

... > **Medical History** > **NA_CHECKBOX:MH** > **NA_CHECKBOX** > **0**

... > **Medical History** > **DIAGC_MH:MH** > **DIAGC_MH** > ...

The field that is updated is the "Not Applicable" field so the check engine will qualify and missing details for that field.

Folder	Form Name	Field Name	Variable	Record Position
...	Medical History	NA_CHECKBOX :MH	NA_CHECKBOX	0
Day -1	Medical History	NA_CHECKBOX :MH	NA_CHECKBOX	0

- After qualifying the first field it moves on and qualifies all other fields.
- Folder variable was not qualified in the trigger step so all data value check steps with Folder variable missing will be "Day -1"
- The record position was provided for the trigger step so the system will not assume that all check steps missing record position to be 0.

Folder	Form Name	Field Name	Variable	Record Position
...	Medical History	DIAGC_MH :MH	DIAGC_MH	...
Day -1	Medical History	DIAGC_MH :MH	DIAGC_MH	1
Day -1	Medical History	DIAGC_MH :MH	DIAGC_MH	2

Scenario 2

Assumptions:

- Medical History form is only in a folder called "Day -1"
- The form is initially submitted with the "Not Applicable" field checked and the "Prior History of Condition/Diagnosis" field empty for log line 1.
- A second log line is added and submitted with all the fields blank.
- The "Not Applicable" field is modified and submitted as unchecked
- The two data value check steps are set up as follows:

... > **Medical History** > **NA_CHECKBOX:MH** > **NA_CHECKBOX** > ...

... > **Medical History** > **DIAGC_MH:MH** > **DIAGC_MH** > ...

The field that is updated is the "Not Applicable" field so the check engine will qualify and missing details for that field.

Folder	Form Name	Field Name	Variable	Record Position
...	Medical History	NA_CHECKBOX :MH	NA_CHECKBOX	...
Day -1	Medical History	NA_CHECKBOX :MH	NA_CHECKBOX	0

- After qualifying the first field it moves on and qualifies all other fields.
- Folder variable was not qualified in the trigger step so all data value check steps with Folder variable missing will be "Day -1"
- Record Position variable was not qualified in the trigger step all data value missing Record Position variable will be "0"

Folder	Form Name	Field Name	Variable	Record Position
...	Medical History	DIAGC_MH :MH	DIAGC_MH	...
Day -1	Medical History	DIAGC_MH :MH	DIAGC_MH	0

Best Practices for Writing Checks

The following best practices apply to standard edit checks.

Field Edit Checks

There are 5 field edit checks:

- Auto query for Required Data Entry
- Auto query for Non-conformant Data
- Auto query for Future date/time
- Auto query for Data out of Range
- Auto query for Mark non-conformant Data out of Range

The field edit checks are located on each individual field in the form designer page.

The Auto query for **Required Data Entry**, **Non-conformant Data**, and **Future date/time** edit checks can be utilized by checking the corresponding box.

Auto query for **Data out of Range** and **Mark non-conformant Data out of Range** are used by entering a low and/or high number in the appropriate box.

After checking the box, the field edit check is created in the Edit Check portion of Architect by inserting the field and simple edit check message CURRENTLY provided. Field edit checks can be displayed, but not edited, by navigating to the Edit Check portion of Architect, checking Show System edit checks, and selecting the desired edit check.

Note: The Simple Edit Checks messages (located in Configuration | Settings | Simple Edit Checks) must be configured before creating these edit checks

Additionally, there are further options for Require Response and Require Manual Close. If the Simple Edit Check message or require options are changed, all existing field edit checks must be removed and recreated to reflect the new settings.

Keep in mind the messages are generic and can NOT be customized for specific variables. This means for the Auto query for out of range and non-conformant checks the message cannot contain specific range or format information. In these cases it may be better to build a unique standard edit check or to always provide this information as help text and refer to it in the generic message.

One to Many DataPoint Comparisons ('Edit Check Hash')

In any situation where a DataPoint, which occurs once, is being compared to another DataPoint, which occurs multiple times, it is best to open queries (or any other actions associated with a marking group or protocol deviation classes) on the DataPoint that occurs multiple times. This is to avoid any confusion when multiple instances of the same query fire on the same instance of a variable: one that is visible to the user and others that are hidden. This can be potentially confusing to the end user in both 5.2 and 5.4 and above. If there are multiple instances of a variable triggering a query on the same variable and data is fixed in one instance: in 5.2, the query will completely disappear even though it is still applicable; while in 5.4 and above, there will be no visible change to the query.

Edit Check Hash' is most commonly an issue on edit checks between log and standard fields and those between one form and multiple instances of another form. Example: If you

want to fire a query when the EVAL_DATE on the Forms VITALS, LABS, and ECG in the folders DAY1, WEEK1, WEEK2, and WEEK3 respectively, are after the DEATH_DATE date field on the DEATH form. In this case, it is best to fire a on the EVAL_DATE field.

Note: This best practice is followed in all examples contained in this document.

Specifying Fields and Folders

If an edit check or derivation references two DataPoints, in either the steps or the actions on different Forms, you must specify the Field and Form identifier. If the check or derivation references two DataPoints in different Folders, you usually specify the Field, Form, and Folder. However, it is desirable to specify only the Folder and the Variable in some cases.

This does NOT apply if the DataPoint is being intentionally wild carded, but remember that the non-wild carded DataPoint must be fully specified.

Specifying Nested Folders

If an edit check or derivation references a field within a nested folder (a sub-folder that is assigned to a 'Parent Folder') and it is necessary to specify the folder in the check steps and/or actions, the sub-folder should be specified. Otherwise, the edit check engine may not be able to find the DataPoint.

Use Data Values Correctly

When a DataPoint is used as an edit check or derivation step, the type of value must be selected depending on format and check function. See below for guidelines on use. Keep in mind there may be certain situations where it is necessary to deviate from these suggestions.

- ❑ **Standard Value:** This is the value stored in the database and should be used for the 'Is Greater/Less Than', 'Is Greater/Less Than Or Equal To' and 'Is Not Empty' check functions and most derivation functions. It is also used for all DataPoints that have a checkbox format.
- ❑ **User Value:** This is the data entered by the user into free textboxes. If non-conformant data is entered in the system, edit checks which call on that DataPoint will not run, unless User Value is specified. This should, in general, only be used for the 'Is Empty' check function.
- ❑ **Coded Value:** This points the system to look at the Coded Data when a DataPoint is associated with a dictionary and is generally used with the 'Is Equal To' and 'Is Not Equal To' check functions and derivation functions for DataPoints with Dictionaries.
- ❑ **Data Point:** This is generally used with the 'Is Non-conformant' check function and Custom functions.
- ❑ **Data Status:** This refers to the empty/touched/reviewed/etc status of the DataPoint and is currently not used.

Range and DataPoint Comparison Checks

When using the edit check functions 'Is Greater/Less Than' or 'Is Greater/Less Than Or Equal To' or 'Is Not Equal To', that edit check should also include 'And Is Not Empty' statements for those DataPoints. This is to prevent erroneous edit check messages from firing.

For example, there is an edit check that specifies that Age must be greater than 18. If the Age is empty, the edit check will fire because the system views null as zero, but by including the caveat that Age is not empty, this is prevented from occurring.

Note: There may be times when it is not desired to use 'And Is Not empty' with 'Is Not Equal To', please be sure to check if the client wants the edit check to fire if the DataPoint is empty. This best practice is followed in all examples contained in this document.

Checkboxes

Do NOT use the 'Is Empty' or 'Is Not Empty' edit check steps with checkboxes. If checkboxes are not checked, they have a value of 0 and if they are checked, they have a value of 1 so therefore can never be empty. When using checkboxes in an edit check, use the 'Is Equal To' check function in conjunction with the appropriate value.

Constants as Check Steps

When using constants in edit checks, be sure the format is correct. If the variable is formatted as text, the constant should be formatted as text. If the variable is numeric, then the constant should be numeric as well. The format should also be in the same format used in the Form Design pages.

Set DataPoint Visible Check Action

The Set DataPoint Visible check action allows for two different actions: show and hide. If the 'Visible' box is checked, the action is set to True and will display a hidden field. If the 'Visible' box is not checked, the action is set to False and will hide fields that previously were visible.

Note: These actions will NOT occur until the page has been submitted so make sure this is clear to the client. This means that added fields will appear as 'untouched' giving the form an incomplete status, but no queries will fire containing these DataPoints. In the case of hiding fields, it is possible for data to have been erroneously entered causing queries to fire. If the incorrect data is captured in a hidden field, it will not be possible to correct the data until the fields is once again unhidden. Keep in mind it is better to show rather than hide DataPoints.

Add Matrix vs. Merge Matrix Check Actions

Add Matrix and Merge Matrix are both used to add forms and folders to the EDC Module that have been included in a matrix. The difference between Add Matrix and Merge Matrix is that Add Matrix is used if the folder does NOT exist in the subject prior to the edit check and Merge Matrix is used to add forms to existing folders. If an edit check is being used to add a combination of these two situations it is necessary to split the Matrices into one that will function correctly with the Add Matrix function and another for Merge Matrix. If Add Matrix is used incorrectly, duplicate folders will be added while if merge matrix is used incorrectly, folders that do not exist will not be added.

Do not over Qualify

Do not qualify DataPoints more than required. If all of DataPoints are on the same form, the folder and the field do not need to be specified. This saves time and reduces mistakes where DataPoints could be qualified incorrectly. It also can allow the reuse of edit checks across multiple forms. See below. The form and field should be specified if the same set of Variables exists as a group on two different Forms, but the check is only applicable to one of the forms.

Reuse an Edit Check

If the same group of DataPoints exists on multiple Forms, or a Form is dynamically added repeatedly in the same Folder, and you want an edit check or derivation to work on all of these Forms, do not fully qualify the steps. By only specifying the Variable you enable the check to fire only on the current instance of the Form that was submitted, and not across all Forms that have the same group of DataPoints. This can help decrease the time the system requires to submit a form in the EDC Module by limiting the number of edit checks that run only to the applicable form.

A Single Form Checks the Same Field Across Multiple Folders

For example, you want to ensure that the date field FUP_DATE on the form FOLLOWUP that resides in all three folders FOLLOWUP1, FOLLOWUP2, and FOLLOWUP3, has a date that comes after the IC_DATE date field on the form DEMOG in the folder SCREEN. Write one edit check, fully qualify the check step that references IC_DATE, and for the remaining steps only specify the form and field, assuming the variable is not unique. If the folder was selected to assist in locating the desired field/variable, click 'apply to all folders' and the selected folder will be replaced with the wildcard. See below. Note: If the FUP_DATE Variable is unique the form does not need to be selected. Also notice the query is fired on the FUP_DATE, see step #3, 'Edit Check Hash' for further details.

Example of what:

Folder	Form	Field	Variable	Data Value
...	FOLLOWUP	FUP_DATE	FUP_DATE	Standard Value
SCREEN	DEMOG	IC_DATE	IC_DATE	Standard Value
Is Less Than				
...	FOLLOWUP	FUP_DATE	FUP_DATE	Standard Value
Is Not Empty				
SCREEN	DEMOG	IC_DATE	IC_DATE	Standard Value
Is Not Empty				
And				
THEN				
...	FOLLOWUP	FUP_DATE	FUP_DATE	
Fire Query: Follow-Up Date must be greater than the Informed Consent Date. Please revise.				

Multiple Forms Check the Same Field Across a Single Folder

For example, you want to make sure that all the date fields EVAL_DATE on the Forms VITALS, LABS, and ECG that all reside in the DAY1 folder, are equal to the DOSE_DATE date field on the form STUDYMED in the Folder STUDYMED. Write one edit check, fully qualify the check step that references DOSE_DATE, and for the remaining steps, specify the folder DAY1, and select any of the forms that the field resides on in order to select the field. Click the 'apply to all fields' checkbox, and the selected form will be replaced with the wildcard.

Also notice the query is fired on the EVAL_DATE, see step #3, 'Edit Check Hash' for further details.

Example of what:

Folder	Form	Field	Variable	Data Value
DAY1	EVAL_DATE	Standard Value
STUDYMED	STUDYMED	DOSE_DATE	DOSE_DATE	Standard Value
Is Not Equal To				
DAY1	EVAL_DATE	Standard Value
Is Not Empty				
STUDYMED	STUDYMED	DOSE_DATE	DOSE_DATE	Standard Value
Is Not Empty				
And				
And				
THEN				
DAY1	EVAL_DATE	
Fire Query: Evaluation Date should be equal to the Study Medication Administration Date. Please revise or clarify.				

Multiple Forms Check the Same Field Across Multiple Folders

For example, you want to make sure that the date field EVAL_DATE on the Forms VITALS, LABS, and ECG in the folders DAY1, WEEK1, WEEK2, and WEEK3 respectively, are after the IC_DATE date field on the DEMOG form in the Folder SCREEN. Then write one edit check, fully qualify the check step that references IC_DATE, and for the remaining steps only specify the field. The wild card match will take care of the rest, generating a check for each situation. If, in order to locate the desired variable, the folder and form had been qualified, click the 'apply to all folders' checkbox, and the selected folder (if any) will be replaced with the wildcard. Click the 'apply to all fields' checkbox, and the selected form will be replaced with the wildcard. Also notice the query is fired on the EVAL_DATE, refer to #3 above, 'Edit Check Hash' for further details.

Example:

Folder	Form	Field	Variable	Data Value
...	EVAL_DATE	Standard Value
SCREEN	DEMOG	IC_DATE	IC_DATE	Standard Value
Is Less Than				
...	EVAL_DATE	Standard Value
Is Not Empty				
SCREEN	DEMOG	IC_DATE	IC_DATE	Standard Value
Is Not Empty				
And				
And				
THEN				
...	EVAL_DATE	
Fire Query: Evaluation Date must be greater than the Informed Consent Date. Please revise.				

Record Position- Checks between Logs and Standard Fields

For edit checks or derivations between log and standard fields, the Record Position must be set to 0 for all check/derivation steps and actions that reference a standard field. If the Record Position of the standard fields is left wild carded, the system will attempt to match the record position of the log line being submitted so the edit check will not run. Also when firing queries (or any other actions associated with a marking group or protocol deviation class) between logs and standard fields, it is best to place the query on the log field. See 'Edit Check Hash.'

EXCEPTION: When an edit check between log and standard fields only references the Variable OIDs, the record position should not be set in any of the check steps or actions. This is because the log property is associated with the field and NOT the variable. For example, if an edit check is desired to require the AE_DESC (log field) when AE_YN (standard field) is equal to Yes (1), the record position of AE_YN must be set to 0. Both variables are located on the AE form in the AE folder.

Folder	Form	Field	Variable	Data Value
...	AE	AE_YN	AE_YN	0
Is Equal To				
...	AE	AE_DESC	AE_DESC	...
Is Empty				
And				
THEN				
...	AE	AE_DESC	AE_DESC	
Fire Query: Adverse Event description is required. Please provide.				

Record Position

Checks between Standard Fields Only: In certain circumstances, it is necessary to specify the record position on an edit check where only standard fields are present as check steps and actions. This usually occurs when actions are being performed on a standard field that resides on a log form. A common example of when this would be necessary, is adding a form using a standard field on a log form. If the record position is not set to 0, a new form will be added for every log line submitted.

Form Repeat Number

If a non-repeating form is compared to any repeating forms (applies ONLY when the form is repeated within the same folder), then the form repeat number of the non-repeating forms must be specified as 0. If the form repeat number is NOT specified, the edit check or derivation will only work on the first instance of the form created so be sure to check any cross form edit checks or derivations on at least two instances of a repeating form. For example, if an edit check is desired to compare the IC_DATE on the DEMOG form in the SCREEN folder to the UNSLAB_DATE to the UNSLAB form in the UNSLAB folder (Add Event, UNSLAB folder is reusable), the folder repeat number of the DEMOG form must be set as 0.

Note: When comparing two repeating forms, a custom function is required. Unless only those forms with the same repeat number are being compared.

Folder	Form	Field	Variable	Data Value
UNSLAB	UNSLAB	UNSLAB _DATE	UNSLAB _DATE	...
SCREEN	DEMOG	IC_DATE	IC_DATE	0
Is Less Than				
UNSLAB	UNSLAB	UNSLAB _DATE	UNSLAB _DATE	...
Is Not Empty				
SCREEN	DEMOG	IC_DATE	IC_DATE	0
Is Not Empty				
And				
And				
THEN				
UNSLAB	UNSLAB	UNSLAB _DATE	UNSLAB _DATE	
Fire Query: Unscheduled Lab Date must be greater than the Informed Consent Date. Please revise.				

Folder Repeat Number

If a non-repeating folder is compared to any repeating folders (ie- Add Events), then the folder repeat number of the non-repeating forms must be specified as 0. If the folder repeat number is NOT specified, the edit check or derivation will only work on the first instance of the folder created so be sure to check any cross folder edit checks or derivations on at least two instances of a repeating folder. For example, if an edit check is desired to compare the IC_DATE on the DEMOG form in the SCREEN folder to the VISIT_DATE to the UNSVISIT form in the UNSVISIT folder (Add Event), the folder repeat number of the SCREEN folder must be set as 0.

Note: When comparing two repeating folders, a custom function is required, unless only those folders with the same repeat number are being compared.

Folder	Form	Field	Variable	Data Value
UNSVISIT	UNSVISIT	VISIT_DATE	VISIT_DATE	...
SCREEN	DEMOG	IC_DATE	IC_DATE	0
Is Less Than				
UNSVISIT	UNSVISIT	VISIT_DATE	VISIT_DATE	...
Is Not Empty				
SCREEN	DEMOG	IC_DATE	IC_DATE	0
Is Not Empty				
And				
And				
THEN				
UNSVISIT	UNSVISIT	VISIT_DATE	VISIT_DATE	...
Fire Query: Unscheduled Visit Date must be greater than the Informed Consent Date. Please revise.				

Checks are Log Line Specific

Checks and derivations do not cross log lines. If a check or derivation is activated by a DataPoint on log line 12, then the engine by default looks for DataPoints whose record is also log line 12 when resolving the other DataPoints in the check or derivation. See the above issue on how to get around this in some situations.

EXCEPTION: Logical record position is a way to implement cross log line checks.

Logical Record Position (LRP)

LRP exists independently of the normal Record Position. Any Check or Derivation Step that references a DataPoint can be further qualified with the following values for Logical

RecordPosition: MinBySubject, MinByInstance, MinByDataPage, MaxByDataPage, MaxByInstance, MaxBySubject, First, Previous, Next, and Last. NOTE: The form and field should be specified when using LRP.

The Min and Max LRP functions find the minimum or maximum value of a field on the level specified; for numbers, this would be the lowest or highest value entered and for strings, it is ordered alphabetically. NOTE: This can slow the system substantially depending on the amount of data the system must search through and it may be better to use custom functions in some cases. An example of how to use LRP Max/Min would be in a derivation to calculate the difference between the maximum and minimum HEART_RATE on the VITALS form, the example follows:

Example:

Folder	Form	Field	Variable	Data Value
...	VITALS	HEART_RATE	HEART_RATE	MaxbySubject
...	VITALS	HEART_RATE	HEART_RATE	MinbySubject
Subtract				
THEN Set Value to				
...	DERIVATIONS	Z_DIFF_HEART		
_RATE	Z_DIFF_HEART			

Qualify Steps and Actions the Same

If a DataPoint appears as both a check step and a check action, then qualify that DataPoint the same in both the step and the action. While having different qualifications for a DataPoint in both a step and an action is legal, and may be desired in some cases, it can cause unintended behavior.

Requires Response and Requires Manual Close

When opening a query on a DataPoint, there are additional options to Require Response and Require Manual Close.

Require Response provides a text box for the CRC to provide reasons for why data may be different than expected.

Require Manual Close means that a query will not close even if a response is entered, until a user with close query rights manually closes the individual query.

It is important to realize that these usually are used together. If a query only Requires Response, after the CRC enters a response, the query will close on its own, even if the data is not corrected. If data is changed so that the edit check no longer fires, queries that Require Response will automatically include 'Value Changed' as the answer to the query in the audit trail and those that Require Manual Close will close by themselves.

Use of HTML – is this still the case?

Do NOT use HTML tags in edit check messages. They won't apply the format, but will display as part of the text.

Do not Create Cyclical Derivations

If a derivation derives the Variable LD_POSDAT, then you cannot reference the Variable LD_POSDAT in any of the steps of the Derivation. The derivation will not run.

Hidden Derivations or DataPoints in Edit Check

Do NOT hide DataPoints used in derivations by setting the 'Is Visible' property to False, instead use view restrictions. This will also better enable troubleshooting because the hidden derivations can be left visible to the Developer role.

Placement of View Restricted Derivations

In 5.2, hidden fields can affect the status of forms. If the derivations do not run the form will appear as incomplete to most clinical users, which can be potentially confusing. To avoid this, place any hidden derivations on a separate derivations form. In 5.4/5.5, hidden fields do NOT affect the status and so can be placed directly on the appropriate form.

Derivations Between Log and Standard Fields

This is similar to 'Edit Check Hash.' When calculating derivations between log and standard fields, the derived field should also be a log field placed on the log form. If the derivation is a standard field it will recalculate every time a new log line is submitted.

DaySpan VS AddDay in Derivations

Consider the edit check that requires DATE2 to be five days either before or after DATE1. This requires a derivation, and can be done in one of two ways.

The first is to derive two dates, one five days before DATE1, the other five days after DATE1, and then to compare DATE2 to both those dates.

The second, and more efficient (since it requires one derivation instead of two), way is to derive the DaySpan between DATE1 and DATE2, and to make sure that this number equals five. On the other hand, if the check required that DATE2 be five days after DATE1, it would be better to derive the date five days after DATE1 and to crosscheck DATE2 with the derived date. This is because the DaySpan derivation always returns a positive value, and will allow a date five days before DATE1 to be entered without firing the edit check.

This principle applies to any DataPoint that is to be checked to ensure its value, and the value of subsequent similar variables, falls within a certain fixed range, e.g. two days after a certain date, or 15 mmHG more or less than an initial blood pressure reading.

When non-consistent values are involved, for example, DATE2 may be two days before DATE1 or three days after DATE1, the two derivation method is preferred for accuracy's sake.

Know the Wildcard Match

Understand the wild card match. While this isn't a best practice, it is the key to understanding how the check and derivation engine find DataPoints.

Appendix C: Commonly Used Objects, Properties and Methods

The following topics are discussed in this chapter:

- Commonly Used Objects, Properties and Methods

Commonly Used Objects, Properties and Methods

Object	Property/Method	Usage
ActionFunctionParams		Object passed into a Custom Function from Edit Check action, other than Dynamic Search List. ActionFunctionParams afp = (ActionFunctionParams)ThisObject; Note: if a CF is called from a Derivation or EC step, then the object passed is a DataPoint DataPoint dp = (DataPoint)ThisObject;
	ActionDataPoint	Actual DataPoint passed to the CF
	ActionResult	Boolean result of the evaluation of the EC steps
DynamicSearchParams		Object passed into a Custom Function's Dynamic Search List action. DynamicSearchParams dsp = (DynamicSearchParams)ThisObject;
	DataPoint	Actual DataPoint passed to the CF
DataPoint	Active	Check if datapoint is active: if (dp.Active) {}
	EntryStatus	Check if datapoint is non conformant: if (dp.EntryStatus == EntryStatusEnum.NonConformant) {} Check if datapoint is not empty and conformant: if (dp.EntryStatus == EntryStatusEnum.EnteredComplete) {}
	FreezeDisplayStatus	Check if datapoint is Frozen or Locked: if (dp.FreezeDisplayStatus == FreezeDisplayStatusEnum.Frozen) if (dp.FreezeDisplayStatus == FreezeDisplayStatusEnum.Unfrozen) if (dp.LockDisplayStatus == LockDisplayStatusEnum.Locked) if (dp.LockDisplayStatus == LockDisplayStatusEnum.Unlocked)
	Field.VarDataFormat Variable.DataFormat Field.ControlType	Get DataPoint's variable format: dp.Field.VarDataFormat dp.Variable.DataFormat Check if the DataPoint is of DateTime format:

		if (dp.Field.ControlType == "DateTime") { }
	DataDictionaryEntry	Check if DataPoint dp has a dictionary attached: bool isDictionary = (dp.DataDictionaryEntry != null);
	IsObjectChanged	Will be true if the datapoint is being changed at this current moment, while the CF is executing.
	ChangeCount	How many times the value of the datapoint was changed. Initially for unsubmitted datapoint ChangeCount = 0
	Field.OID	OID of the field, from which the datapoint was created
	Field.PreText	Label of the field from which the datapoint was created
	Field.PostText	Fixed unit of the field which the datapoint was created
	Enter()	Enter a value into a datapoint, i.e.: dp.Enter("data", "unit", 0); Note: for complete details, see "Enter Data" in the Custom Function Actions section.
		Reading data from a datapoint with an associated dictionary: If the following data is submitted: Coded Value = 5, User Value = Fatal Get datapoint's value exactly as it is stored in DB (as entered by the user): dp.Data Note: For a dictionary field, dp.Data returns the coded value. For Dynamic Search List dp.Data returns the user value. Get datapoint's value converted to standard units. dp.StandardValue().ToString() Get user value from a dictionary field: dp.UserValue().ToString() Get coded value from a dictionary field: dp.CodedValue().ToString() Get coded value selected in a Dynamic Search List: dp.AltCodedData

		<p>Note: If a datapoint with associated dictionary is submitted empty then dp.CodedValue() and dp.UserValue() will all return null.</p> <p>Getting data from dictionaries with "specify" option used: If a coded value 99 with "Specify" is selected and the "Specify" option is filled out with "xyz", then</p> <pre>dp.DataDictionaryEntry.Specify = true; dp.Data = "xyz" dp.UserValue() = "xyz" dp.CodedValue() = 99 dp.StandardValue() = 99</pre> <p>Getting DateTime value from a datapoint</p> <pre>Object obj = dp.StandardValue(); if (obj is DateTime) { DateTime date = (DateTime)obj; }</pre>
Record	Active	Used to check if the record is active, inactivate or re-activate it.
	IsObjectChanged	Will be true if the record is being inactivated or re-activated at this current moment, while the CF is executing.
	RecordPosition	Standard records have RecordPosition = 0.
DataPage	Active	Used to check if the datapage is active, inactivate or re-activate it.
	dpg.Form.OID	OID of the form, from which the datapage was created
	PageRepeatNumber	Ordinal number of a repeated datapage under the same parent, starts with 0
	MasterRecord	Returns the record from the datapage with RecordPosition = 0 (standard master record)
	AddLogRecord()	Add a new unsubmitted record
Instance	Active	Is folder active
	Folder.OID	OID of the folder, from which the instance was created
	ParentInstance	Parent instance of the current instance
	InstanceRepeatNumber	Ordinal number of a repeated instance under the same parent, starts with 0
Subject	Name	Name of the subject

StudySite	Users	Users assigned to the study and site
Site	Number	Site number
Study	Environment	Study environment (DEV, UAT, etc.)
	Name	Name of the project
	Users	Users assigned to the study
DataDictionary	Entries	Get a list of dictionary entries
	Entries[i].CodedData	Get the coded value of a specific entry
	Entries[i].UserDataString	Get the user value of a specific entry
	FetchByOID()	Fetch an existing dictionary by it's OID: DataDictionary.FetchByOID(dictionaryOID, currentPage.Form.CRFVersion.ID) Note: since Dictionary OID is not displayed in Rave, it has to be obtained from database.
ActionFunctionParams	ActionDataPoint	The DataPoint passed to the CF
	ActionResult	result of the EC steps evaluation
	CheckHash CheckID	Used for performing Queries, Comments, Stickies.
Interaction	TrueUser.Login TrueUser.Email	Login and Email of the user, who's currently logged in and triggered the executing CF. Interaction property can be obtained from datapoint, record, datapage, instance or subject Note: do not use User, because Interaction.User is Rave itself.

Searching for Data

DataPoint(s)	Finding another datapoint on the same record DataPoint dp = dpStart.Record.DataPoints.FindByFieldOID("FIELDROID"); Finding collection of datapoints with speficied OID path: Across subject: DataPoints dps1 = CustomFunction.FetchAllDataPointsForOIDPath("FIELDROID", "FORMOID", "FOLDEROID", subject); Across datapage: DataPoints dps2 = CustomFunction.FetchAllDataPointsForOIDPath("FIELDROID", "FORMOID", "FOLDEROID", dataPage);
--------------	--

	<p>If used to find log datapoints on log forms, then it will NOT return datapoints of MasterRecord from log forms</p> <p>If used to find standard datapoints on log forms, then it will NOT return datapoints with the same OID from log records</p> <p>This method DOES return datapoints just being submitted with newly entered value</p> <p>To wildcard a form or a folder (to search for a datapoint in any form/folder) use null for these parameters.</p> <p>Never use activeOnly parameter from overloaded method. Instead, filter out inactive datapoints within a loop across the returned collection.</p>
DataPage	<p>DataPage dpg = dpStart.Record.DataPage.Instance.DataPages.FindByFormOID("FORMOID");</p> <p>Note: FindByFormOID() can only return one datapage. If there can be multiple instances of datapages with the same Form OID, then this method cannot be used. Multiple datapages can be found by looping across all datapages or by using FetchAllDataPointsForOIDPath().</p> <p>DataPage dpgPrimary = dpStart.Record.DataPage.Instance.Subject.PrimaryDataPage;</p>
Record	<p>Looping across all records:</p> <pre>for (int i = 0; i < dpStart.Record.DataPage.Records.Count; i++) { Record record = dpStart.Record.DataPage.Records[i]; }</pre> <p>Finding a master record on any datapage:</p> <p>Record recMaster = dpStart.Record.DataPage.MasterRecord;</p>
Instance	<p>Instance ins = dpStart.Record.DataPage.Instance.Subject.Instances.FindByFolderOID("FOLDEROID");</p> <p>Note: FindByFolderOID () can only return one instance. If there can be multiple instances of datapages with the same Form OID, then this method cannot be used. Multiple instances can be found by looping across all instances or by using FetchAllDataPointsForOIDPath().</p>
User	<pre>for (int i = 0; i < dp.Record.Subject.StudySite.Study.Users.Count; i++) { User user = dp.Record.Subject.StudySite.Study.Users[i]; Role role = user.UsersRoleInStudy(dp.Record.Subject.StudySite.Study); if (role != null && role.RoleName == "TRIAL MANAGER") { //User found string email = user.Email; } }</pre>
DateTime	<p>//Get a DateTime in user's local time zone:</p> <pre>DataPoint dp = ((ActionFunctionParams)ThisObject).ActionDataPoint; int localZoneID = dp.Interaction.TrueUser.TimeZone; DateTime dateTime = Timezone.LocalTime(localZoneID);</pre>

Custom Function Actions

Freeze/UnFreeze a datapoint	<pre>dp.Freeze(); dp.UnFreeze();</pre>
Lock/UnLock a datapoint	<pre>dp.Lock(); dp.Unlock();</pre>
Query	<p>CustomFunction.PerformQueryAction() method always matches the query by text.</p> <p>Parameters CheckID and CheckHash should always be provided in this method, so that if an EC is inactivated or deleted, the queries opened by it will be cancelled during migration.</p> <p>In order to change a query message in a CF for a new CRF version, EC, from which the CF is called, has to be replaced, so it would have a new CheckID and CheckHash:</p> <ol style="list-style-type: none"> 1. Go to the EC where the CF is called from. 2. Open this EC with Quick Edit and append "1" to the EC name. 3. Save the EC. Now there will be a new EC with identical function. 4. Delete the original EC. 5. Rename the new EC back to original name, by removing "1" from step 2. Now this EC will have a new CheckID. 6. Do the migration. 7. After migration has finished, original query with old text will be canceled and, instead, a query with new text will be opened. <p>If CheckID and CheckHash are not used in CustomFunction.PerformQueryAction(), then the old queries will remain open. CheckID and CheckHash are now being used for opening queries by simple ECs as well.</p>
Enter Data	<p>Enter data to a datapoint without a unit:</p> <pre>dp.Enter("data", null, 0); dp.Enter("data", string.Empty, 0);</pre> <p>Enter data to a datapoint with a unit:</p> <pre>dp.Enter("data", "unit", 0);</pre> <p>Populate a datetime field using specific format. The field's variable must be of this same format:</p> <pre>string strTimeStamp = DateTime.UtcNow.ToString("dd MMM yyyy HH:nn:ss"); dpTime.Enter(strTimeStamp, string.Empty, 0);</pre> <p>Enter coded value from dictionary "O" with "specify" string "A,B,C"</p> <pre>dp.Enter("99,A,B,C", null, 0);</pre> <p>Note: everything in front of the first "," is a coded value, after the first "," is the "specify" text. Therefore, dictionary coded data string must not contain ",".</p> <p>Enter data into a datapoint associated with Dynamic Search List (DSL):</p> <pre>dpDSL.Enter(strUserValue, strCodedValue, "", 0, DataEntryState.Standard);</pre> <p>Note: DataEntryState.None defaults to DataEntryState.Standard.</p>

	<p>dpDSL.Enter(strData, "", 0) does not work for datapoints associated with DSL.</p> <p>Change codes: Change codes are defined in Configuration > Settings > Change Codes. In CF, use 0 as the default change code. When change code 0 is specified, the very first defined code (with the smallest code value) will be used. On training1.mdsol.com this default code is "Data Entry Error".</p> <p>To find out the code for a specific type of change, run this script against the DB of the url: Select ChangeCode, dbo.fnLDS(StringID, 'eng') as [Name] From ChangeCodes</p>
Add a record	dataPage.AddLogRecord()
Add a form	<p>Form form = Form.FetchByOID("FORM_OID", dp.Field.CRFVersion.ID); subject.AddCRF(parentInstance, form, dp.Record.SubjectMatrixID);</p> <p>Note: parentInstance is a subject or an instance of folder to add the form to</p>
Add a folder	<p>Folder folder = Folder.FetchByOID("FOLDER_OID", dp.Field.CRFVersion.ID); subject.AddInstance(parentInstance, folder, dp.Record.SubjectMatrixID);</p> <p>Note: parentInstance is a subject or an instance of folder to add the form to</p>
Add a matrix	<p>Matrix matrix = Matrix.FetchByOID("MATRIX_OID", subject.CRFVersion.ID); subject.AddMatrix(matrix);</p>
Set datapoint as (non)conformant	<p>Set dp non conformant: dp.SetNonConformant(true);</p> <p>Set dp conformant: dp.SetNonConformant(false);</p>
Set datapoint visible/invisible	<p>dp.IsVisible = true; dp.IsVisible = false;</p>
Send email	<p>Message.SendEmail("user@mdsol.com", "rave@mdsol.com", "subject", "body"); Message.SendEmail("user1@mdsol.com;user2@mdsol.com", "rave@mdsol.com", "subject", "body");</p> <p>Note: user emails can be delimited by ";" or ",".</p>
Inactivate datapage or instance	To remove a datapage or instance inactivate them by setting their Active property to false.

Generic Custom Functions

Generic Custom Function:

```

public class CfLibrary_GetSortedDataPoints
{
    public object Eval(object ThisObject)
    {
        /*
        Developed by:
        Last modified by:
        Last modified on:
        Description: This generic CF will return datapoints sorted
by data and then by record position.
        Parameters:
            DataPoints dataPoints (of string format)
            bool order (true = Ascending)
            bool excludeInactive (true = exclude inactive
            datapoints)
        Returns: DataPoints
        In case of invalid parameters returns null;
        */
        object[] parameters = (object[])ThisObject;

        //Validate parameters
        if (parameters.Length != 3 || !(parameters[0] is
            DataPoints))
            return null;

        //Get parameters
        DataPoints dps = (DataPoints)parameters[0];
        isAscending = (bool)parameters[1];
        bool excludeInactive = (bool)parameters[2];
        /*
            Implement the logic for the requirement
        */
        return dps;
    }
}

```

Calling a generic CF

```

Object[] parameters = new object[] { parameter1, parameter2, parameter3 };
DataPoints dps = DataPoints)CustomFunction.PerformCustomFunction("CfLibrary_CFName",
start_DP.Field.CRFVersion.ID, parameters);

```


Dynamic Search List

SQL

```
SELECT DeviceName, DeviceCode FROM cDevices cd
  INNER JOIN StudySites sts on sts.StudyID = cd.StudyID
  INNER JOIN Subjects sb on sb.StudySiteID = sts.StudySiteID
WHERE TypeCode = '{text='datapoint.standardvalue' fieldoid='DEVICETYPE'}' --
Must not have spaces around "="
  AND sb.SubjectID = '{subjectid}'
```

C#

```
public class DynamicSearchList
{
    public object Eval(object ThisObject)
    {
        DataPoint dpDSL = ((DynamicSearchParams)ThisObject).DataPoint;
        DataPoint dpDeviceType = dpDSL.Record.DataPoints.FindByFieldOID("DEVICETYPE");
//Control field
        string code = (CustomFunction.DataPointIsEmpty(dpDeviceType)) ? string.Empty :
dpDeviceType.Data;

        KeyValueCollection devices = new KeyValueCollection();

        switch (code)
        {
            case "C":
                devices.Add(new KeyValue("CATH STD", "Standard Occlusion Balloon Catheter"));
                devices.Add(new KeyValue("CATH STER", "Sterling Balloon Dilation Catheter"));
                devices.Add(new KeyValue("CATH PERIPH", "Peripheral Cutting Balloon Device"));
                devices.Add(new KeyValue("CATH ANGIO", "Imager II Angiographic Catheter"));
                break;
            case "S":
                devices.Add(new KeyValue("STENT DYN", "Dynamic (Y) Stent"));
                devices.Add(new KeyValue("STENT POLY", "PolyFlex Airway Stent"));
                devices.Add(new KeyValue("STENT TRACH", "UltraFlex Tracheobronchial Stent
System"));
                break;
            case "CL":
                devices.Add(new KeyValue("COIL GDC", "GDC Detachable Coil"));
                devices.Add(new KeyValue("COIL MATRIX", "Matrix2 Detachable Coil"));
                break;
        }

        //For this DSL CF, cannot return null. Only an object of KeyValueCollection type.
        return devices;
    }
}
```


Appendix D: The Development Utility

- ❑ Using the Custom Function Development Utility
- ❑ Setting Up the Utility
- ❑ General Configuration of the Development Utility
- ❑ Testing a Custom Function

Using the Custom Function Development Utility

To streamline development, Medidata offers the Custom Function Development Utility (“the Utility”, “the debugger”).

The Utility is a Microsoft Visual Studio Project where a user can write a custom function and execute it against an instance of Rave by connecting to its backend SQL database. The tool lets the user browse through a URL’s subject data and pick a data value to pass to a custom function, simulating the triggering of that function from an edit check or derivation.

The advantage this tool affords is twofold. First, developers can actually step through their code and use Visual Studio’s advanced debugging features to pinpoint bugs. Second, and most importantly, changes made to a custom function can be tested immediately, without having to re-paste the code into a Rave draft and publish and push a new CRFVersion.

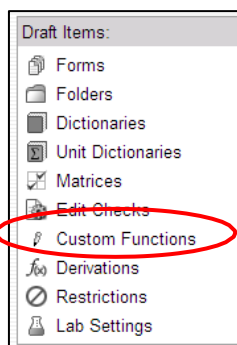
Use of the tool can be divided into three main tasks: initial, one-time setup; general configuration; and testing a particular custom function.

Setting Up the Utility

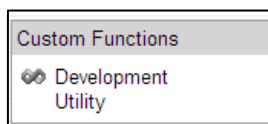
The debugger is provided as a zip-compressed Visual Studio project. Setup consists of extracting the project from the zip file onto into a local directory, opening the project, and testing the installation. The steps below need only be performed once per machine installation, regardless of how many custom functions will be written.

To Set Up the Utility:

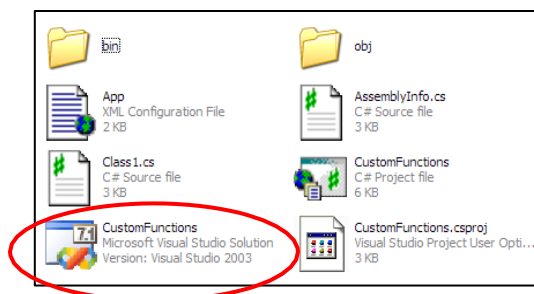
From within any Project Draft inside the Architect module, click the Custom Functions link on the sidebar.



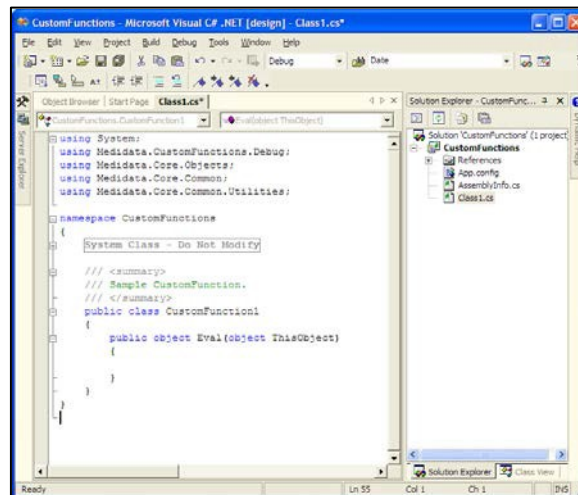
Click the Custom Functions Development Utility link on the sidebar.



Unpack the zip file downloaded (~1 MB) into a directory of your choice. The resulting file structure should look like this:

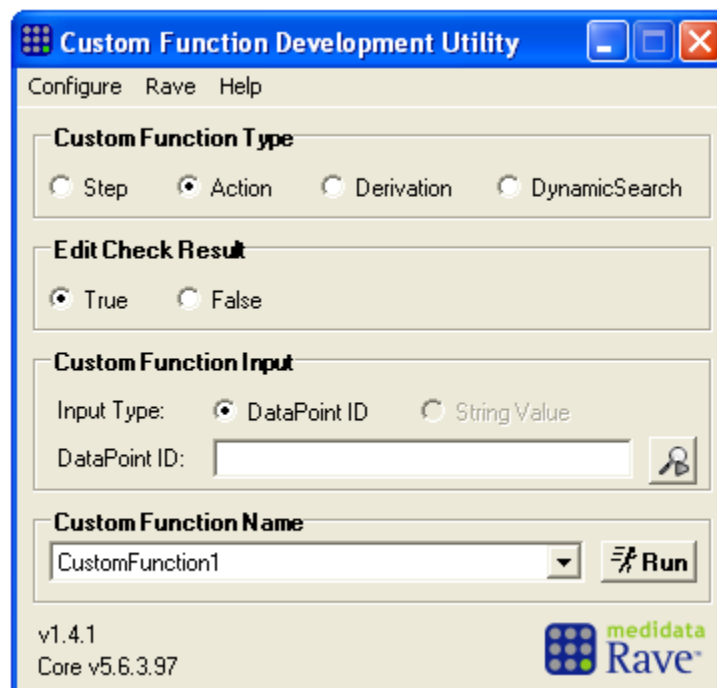


Open the “CustomFunctions” Visual Studio Solution File (see above). Your screen will look like this:



Custom function classes can be placed anywhere in the CustomFunctions namespace – usually right inside Class1.cs to start.

To test the installation, choose Start from the Debug menu, or press the F5 key. If the build process is successful, you will see the Utility appear. If the project does not build, correct any compile-time errors indicated by Visual Studio.

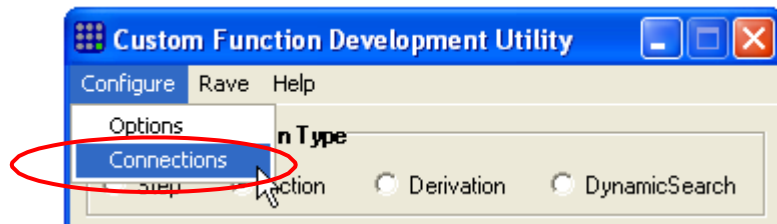


General Configuration of the Development Utility

Configuring the Development Utility is a two step process of specifying what Rave database it should connect to and how it should interact with it. The following screens will appear automatically upon first.

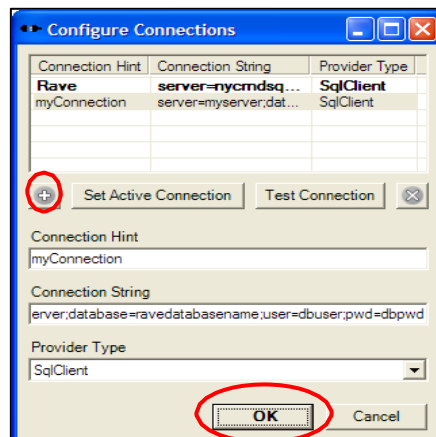
To Configure the Development Utility

From the Configure menu, select Connections.

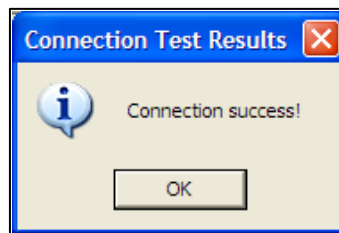


On the Configure Connections window that appears, click the plus button to add a new connection. Provide a

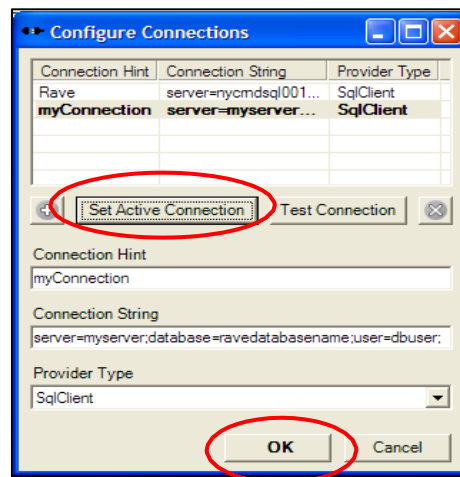
- ⇒ **Connection Hint** to give the database connection a name. Actual value does not matter – just used so you can keep track of your connections.
- ⇒ **Connection String** specifying which Rave database should be used in debugging custom functions. It should be in the form *server=myserver; database=ravedatabasename; user=dbuser; pwd=dbpwd*
- ⇒ **Provider Type**. Indicating the type of Rave database. Always select **SqlClient**, as Rave currently supports only Microsoft Sql Server 2005 databases.



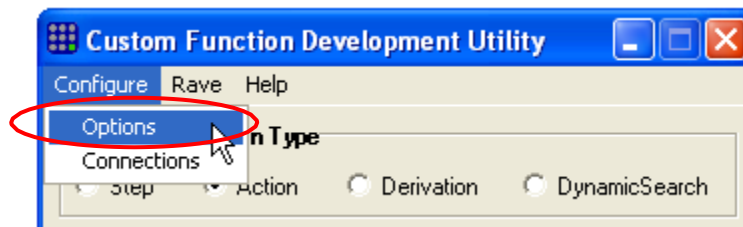
To confirm that a valid connection string has been entered, click Test Connection. The following message should appear:



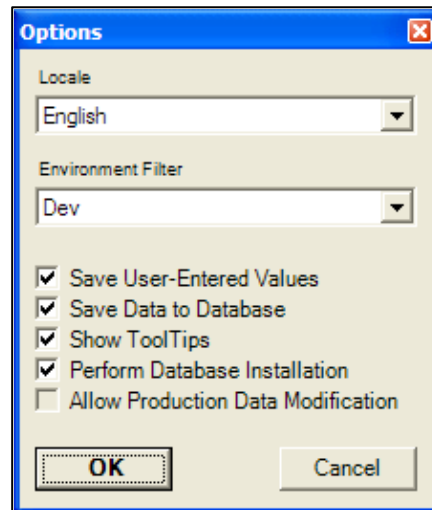
To use this connection when testing custom functions, click Set Active Connection. The connection will be shown in bold in the table at the top of the window. Finally, click OK to accept all changes and return to the main window.



From the main window, select Options from the Configure menu.



On the Options window that appears, select a Locale under which to run the debugger. Optionally, select an Environment Filter to see data from a particular auxiliary environment only. (By default, data from *all* environments will be presented when browsing.)



Use the check boxes at the bottom of the window to specify advanced options:

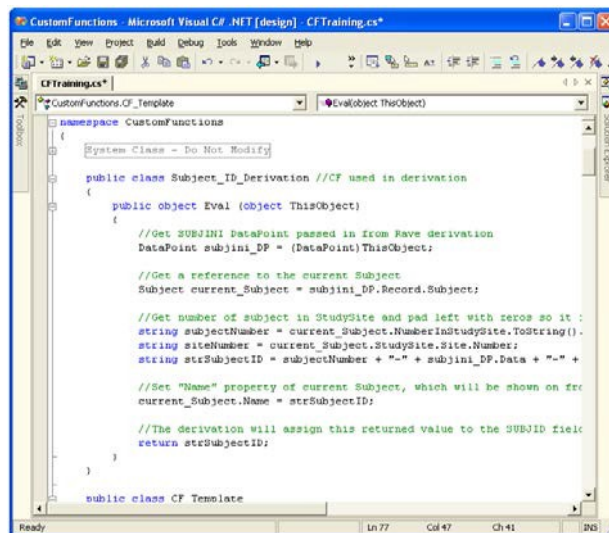
- ⇒ **Save User-Entered Values:** When this box is checked, the user can modify data when browsing through / searching for Data Points
- ⇒ **Save Data to Database:** When this box is checked, the system will save data modifications made by custom functions executed in the debugger.
- ⇒ **Show ToolTips:** Context sensitive help will be enabled when this box is checked.
- ⇒ **Perform Database Installation:** This is required for debugger localization.
- ⇒ **Allow Production Data Modification:** This can only be enabled by executing a specific stored procedure; the checkbox is disabled by default.

Testing a Custom Function

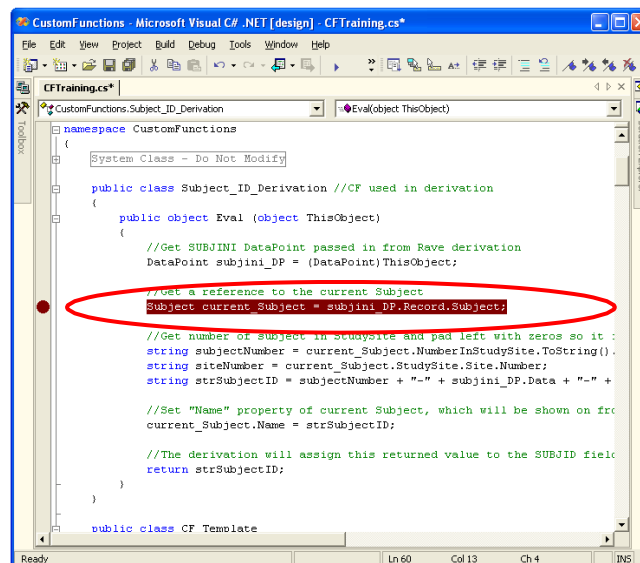
After the Utility's Visual Studio application has been installed and a database connection is specified, custom functions can be tested. This involves specifying the name and type of the function, as well as picking an input data value by browsing through, or searching for, subject data. From here, it is possible to run through the function code line by line, adding watch variables and using other Visual Studio debugging features.

To Test a Custom Function:

Place the code inside a class in the CustomFunctions namespace (in Class1.cs).

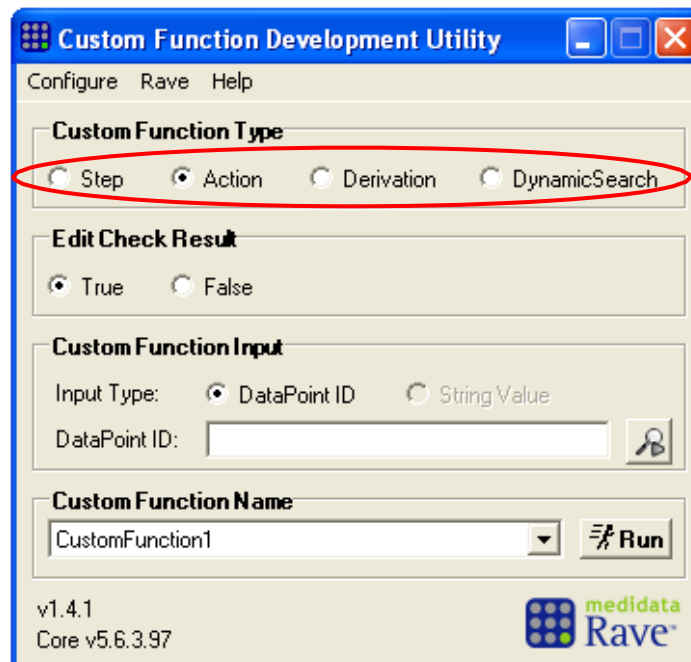


Be certain to insert a breakpoint on the very first line of the custom function so as to enable step-through later on. With the cursor on the first line, press Control+B.

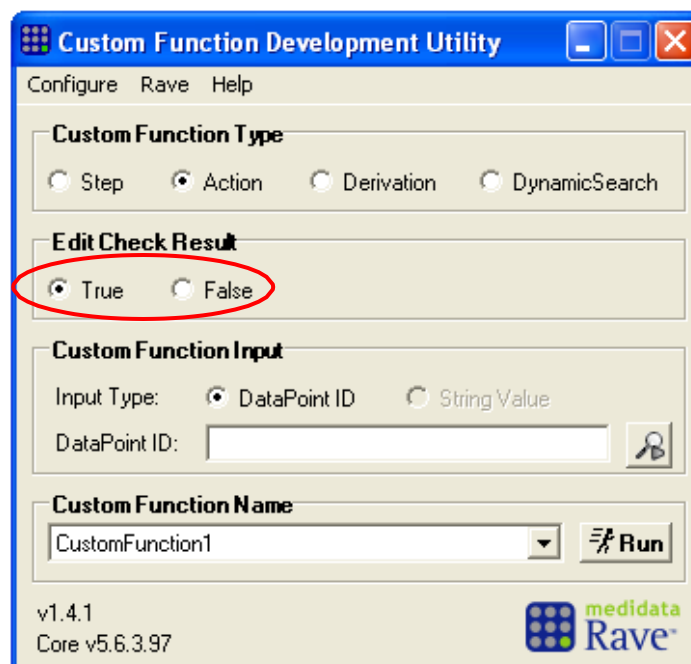


Select Start from the Debug menu.

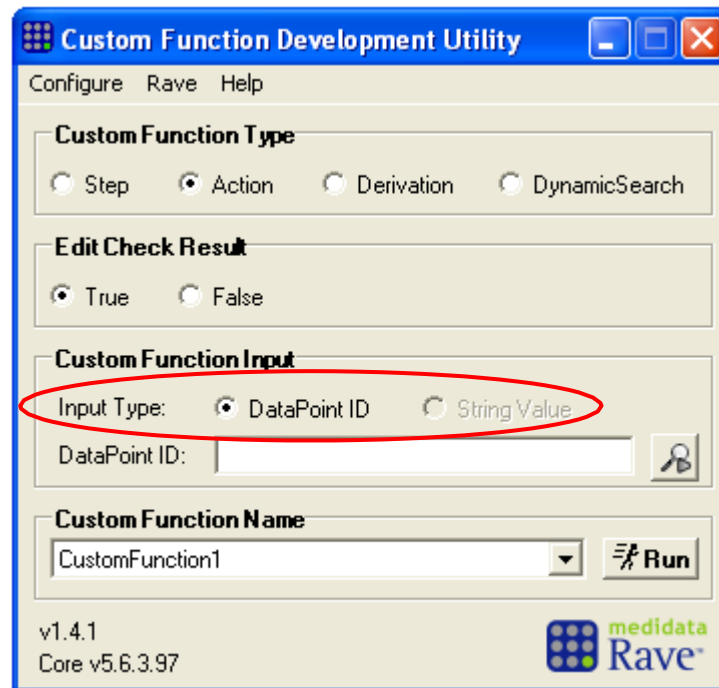
On the main debugger window, specify whether the code is meant to be invoked from an edit check step or action, or from a derivation. This will determine the type of object passed to the custom function.



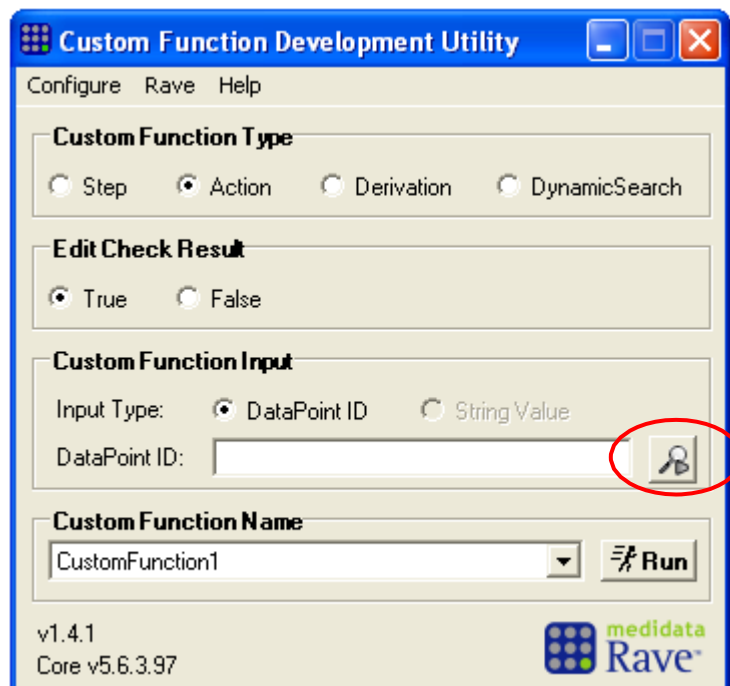
If the custom function is called from a check action, specify whether the `ActionResult` should be passed as `True` or `False`. Note that this field will be disabled unless `Action` has been specified above.



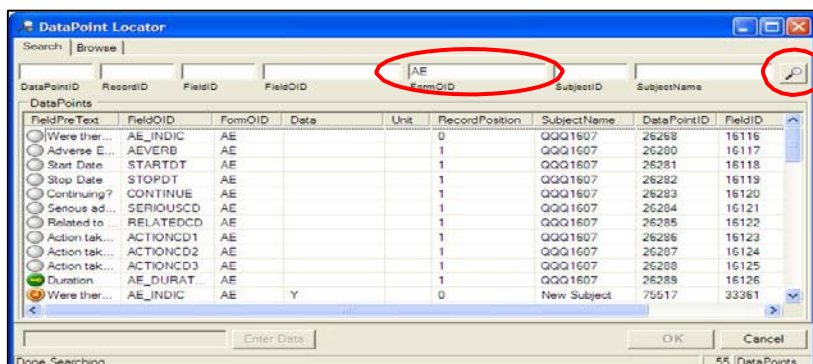
Under Custom Function Input, specify whether the function should be passed a Datapoint or a string. String Value is unavailable for check action custom functions.



To choose a Data Point that should be passed to the custom function, click the magnifying glass button.

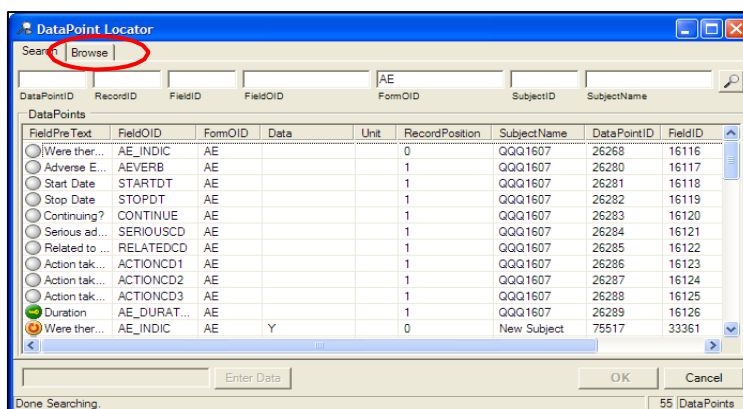


The DataPoint Locator window will pop-up with the Search tab loaded. To search for a Data Point, enter some combination of DataPointID, RecordID, FieldID, FieldOID, FormOID, SubjectID, and/or SubjectName, and click the magnifying glass. Results appear in a grid below with their relevant status icons.

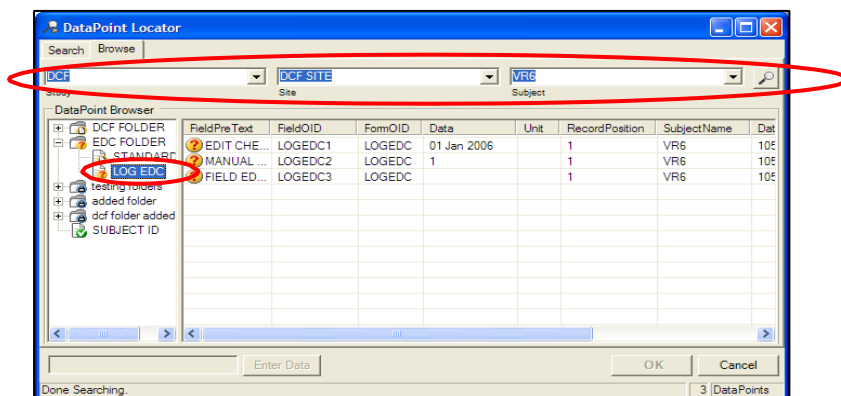


Alternatively, a Data Point can be selected by browsing. To do this, click the Browse tab in the DataPoint Locator window.

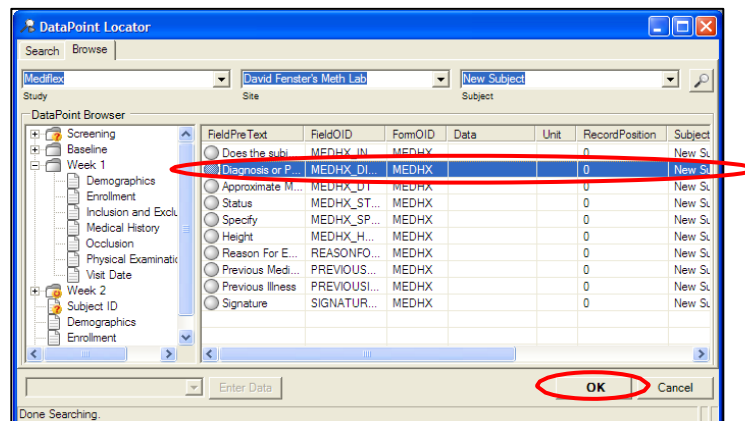
On the browse window, select a Study, Site and Subject, and click the magnifying glass



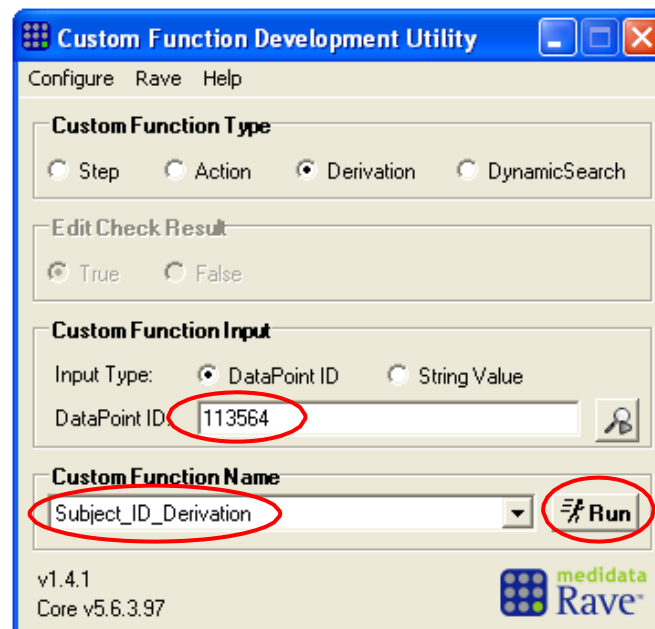
icon to start. Then, in the DataPoint Browser pane inside the window, select a form to display its Data Points.



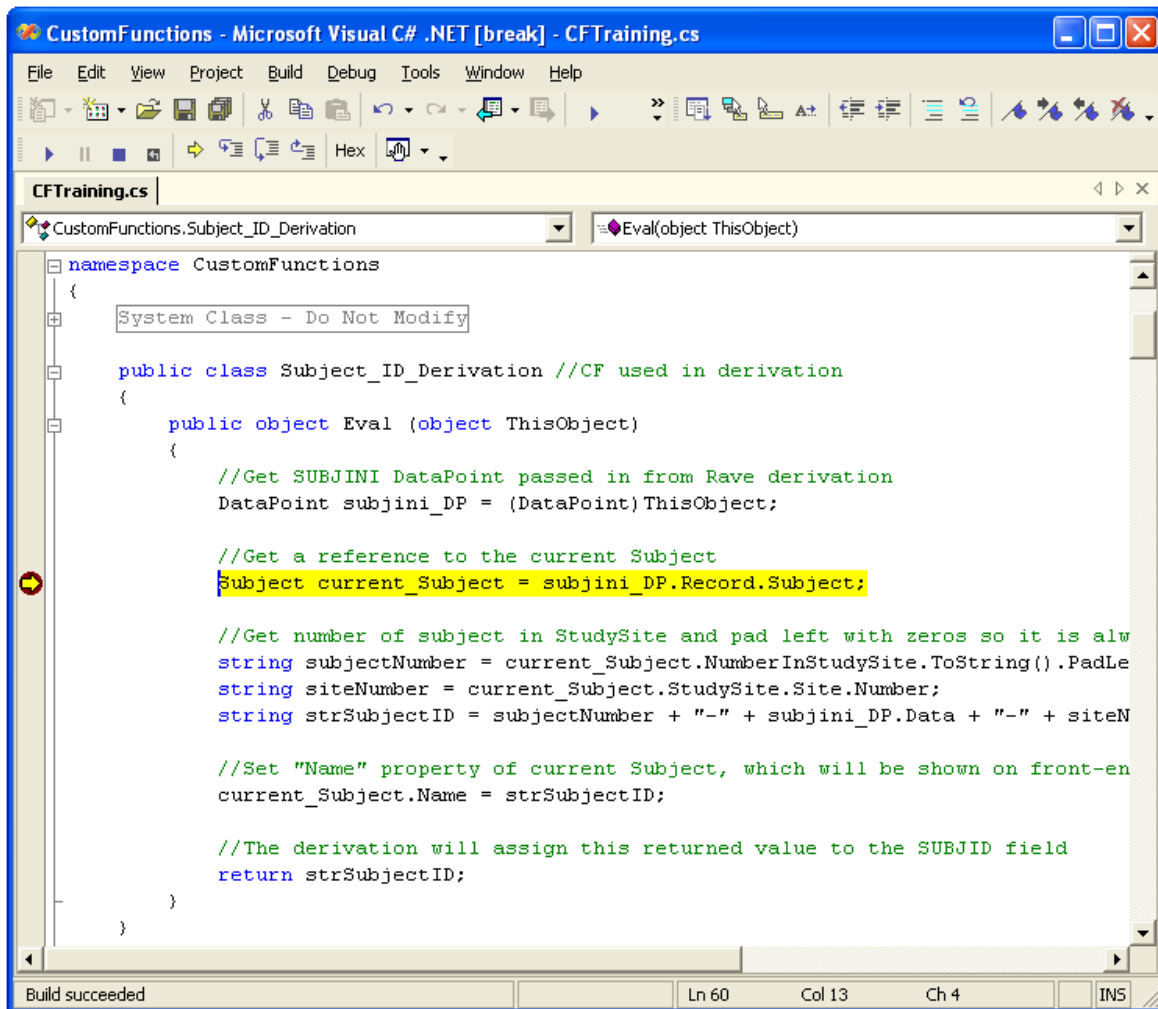
To select a Data Point from the results table (in either the Search or Browse areas), click on its row. Then click OK.



Once the locator window has closed, the main window's DataPointID field is populated based on what Data Point was just selected. To start testing and jump to the breakpoint set in the beginning, select a Custom Function Name, then click the Run button.



The debugger will step into your function's code and stop at the breakpoint you defined. Consult Microsoft Help for more information on adding watch variables and other Visual Studio debugging features.



Appendix E: Enhancements to Examples

- ❑ SAE Email
- ❑ Diagnosis Date Check
- ❑ Duplicate Diagnosis

SAE Email

```

const string YES = "1";
const string FROM = "rave@mdsol.com";
const string TO_DEV = "user_dev@mdsol.com"; const string TO_PROD =
"user_prod@mdsol.com"; const string TO_DEFAULT = "user_any@mdsol.com";

//Get SERIOUS DataPoint passed in from Rave
DataPoint dpSerious = ((ActionFunctionParams)ThisObject).ActionDataPoint;

//Only continue if this is a serious AE
if (dpSerious.Data != YES)
    return null;

Subject currentSubject = dpSerious.Record.Subject;

string emailSubject = string.Empty;
string body = string.Empty;

if (dpSerious.IsObjectChanged) //New
{
    emailSubject = String.Format("SAE Added! Site: {0}, Subject: {1}",
                                currentSubject.StudySite.Site.Number,
                                dpSerious.Record.Subject.Name);
    body += BodyForNewAE(dpSerious);
}
else //Updated
{
    emailSubject = String.Format("SAE Updated! Site: {0}, Subject: {1}",
                                currentSubject.StudySite.Site.Number,
                                dpSerious.Record.Subject.Name);
    body += BodyForUpdatedAE(dpSerious);
}

string to;
//Chose recipient based on current environment
switch (currentSubject.StudySite.Study.Environment.ToUpper())
{
    case "DEV":
        to = TO_DEV;
        break;
    case "PROD":
        to = TO_PROD;
        break;
    default:
        to = TO_DEFAULT;
        break;
}
Message.SendEmail(to, FROM, emailSubject, body);
return null;
}

//Composes the body of the email message for NEW serious AEs.
//Retrieves values of every field on the current AE log line.
string BodyForNewAE (DataPoint dpSerious)
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    sb.AppendFormat("A new serious AE has been recorded by {0} {1}",
                    dpSerious.Interaction.TrueUser.Login, Environment.NewLine);
    DataPoint dp;

```

```
        for (int i=0; i < dpSerious.Record.DataPoints.Count; i++)
        {
            dp = dpSerious.Record.DataPoints[i];
            sb.Append(string.Format("{0}: {1}", dp.Field.PreText,
            dp.UserValue())); sb.Append(Environment.NewLine);
        }
        return sb.ToString();
    }

    //Composes the body of the email message for UPDATED serious AEs.
    //Retrieves values for fields that have been updated.
    string BodyForUpdatedAE (DataPoint dpSerious)
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        sb.AppendFormat("An existing serious AE has been updated by {0} {1}",
            dpSerious.Interaction.TrueUser.Login,
            Environment.NewLine); sb.Append("The following fields have been
            updated:"); sb.Append(Environment.NewLine);

        DataPoint dp;
        for (int i=0; i < dpSerious.Record.DataPoints.Count; i++)
        {
            dp =
            dpSerious.Record.DataPoints[i];
            if (dp.IsObjectChanged)
            {
                sb.Append(string.Format("{0}: {1}", dp.Field.PreText,
                dp.UserValue()));
                sb.Append(Environment.NewLine);
            }
        }
        return sb.ToString();
    }
}
```

In order for a custom function containing multiple methods to work, it must be pasted into Rave as shown in the dashed-rectangle above.

Diagnosis Date Check

Edit Check 1 of 2:

If ENRLDT in Enrollment in Screening with record position 0 IsPresent then... execute the "05 Diagnosis Date Check (good)" custom function

Check Steps

Type	Step	Edit
Data Point (Data Point)	Screening>Enrollment>ENRLDT>ENRLDT>0>...>...>None	
Check Function	IsPresent	

Add Check Step

Check Actions

Data Point	Action	Edit
Screening>Enrollment>ENRLDT>ENRLDT>0>...>...	Custom Function:05 Diagnosis Date Check (good)	

Add Check Action

Edit Check 2 of 2:

If MEDHX_DT in Medical History in Screening IsPresent then... execute the "05 Diagnosis Date Check (good)" custom function

Check Steps

Type	Step	Edit
Data Point (Data Point)	Screening>Medical History>MEDHX_DT>MEDHX_DT>...>...>None	
Check Function	IsPresent	

Add Check Step

Check Actions

Data Point	Action	Edit
Screening>Enrollment>ENRLDT>ENRLDT>0>...>...	Custom Function:05 Diagnosis Date Check (good)	

Add Check Action

Custom Function code:

```
const string FIELD_OID = "MEDHX_DT";
const string FORM_OID = "MEDHX";
const string FOLDER_OID = "SCRN";

const int DAYS_INTERVAL = 180;
const string QUERY_TEXT = @"Diagnosis date must be before enrollment date, but
                           not more than 180 days prior.";

const int MARKING_GROUP_ID = 1;
const bool ANSWER_ON_CHANGE = false;
const bool CLOSE_ON_CHANGE = false;

ActionFunctionParams afp =
(ActionFunctionParams)ThisObject; DataPoint dpEnroll =
afp.ActionDataPoint;

Object obj = dpEnroll.StandardValue();
DateTime enrollment_DT = (obj is DateTime) ? (DateTime)obj : DateTime.MinValue;

Subject curr_subj = dpEnroll.Record.Subject;
DataPoints dpsDiagnosis = CustomFunction.FetchAllDataPointsForOIDPath(FIELD_OID,
                                                                    FORM_OID, FOLDER_OID, curr_subj);

for (int i=0; i<dpsDiagnosis.Count; i++)
{
    DataPoint dp = dpsDiagnosis[i];
    if (!dp.Active) continue;
    bool doQuery = false;
    if (enrollment_DT != DateTime.MinValue
        //Do not deal with dates if both dpEnroll and dp didn't change
        && (dp.IsObjectChanged || dpEnroll.IsObjectChanged)
        && dp.StandardValue() is DateTime)
    {
        DateTime diagnosis_DT = (DateTime)dp.StandardValue();
        TimeSpan interval = new TimeSpan(DAYS_INTERVAL, 0, 0, 0, 0);
        doQuery = (diagnosis_DT < enrollment_DT.Subtract(interval)
                    || enrollment_DT <= diagnosis_DT);
    }
    //Do not perform query if both dpEnroll and dp didn't change
    if (dp.IsObjectChanged || dpEnroll.IsObjectChanged)
        CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID,
            ANSWER_ON_CHANGE, CLOSE_ON_CHANGE, dp, doQuery,
            afp.CheckID, afp.CheckHash);
}
return null;
```


Duplicate Diagnosis

```

const string FIELD_OID = "MEDHX_DIAG";
const string FORM_OID = "MEDHX";
const string FOLDER_OID = "SCRN";
const string QUERY_TEXT = "Duplicate diagnosis. Please, correct.";
const int MARKING_GROUP_ID = 1;
const bool ANSWER_ON_CHANGE = false;
const bool CLOSE_ON_CHANGE = false;

ActionFunctionParams afp = (ActionFunctionParams)ThisObject;
//Passed is a standard field from another form (used to enable inactivation)
DataPoint dpStart = afp.ActionDataPoint;

//First get all datapoints by which records are compared
DataPoints dpsMEDHX = CustomFunction.FetchAllDataPointsForOIDPath(FIELD_OID, FORM_OID,
    FOLDER_OID, dpStart.Record.Subject);

//Remove inactive datapoints before sorting
DataPoints dpsMEDHX_Active = new DataPoints();
for (int d=0; d<dpsMEDHX.Count; d++)
{
    DataPoint dp = dpsMEDHX[d];
    if (dp.Active)
        dpsMEDHX_Active.Add(dp);
}

//Put all datapoints above to a sorted arraylist
//to ensure that all dps are sorted by record
dpsMEDHX = GetOrderedDPs(dpsMEDHX_Active);

DataPoint dpPrev = null;
DataPoint dpNext = null;

int i=0;
while (i <= dpsMEDHX.Count - 2)
{
    dpPrev = dpsMEDHX[i];
    //Close the query initially
    CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID,
ANSWER_ON_CHANGE, CLOSE_ON_CHANGE, dpsMEDHX[i], false, afp.CheckID, afp.CheckHash);
    for (int j=i+1; j<=dpsMEDHX.Count-1; j++)
    {
        dpNext = dpsMEDHX[j];
        i++;
        if (String.Compare(dpPrev.Data, dpNext.Data, true) == 0)
            CustomFunction.PerformQueryAction(QUERY_TEXT, MARKING_GROUP_ID, ANSWER_ON_CHANGE,
CLOSE_ON_CHANGE,dpNext, true, afp.CheckID, afp.CheckHash);

        else
            break; //Start again from next log
    }
}

```

```
return null;
}

DataPoints GetOrderedDPs (DataPoints dps)
{
    System.Collections.ArrayList arlDPs = new System.Collections.ArrayList();

    for (int i = 0; i<dps.Count; i++)
    {
        //Skip just inactivated record
        if (dps[i].Active)
            arlDPs.Add(dps[i]);
    }

    arlDPs.Sort(new DataPointComparer());

    //Now, convert arlDPs back to DataPoints
    dps.Clear();
    for (int i = 0; i<arlDPs.Count; i++)
    {
        dps.Add((DataPoint)arlDPs[i]); //Adding in sorted order
    }
    return dps;
}

private class DataPointComparer : System.Collections.IComparer
{
    //This Comparer sorts DataPoints by Data (strings) and RecordPosition
    int System.Collections.IComparer.Compare(object x, object y)
    {
        DataPoint d1 = (DataPoint)x;
        DataPoint d2 = (DataPoint)y;
        string s1 = d1.Data;
        string s2 = d2.Data;
        int compareResult = String.Compare(s1, s2);
        if (compareResult != 0)
            return compareResult;
        else //if (s1 == s2) compare by record position
            return
                d1.Record.RecordPosition.CompareTo(d2.Record.RecordPosition);
    }
}
```